commodore

presents

The Best of The Transactor Volume 2

COMMODORE BUSINESS MACHINES LIMITED
3370 PHARMACY AVENUE, AGINCOURT, ONTARIO M1W 2K4
TELEPHONE (416) 499-4292 – CABLE ADDRESS: COMTYPE
TELEX NUMBER 06-525400

The Best of The Transactor Volume 2 Contains:

The Transactor #1

Watching a Cassette Load 1
Decimal/Hex Conversion 2
The LIST Chain 3
16/32 K PET Notes 8
Review: BASIC For Home Computers 10

The Transactor #2

Bits and Pieces 11
    Crash Your PET 11
Merging PET Programs 12
PET to Teletype Interface 14
Additional I/O Interface 17

The Transactor #3

PET Variable Storage 19
IEEE Bus Handshake Routine 27

The Transactor #4

Computed GOTO 31
More PET Quirks 33
Indenting Program Text 34
IFless Decisions 35
A Fast Sort 37
Disabling The STOP Key 38

The Transactor #5

Bits and Pieces 39
Memory Expansion. Cost $0.00 41
?LOAD ERROR 45
The IEEE-488 Bus 49
Screen Print Routine 52
The DRIP Instruction 54
FOR/NEXT Loop Structure 56
Attention Multi-Peripheral Users 60

The Transactor #6

Inside The 2040 Disk Drive 61
Printer Formatting 63
Bits and Pieces 64
    Bullet-Proof Input 64
    Floating Binary 65
Infinitely Long PET Programs 66
Screen I/O 67
Random Access File Indexing 70

The Transactor #7

;Reference Info 200

Auto Repeat Routine 75
Block Get Routine 76

The Transactor #8

Bits and Pieces 79
Re-DIMensioning Arrays 79
Dynamic LOADing 80
Cursor Positioning 81
SAVing With The Monitor 82
Transactor - A Philosophy 83
The POP Instruction 84
Disk Merge 85
Supermon 1.0 86
RS232-C: An Overview 94
Program Plus 101

The Transactor #9

Bits and Pieces 109
Printer Tabbing 109
Machine Language I/O 110
Instring Utility 111
IEEE-488 Logic Analyzer 112
Cross-Reference 114
Better Auto-Repeat 119
Case Converter 120
RESTORE DATA Line Program 122

The Transactor #10

Bits and Pieces 123
REMAINDER$ 123
BASIC 4.0 Prelim Note 123
ID Changer 124
Printer ROMs 124
Print Speed-Up 124
Controlling Garbage Collections 126
True ASCII Output 127
2040 Disk Buffer I/O Routine 130
Filestatus 147
More On Screen Print 148

The Transactor #11

Bits and Pieces 149
WordPro and The NEC Spinwriter 149
Soft Disk Device Number 150
CEAB 150
TPUG Minutes 150
Supermon Notes 151
PET Sound 151
Disk Data Salvaging 153
Text Editor 154
Card Printing Utility 157
Simple 8010 Modem Programs 164
PET/CBM Bibliography 166

The Transactor #12

Contest Winners 169
Bits and Pieces 170
    Exclusive OR 170
    BASIC Differences 171
    Screen Loading 171
PET BASIC Labelling 172
BASIC 4.0, DOS 2.0 & REL Records 178
Converting BASIC 2.0 to BASIC 4.0 186
DOS 2.0 Commands from BASIC 2.0 190
The PET NMI Vector 192
Fun With WAIT Statements 194
8032 Control Characters 196
More on 80 Columns 198
Subscription Form 199

Reference Information

BASIC 1.0, 2.0 Memory Maps 200
DEC, HEX, ASCII, BASIC, 6502 Chart 209
BASIC 1.0, 2.0 & 4.0 Memory Maps 210
BASIC Entry Points 212
6502 Instruction Set 213
DS & DS$; Disk Status Variables 213
ST; The Status Word 213
8032 Control Characters 214
40 & 80 Column Screen Addresses 214
POKE, CHR$ and RVS Key Codes 216
BASIC Tokens 216
BASIC Commands and Statements 216
External Connector Data 220
Addressing Modes 224
Hex Conversion Table 225
Standard ASCII Codes 226

commodore

comments and bulletins
concerning your
COMMODORE PET tm

The Transactor

Vol. 2
BULLETIN # 1
May 31, '79

PET tm is a registered trademark of Commodore Business Machines Inc.

Watching a cassette load

Jim Butterfield, Toronto

It may not be to useful, but it's very satisfying to watch a program coming in from cassette tape. Much of what comes in will look like gibberish, since the program contains things like pointers, flags and tokens. But it's interesting to see and here's how you can do it.

Step 1: Load any Basic program on cassette 1. The program doesn't matter; the LOAD activity sets up certain internal things that will help us.

Step 2: Set up the cassette with any Basic program ready to load. A short one would be good; that way you may catch the whole program on the screen. But any Basic program will do.

Step 3: Set graphic mode with POKE 59468,14. This may help you spot a few recognizable pieces of your program.

Step 4: Give SYS 62894. PET will ask you to press PLAY. Do so, and in twenty seconds or so, PET will report FOUND... and stop.

Step 5: Clear the screen so you'll get a better view of the program as it comes in. Now move the cursor down to a few lines from the bottom of the screen.

Step 6: Enter POKE 636,128 : POKE 638,132 : SYS 62403

Step 7: Sit back and watch the program load to the screen. You won't be able to RUN it, of course, since it's in the wrong part of memory... but isn't it fascinating to watch?

Decimal/Hex conversion: a few techniques

Jim Butterfield

If you stay clear of machine language, you'll never need to explore the mysteries of Hexadecimal numbering. If you do need to use this kind of numbering scheme, you'll often want to convert back and forth between decimal and hexadecimal. For example, a program contains a SYS(2345); and you want to use the Machine Language Monitor to see what's in that part of memory. But the MLM wants the address in Hex ... how to convert?

Most of these techniques can be readily done with a pocket calculator, or with a program. But when your calculator battery has gone dead, and your PET is already loaded with a different program that you don't want to disturb, it's handy to work it all out using immediate statements.

Converting Decimal to Hexadecimal.

My favorite quick method is this (let's convert 2345 as an example):

```
X = 2345/4096 : FOR J=1 TO 4 : ? INT(X); : X=(X-INT(X))*16 : NEXT J
```

.. and out come the numbers 0 9 2 9, representing hex 0929. If you get numbers greater than 9, remember that 10 is written as A, 11 as B, and so on up to 15 as F.

Converting Hexadecimal to Decimal.

This is a simple matter of multiplying the previous total by 16 and adding the new digit. To convert hex 0929 back to decimal we type:

```
?  ((0*16 + 9)*16 + 2)*16 + 9
```

ana we get our original value of 2345. If you don't like brackets, you could try the alternative:

```
? 0*4096 + 9*256 + 2*16 + 9
```

with the same result. In the example, the leading zero can be dropped from the calculation, of course.

Gilding the lily.

You really don't need to dress up immediate statements any more than is shown above. In programs, you'll probably want the value to print in a more classy manner - with the alphabetics already done.

The easiest way is a variant of IF X>9 THEN PRINT CHR$(X+55); but if you like to baffle your friends with obscure coding you can try either or both of these:

```
  X=2346/4096:FORJ=1TO4:Y=INT(X):?CHR$(Y+55+7*(Y<10));:X=(X-Y)*16:NEXTJ

Z=0: X$="092A":FORJ=1TO4:Y= ASC(MID$(X$,J)):Z=Z*16+Y-48+7*(Y>57):NEXTJ:?Z
```

The LIST Chain

One of the most often used commands to be executed directly from the keyboard is LIST: a most fundamental function as it allows us to observe the contents of our BASIC and proceed to implement the screen editor, a feature of the PET that most of us have taken for granted. This very powerful programming tool permits deletion, insertion and alteration of lines and characters. But as all this occurs, PET is busy doing some rather extensive internal housekeeping; checking available space, updating FRE(0), manipulating pointers in zero page and a number of other things.

Of these tasks PET performs for itself, it also creates the LIST chain, a function of equal importance to PET and User. As a line of BASIC is completed, PET inserts three extra bytes of information which it uses to keep track of where the line ends and also where the next line begins. The best way to observe this is to load 'View', one of the machine language programs which appeared in Transactor 10, Volume 1. Proceed as follows:

1. LOAD and RUN View. This will set up the machine language for View in the second cassette buffer.

2. SYS64824. This will clear out the BASIC memory space but will not affect the second cassette buffer.

3. POKE 849,4 This will cause View to display page 4 of memory which is the first block or 'page' of BASIC memory space. (BASIC begins at Hex Ø4ØØ or decimal 1Ø24 which is 4x256.)

4. SYS826 The View program should now be operating and displaying page 4 at the top of the screen. The display should consist of mostly '$' signs representing empty space.

Preceding the '$' signs you should see three '@' signs. The '@' sign represents a zero (try POKE 33400,0). The first '@' or zero in location 1024 is a dummy end-of-line character. The next two zeroes represent the first pointer to the next line of BASIC but since they are zeroes this indicates to PET that nothing exists beyond this point. The three '@'s are automatically placed at the end of the last line of BASIC. The first '@' or zero is automatically placed at the end of every line of BASIC to indicate, of course, 'end-of-line'.

The three zeroes will not stay zeroes for long as we are about to enter into PET the following:

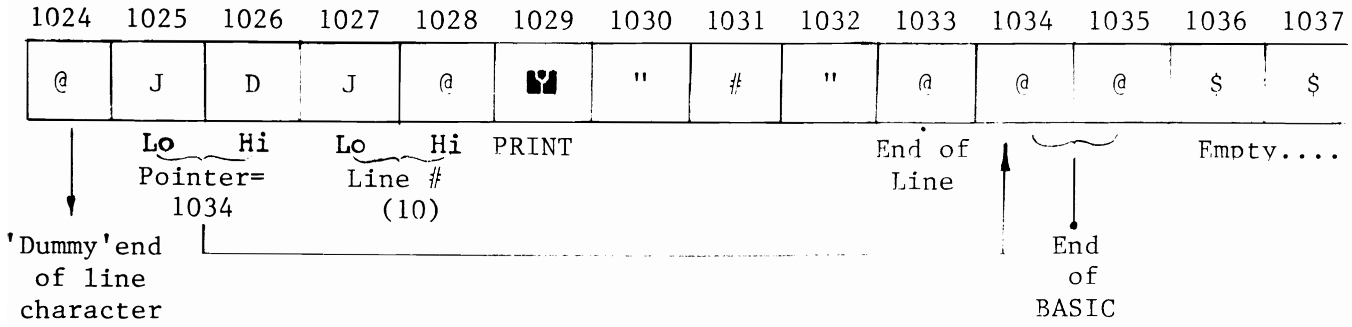
```
10?"#"     (without spaces)
```

Upon hitting 'RETURN' you should notice the top line of the screen change. The first character will still be an '@' representing our dummy end-of-line. (As a rule, location 1024 will always be a zero unless POKEd by the User.) The next two characters should be, in order, a 'J' and a 'D' where J=10 and

D=4. These represent the low order and high order bytes (respectively) of the pointer to next line of BASIC. But just exactly what do they point at? Since these numbers are in hexadecimal, the high order byte is used as a multiplier of 256 and the low order byte is multiplied by 1 and added to give us the decimal byte address. In this case the result will be:

$$\begin{aligned} & P = (D \times 256) + (J \times 1) = (D \times 256) + J \\ \text{OR} \quad & P = (4 \times 256) + (10 \times 1) = 1024 + 10 \\ & \qquad = 1034 \end{aligned}$$

If we start counting at 1024 (the 'HOME' position) across 10 character spaces to 1034, we find ourselves at the byte which our pointer points at (Figure 1.0). Since the only existing program consists of line 10, this byte will be a zero as is the following byte.



1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037
@	J	D	J	@	Y	"	#	"	@	@	@	$	$

Figure 1.0

These indicate end-of-valid-BASIC. Notice also the preceding byte which indicates end-of-line.

Getting back to our pointer, immediately following should be another 'J' and an'@' (Figure 1.0). These represent the low and high order bytes, respectively, of the line number. These are also in Hex such that:

$$\begin{aligned} & L.N. = (@ \times 256) + (J \times 1) = (@ \times 256) + J \\ \text{OR} \quad & L.N. = (0 \times 256) + (10 \times 1) = 0 + 10 \\ & \qquad = 10 \end{aligned}$$

...which is of course the line number of the only existing BASIC so far.

The next character on the top line is a reverse field 'Y' or the token for 'PRINT' which is 153. (see table following) The remaining characters are self explanatory.

Now let's get a little deeper and enter the following extra code:

```
400?"!"
```

Before hitting 'RETURN' watch closely the two last '@' characters (locations

1034 and 1035). Now hit 'RETURN' and they will change to an 'S' and a 'D' or a 19 and a 4 which equals 1043 (4 x 256 + 19). If we count across to 1043 we find ourselves once again at the second last '@' character indicating end-of-BASIC (Figure 1.1). Recall that our first pointer in locations 1025 and 1026 pointed at 1034. Therefore the first line pointer points at the next line pointer which in turn points at the next line pointer and so on to the end of your BASIC program. Sounds simple doesn't it? Well it is! This is the LIST Chain and PET employs these pointers to execute commands such as:

1. LIST
2. LIST-500
3. LIST500-5000
4. RUN20
5. GOTO500
6. GOSUB1000

When implementing these commands, either directly or under program control, PET immediately jumps to the pointer at 1025 and 1026 and stores it. PET then examines the following two bytes (1027 and 1028) which make up the line number and compares them against the given argument. If none is given the comparison is of course unnecessary. In the case of LIST 500-5000, PET will first compare the line number bytes with 500. If the test yields a "less than", PET recalls the pointer bytes and uses their values to jump to the next pointer. This new pointer is stored in place of the first and the above procedure is repeated until an "equal" or "greater than" test result is obtained. PET then begins LISTing by 'PRINTing' out the converted-to-decimal line number followed by a space followed by the text belonging to that line number. Text continues 'PRINTing' until the zero end-of-line character is detected. PET halts here, recalls the pointer last stored and makes the jump to the next pointer. This pointer is again stored and the line # bytes are examined...but this time compared to the second argument; 5000. If a greater than result occurs the LIST procedure terminates. Otherwise PET continues to:

a) display text while testing for Ø
b) recall pointer in storage
c) jump to new pointer and store
d) examine line # if so instructed
e) continue or terminate

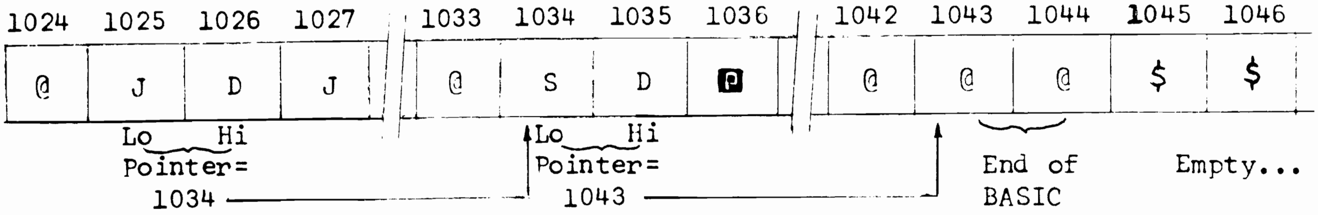


Figure 1.1

Some of you may now be asking....

> "Why does PET do all that back-tracking to recall a pointer value which is simply the address of the byte immediately following the zero end-of-line character? Why doesn't PET just test for zero and have the line # bytes i- the following two locations. This would free up those extra two bytes used by PET every time it creares a line pointer.... "

The answer is speed. The difference would not be noticed so much with LIST or even LIST with parameters. The real decrease in speed would occur upon execution of GOTO or GOSUB instructions. PET would have to test every byte for a zero (starting at 1024) and then, of course, look at the line #. This testing for zero would take some time, especially if the average number of bytes per line were up around 30 or 35. Coupled with the number of GOTOs and GOSUBs in your program, BASIC execution speed would be considerably slower. Using the present method, PET skips across the pointers like a frog across the lily pads (only it won't eat up your bugs at the same time).

Insertion and Deletion

When we program a line of text that is to fall between two existing lines, PET must know where to put it. We won't discuss how PET splits the existing code; that's another subject. PET jumps along looking at line numbers until an "in-between" condition is satisfied. Existing text is moved up from the right point exactly far enough for the code to be inserted. The pointers (line pointers and pointers in zero page) are updated and control is returned to the keyboard.

For deletion of lines, PET simply finds the line and "squeezes" it out. Pointers are updated....operation complete.

Try experimenting with the View program to watch how PET handles its editing.

Assuming that View is still running, type in NEW but before hitting 'RETURN', record the second and third characters of page 4. They are reset by a 'NEW' but notice how the rest of memory still exists. If these locations are POKEd back to what they were, the program can now be LISTed once again. However, zero page pointers were all reset by NEW also. Editing or assigning variables to values will cause a crash (and a rather interesting one at that) so do not try a RUN. About the only way to 'SAVE' it is to use the UNLIST routine.

What You Can Do

Some interesting results can be obtained by manipulating these line pointers; particularly locations 1025 and 1026. If we POKE1025,0 we've essentially aimed the pointer at location 1024 (4 x 256 + 0). If a LIST is executed, PET will pick up the first pointer, display the text and jump to 1024. Since 1024 is a zero and is now followed by a zero, PET is fooled into thinking end-of-BASIC....try it!

Similarly, if we POKE1025,1 we have aimed the first pointer at itself! Now try a LIST. By the same token you can point that first pointer (or any pointer for that matter) at any other pointer in BASIC; but only at pointers ....anything else and PET will crash. By doing this manipulation of pointers you can have LIST-inhibit on any lines you wish without affecting RUN (so long as you do not use RUN with arguments that lie within the LIST-inhibited lines). Be careful though....mistakes can be hazardous!

Now let's have some fun with View. Type the following on a clear screen near the bottom (View, of course, will not clear):

```
FOR T = 32 TO 135: POKE849,T: FOR R = 1 TO 250: NEXT R,T
```

Hit 'RETURN' and next month we'll discuss how PET stores variables.

16/32 K PET Notes - Collected by Jim Russo

The Operating System of the 16/32K PET is about 90% the same as the 8K PET, but has been re-assembled so that almost everything is in a slightly different place in memory than it used to be. Most bugs have been fixed and some limitations removed.

Any pure BASIC program (no PEEK, POKE, SYS, or WAIT) that works on an 8K PET should also work on a 16/32K PET. POKing and PEEKing screen memory (32768 to 33767) is still safe but POKing the operating system (below 1024 decimal) or using an operating system PEEK value to make a decision could be hazardous. Other programs can be made to work properly if references to RAM and ROM locations are changed. Commodore's 16/32K PET manual contains a memory map for pages 0. 1 and 2. A list of new ROM addresses follows. These two lists should contain the information needed in most cases.

Some Hardware Differences:

- The character generator ROM has been revised so that when lower case mode is selected, upper and lower case are interchanged. That is, the ' SHIFT' key must be used to obtain an upper case character. Also, 8K programs using lower case that are run on a 16/32K PET will display all lower case as upper case and vice versa.

- The signal which blanks the video on the 8K is not connected on the 16/32, so POKE 59409, 52 no longer works. The ROM routines still reference this address but the required hardware seems to have been omitted.

Summary of Differences:

- The bug in TI has been fixed. Now every 623rd. interrupt doesn't increment TI. Also, TI is allowed to count 1/60 sec. too far: 24ØØØØ precedes ØØØØØØ.

- Execution (of at least some code) is faster due to more efficient coding and better use of zero page. PRINT (to screen) is faster because extra code to maintain separate POS pointer has been eliminated. Also, screen snow and 'scroll - up flash' has been eliminated thanks to dynamic screen RAMs.

- Standard typewriter operation i.e., shift for upper case.

- RND (0) returns a number derived from interval timers.

- OPENing more than 10 files no longer crashes system.

- OPEN statement correctly sets "current tape buffer pointer".

- Machine Language Monitor included in ROM. BRK vector is initialized to monitor. 'L' and 'S' (LOAD and SAVE from monitor) have new syntax.

- NMI vector no longer tied to +5v. NMI is initialized to BASIC "Warm Start".

- Data file write error corrected. The Tape Output routines now wait 2/3 second after turning on motor before beginning to write tape leader. 8K PET waited 13 ms. on drive 1, 57 ms. on 2.

- Cursor home, left, right, up, down are now tracked properly by the POS function. This causes apparent differences in the TAB function which subtracts POS from its argument to determine the number of spaces needed.

- SPC (0) corrected.

- When output is directed to an alternate device, the ASCII space code $20 is used for all BASIC supplied forward spacing. 8K used $1D.

- Screen blanking (POKE 59409,52) no longer available, however, the scroll routine still uses it as if it did.

- PEEK is no longer restricted.

- Array dimensions now as high as 32767 (used to be 256).

- The memory expansion port uses a different connector.

- Spaces no longer allowed in keywords (e.g. GOSUB cannot be coded as GO SUB).

- POKE and PEEK can now be used in the same line (i.e., POKE 8000, PEEK (9000) now works).

- ST (the status word) if used, must be tested before input of file data.

- Most ROM routines and RAM addresses have changed.

	8K	16/32K
INTFLP	D278	D26D
FLPINT	DØA7	DØ9A
CHRGOT	ØØC2	ØØ7Ø
WARM START	C38B	C389
FLOATING AC	00B0-00B6	005E-0064

- BASIC input buffer is no longer in zero page so programs which used many free locations in this area must be re-written.

- The decoding of screen memory now uses A11 (address buss line 11). Addresses 88ØØ-8FFF (34816 to 36863 decimal) no longer address screen memory.

Review: Basic for Home Computers John Wiley & Sons, Inc.
by Bob Albrecht, LeRoy Finkel, and Jerald R. Brown.

A good teaching book that deals with MICROSOFT^{TM} Basic fundamentals. This is the type of Basic that PET has, and readers will find it a suitable introduction to PET programming.

The book is a self-teaching guide, which means that on almost every paragraph you are asked to "fill in the blanks". The idea behind this type of programmed instruction is that you do more than read - you participate. This makes for a good teaching text; but the book becomes less useful for reference or quick scanning.

The absolute beginner with a PET will encounter a few stumbling blocks at the start of the book. This is due to slight differences in the Basic being described. PET says READY instead of OK; it says ?SYNTAX ERROR instead of SN ERROR; it uses the Delete key instead of the back arrow to correct entry errors. All very minor problems, but they might shake the confidence of a neophyte. Once he gets over these initial rough spots, however, it's all clear sailing. By chapter three, the authors get into the meat of Basic programming, and the reader should have no further trouble.

Each chapter is well organized. First, you're told what you may expect to learn in this chapter. Then the text material, broken into neatly numbered sections. Liberal use is made of illustrations, diagrams, and sample programs; and the programs are usually aimed towards real world examples, not just abstract mathematics. Finally, each chapter ends with a self-test, complete with answers, which allows you to review and make sure you've got it all right.

The order of the chapters is well planned, proceeding from the more basic programming commands towards more advanced concepts. The authors don't suggest it, but by Chapter 5 the reader is equipped to skip ahead to subjects in which he may have a special interest. So if he's anxious to learn about string variables or about subroutines, he can leap ahead to chapter 9 or 10. It's nice to see a book that's so well organized that you can do this.

The book has a light touch, especially in the illustrations and choice of programs. It helps to relieve the hard slugging that is often needed to learn a programming language.

The book is a pretty good approach to learning Basic. It won't take the reader into advanced concepts, but it will give him a good start.

Review by Jim Butterfield

commodore

comments and bulletins
concerning your
COMMODORE PET tm

The Transactor

Vol. 2
BULLETIN # 2

PETtm is a registered trademark of Commodore Business Machines Inc.

June 30, '79

BITS and PIECES

Some interesting discoveries have been unearthed recently for the 8K and 16/32K versions of the PET. The single most important one, I feel, was uncovered by who else but Jim Butterfield. Burried deep in the keyboard interrupt routines is some code which does a test for the "<" key. To see this amazing little feature operate, insert a tape into cassette #1 and simply press 'PLAY' (not a LOAD). Now hold down the "<" key and PET will tell you immediatly if there is something recorded on that tape. If there is, the "<" sign will repeat across the screen at the rate of about 5/sec. If not, no repetition will occur. Now we have a way to check tapes before recording something over material we may have wanted to keep and, more importantly, tapes can now be cued up to blank tape without having to load in the last program or file. Fantastic! The test works on all PETs but only for cassette #1.

CRASH Your PET!

The following is a list of rather interesting crashes for 8K PETs. They can cause absolutely no internal damage to the machine that power-down and up won't fix.

1. This one might make a good screen-alignment test:

```
Type:   10 ABC
 Now:   POKE 1025,0
Type:   10 DEF
```

2. Decimal location 537 (0219 hex) is the low order byte of the hardware interrupt vector. Try the following and also experiment...

```
POKE 537,49
POKE 537,50
```

3. On a clear screen in the 'HOME' position, type:

   2 RVS field '*'s then RVS Off;
   A shifted 'L';
   An '@' sign;
   A RVS '@' sign.

The characters just typed should appear in the first 5 positions of the top line. Hit 'RETURN' and, of course, get a ?SYNTAX ERROR. Now SYS 32768. (SYS to the first location of screen memory) Change the display by holding down various combinations of keys (STOP, RETURN, etc.) The result result can be altered by varying the number of RVS '*'s on the top line.

4. On a clear screen in the 'HOME' position type a shifted closing bracket (◩) and 'RETURN'. Now type:

```
WAIT 32768,32,32
```

Experiment with other characters.

Merging PET programs: a final report

Jim Butterfield, Toronto

To wrap up the various activities surrounding merging or UNLIST, and bring them up to date with information on new ROM:

I. To change a program into a data file on cassette tape:

Mount blank tape on cassette 1. Type:

```
OPEN 1,1,1 : CMD 1 : LIST
```

Cassette tape will write. When writing is complete, the flashing cursor will return, but PET will not print READY - the word READY is in fact written on tape. Now close the CMD and tape file with:

```
PRINT#1 : CLOSE 1
```

This "merge" tape may now be saved for any future occasion.

Variations:

--the file may be named, e.g., OPEN 1,1,1,"TEST MERGE": ... etc. It's good practice to name files if you plan to keep them.

--if desired you may copy only part of the program to tape, e.g., ... CMD 1 : LIST 500-700 ... This is a handy way to extract subroutines from a larger program.

II. To merge a data file (in the above format) into program space:

The procedure is slightly different on original ROM as compared to the new ROM, which I'll call upgrade ROM.

The program with which you wish to merge must first be loaded into memory. The following procedure may be repeated many times, so that you may merge several program blocks together.

Mount "merge" tape on cassette 1. Type:

```
Original ROM:    POKE 3,1 : OPEN 1
Upgrade ROM:     POKE 14,1: OPEN 1
```

Tape will now be read. Eventually, the computer will report FOUND and the cursor will return.

Now: clear the screen and press exactly three cursor downs. Type:

```
Original ROM:    POKE 611,1 : POKE 525,1 : POKE 527,13 : ?"h"
Upgrade ROM:     POKE 175,1 : POKE 158,1 : POKE 623,13 : ?"h"
```

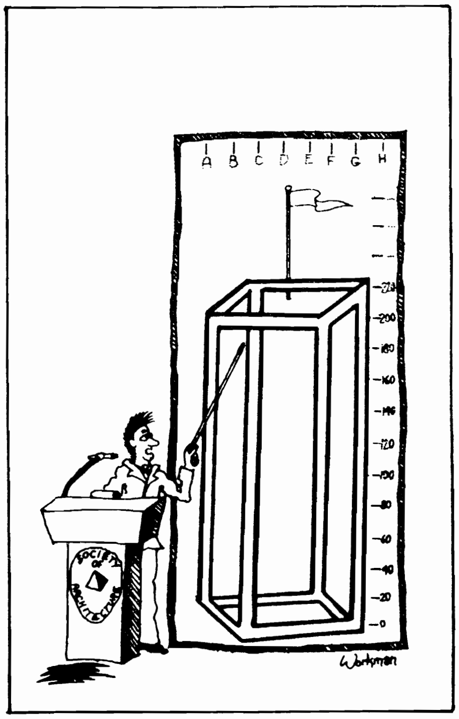
('h' is the cursor home key - it will print as a reverse S).

As soon as you press RETURN at the end of this line, the word READY will appear above the line, and tape will move. When the merge is complete, the computer will print either ?OUT OF DATA ERROR or ?SYNTAX ERROR below the line. This is normal and does not signify a real error. The job is now complete.

Note the four new items:

--a new POKE statement before OPEN 1;
--three cursor downs before the final POKE;
--only one final POKE line to be typed;
--no need to close the file at end of merge.

The new system is simpler, and also corrects a minor problem on the original POKE611 merge. Few people spotted it, but the original procedure caused line 1 to disappear.



'Gentlemen, This Is a Structural Analysis of the Proposed Shopping Complex Made by Our Arch/200 Stress Analyzer. Of Course, You May Want a Second Opinion.'

Courtesy Computerworld Newspaper

PET to Teletype Interface

The interface described below was received from Lt. W. Hawes in Nova Scotia. Note: it will operate with 8 level TTY's but not 5 level machines. Thank you Lieutenant Hawes.

Interface Description

The cct. shown in Fig. 1 is a modification of an interface that was originally built in June '78 to output to a TTY from the PET Parallel User Port. The problem with the Parallel port was that software was required to be resident in memory in order to output data and LISTing of programs was not possible since the operating system has control during a LIST. Clearly the way to go was from the IEEE 488 Port.

The modification to output from the IEEE Port was based on the cct. by Prentice Orwell (Jul/Aug 78 Pet User Notes). Some of the features of my original cct, such as UART vice shift register and clock frequency from PET vice interface oscillator, were retained.

My cct. is as shown in Fig. 1 It uses a +5v and -12v (originally only a dual supply UART was immediately available) for both the UART and the 20mA current loop. The cct. could be further simplified to a single +5v supply as shown in Fig. 2 by using a single supply UART such as the AVA - 1014A or equivalent. The 20mA loop could then be constructed using spare inverters on the 4049's.

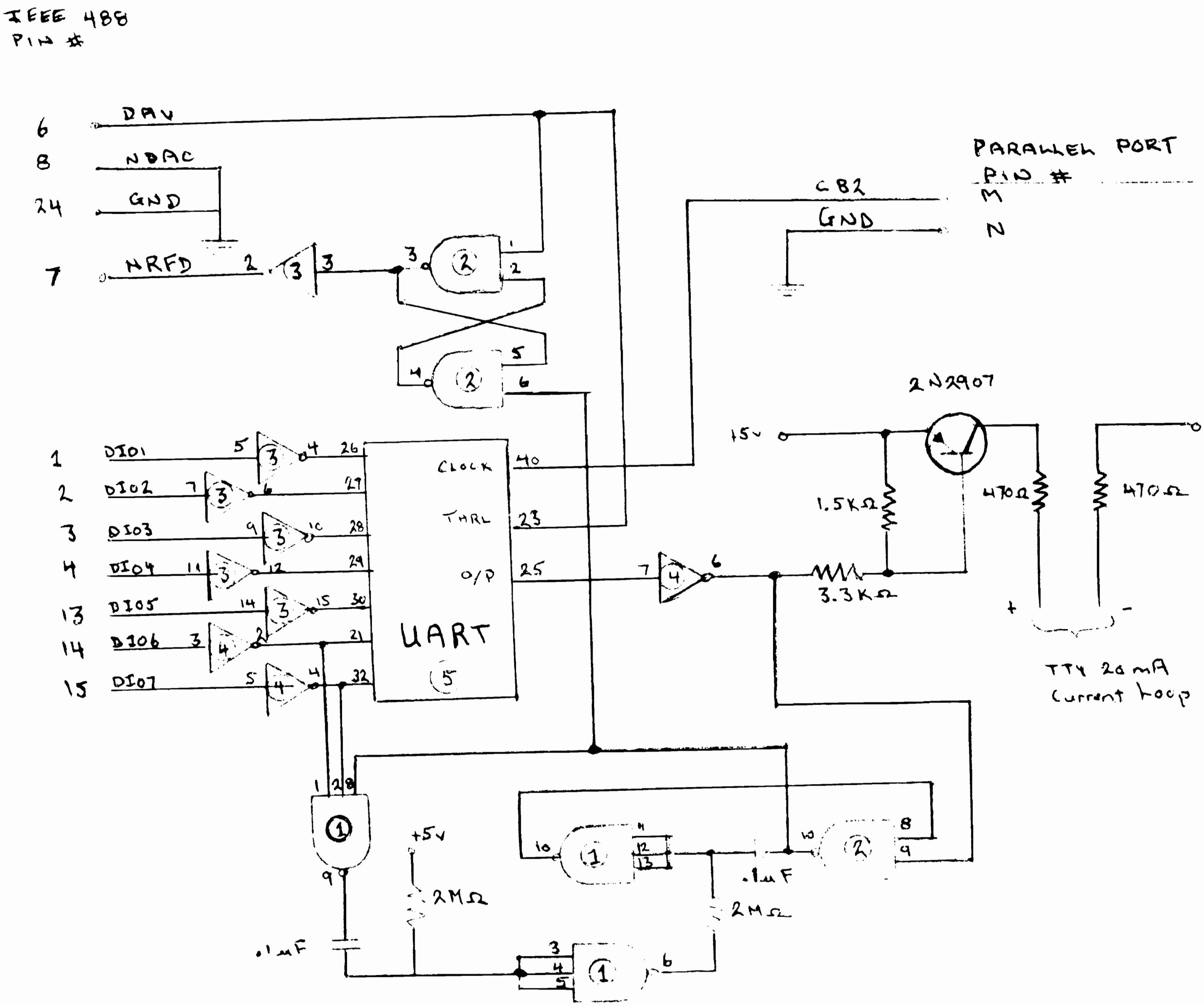
As stated above, hardware is reduced by omitting the interface oscillator. PET itself supplies the 1760 Hz (16 x baud rate) UART clock frequency from CB2 on the parallel port (see Generating Square Waves With The PET by J.R. Kinnard - MAR/APR '78 PET User Notes).

Circuit Operation

Initialize	: POKE 59467,16 : POKE 59464,69 : POKE 59466,51 (outputs 1760 Hz from CB2 to UART, tape I/O disabled)
Operate	: OPEN 4,4 : CMD 4 (Printer primary output device - enter from keyboard to LIST or include in program to be RUN
Return to Screen	: PRINT# 4 (from keyboard or include in program
System Recover	: POKE 59467,0 (restores correct tape I/O)

PET IEEE 488 / TTY 20mA Current Loop

Figure 1

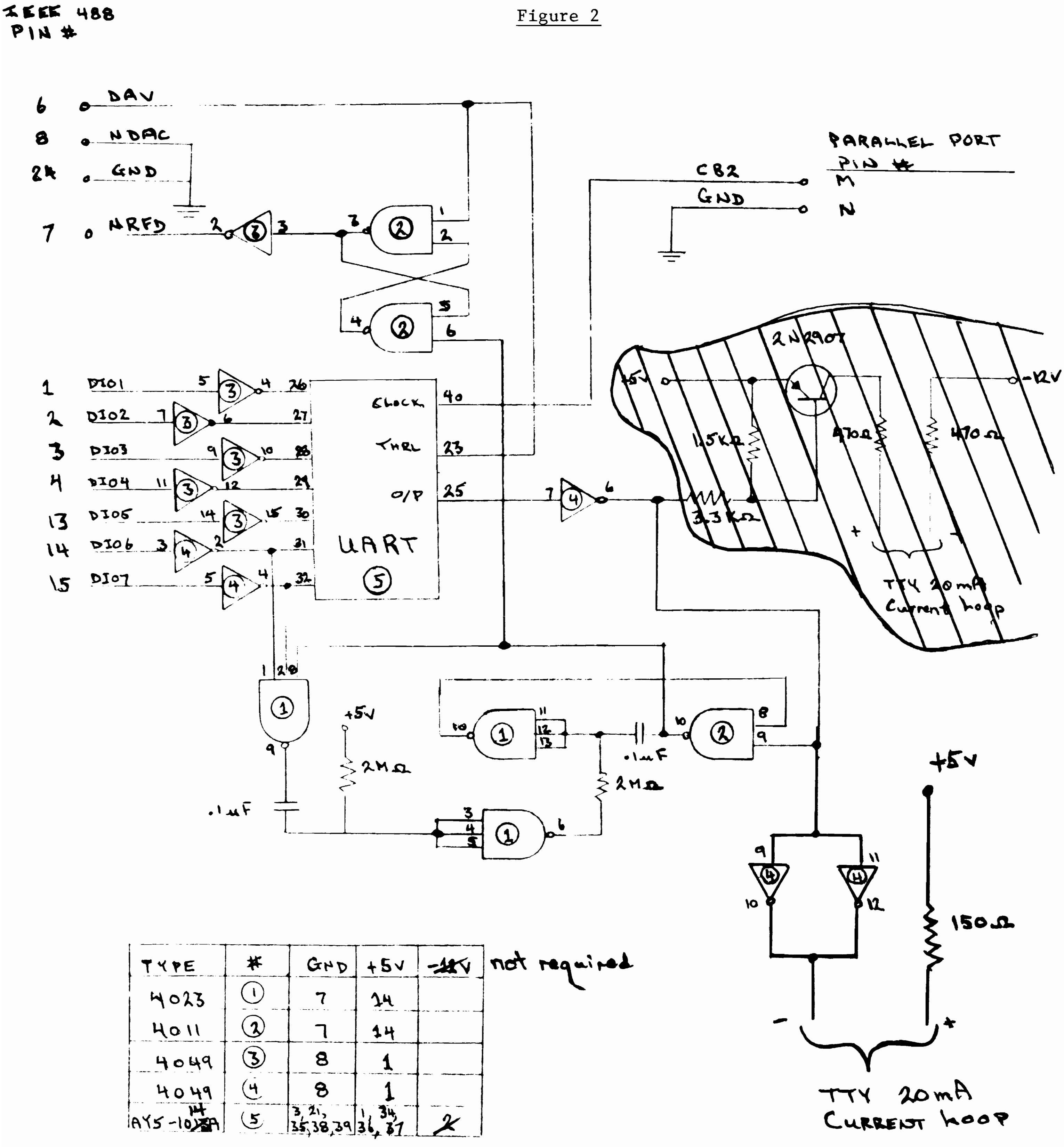


TYPE	#	GND	+5V	-12V
4023	1	7	14	
4011	2	7	14	
4049	3	8	1	
4049	4	8	1	
AY5-1013A	5	3, 21, 35, 38, 39	1, 34 36, 37	2

PET IEEE 488 / TTY 20mA Current Loop

Single +5v Power Supply Option

Figure 2



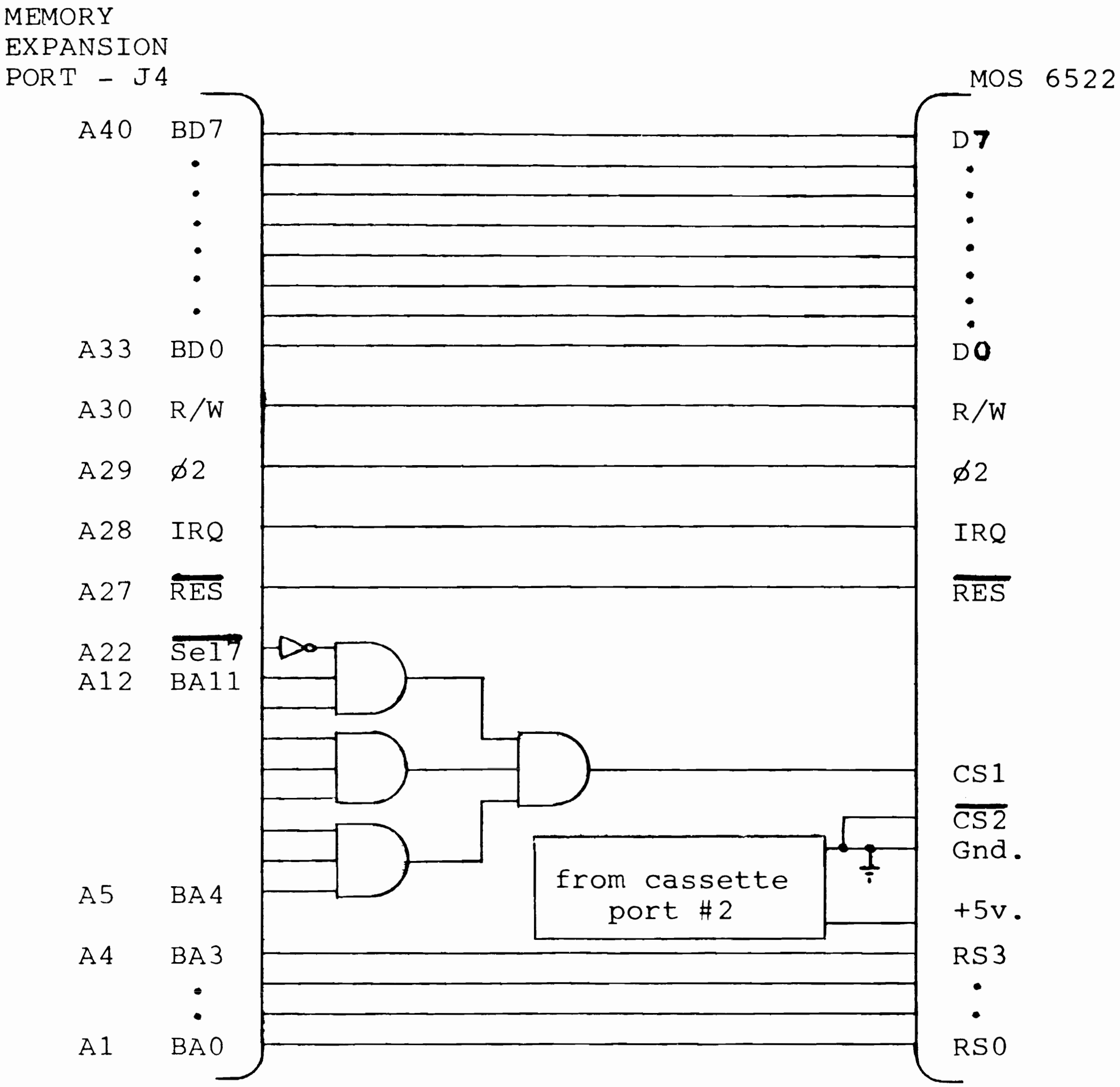
TYPE	#	GND	+5V	-~~12~~V not required
4023	①	7	14	
4011	②	7	14	
4049	③	8	1	
4049	④	8	1	
AY5-1014A	⑤	3, 21, 35, 38, 39	1, 34, 36, 37	2

Additional I/O Interface

Mr. K. Erler of Edson Alberta writes in with:

> ...a schematic of an interfacing idea of mine. It simply interfaces a second VIA chip to the PET, thus tripling the user's I/O capability. Most of it is direct interfacing -- all but the address lines which had to be decoded.
>
> The circuit uses only 4 three input 'AND' gates and one buffer inverter. Once assembled, it connects directly on to the Memory Expansion Port - J4.
>
> After connecting it, operation is very simple. The circuit is designed to use the top 16 bytes of RAM expansion space and since most PETs have only 8K (32K at the most) the very top of the memory would not be used.
>
> The addresses are as follows:
>
> | | | | |
> |---|---|---|---|
> | 32752 | ORB | 32760 | T2L-L T2C-L |
> | 32753 | ORA | 32761 | T2C-H |
> | 32754 | DDRB | 32762 | SR |
> | 32755 | DDRA | 32763 | ACR |
> | 32756 | T1L-L T1C-L | 32764 | PCR |
> | 32757 | T1C-H | 32765 | IFR |
> | 32758 | T1L-L | 32766 | IER |
> | 32759 | T1L-H | 32767 | ORA (no hand shake) |
>
> The advantages are that you get not only PA lines, but also the PB lines and CB1 & CA2 lines.
>
> The operation is as with the other VIA - PEEK and POKE, etc, only with the previously listed addresses.
>
> ### Outut Example
>
> To create a tone on CB2...
>
> ```
> POKE 32762,15  (SR)
> POKE 32760,155 (Timer 2)
> POKE 32763,16  (ACR)
> ```

Great idea, Kevin! Thank you. The schematic follows...



TTY Interface

The Microtronics M65 Morse Flash TTY Interface and Software. It can operate anywhere up to 100 wpm (send and receive modes). Fully compatible with 8, 16, and 32K PETs.

Assembled form - 149.95

Kit form - 109.95

For more information contact:

IRISCO Du QUEBEC
537 Boul Charest Est
Qubec City, Quebec
G1K 3J2

commodore

comments and bulletins
concerning your
COMMODORE PET tm

The Transactor

Vol. 2
BULLETIN # 3

PET^{tm} is a registered trademark of Commodore Business Machines Inc.

July 31,'79

PET Variable Storage

Most PET users have, at one time or another, checked FRE(0) immediately after a LOAD and again after RUNning and found the two don't match. The difference can be minimal or sometimes quite drastic...but why ? The reason (as you have probably already guessed) is variable storage.

In PET BASIC there are generally three types of variables: string, floating point and integer (array variables are handled differently than these and will be discussed later). When PET executes statements such as:

```
A = 4.8              (direct)
10 LET Y = 17.5
20 X% = 1
30 B$ = "XXXX"
```

...it stores these variables and their assignments in variable storage space; RAM memory space determined by pointers in PET's operating system which are set up on power up or after a LOAD, as the case may be. They are stored in the order in which they are encountered.

Let's take a look at how each of these variables are handled by PET. First you'll need the machine language monitor. 8k users will have to LOAD the monitor and follow these 6 preliminary steps:

1. LIST. 10 SYS(1039) should appear. If not, record the "SYS" number.

2. RUN the monitor

3. Type exactly 'M 007A,0097' and hit RETURN. The following should appear:

```
           0  1  2  3  4  5  6  7
.:   007A 01 04 6B 07 6B 07 6B 07
.:   0082 00 20 0C 21 00 20 0A 00
.:   008A 98 0E 00 04 04 08 00 04
.:   0092 CC 0C 8C EA 24 C6 88 80
.
```

4. Now take the cursor up and change the following bytes (hit 'RETURN' after each line):

```
                  0  1  2  3  4  5  6  7
     .:   007A 01 08 03 08 03 08 03 08
     .:   0082 .. .. .. .. .. .. .. ..
     .:   008A .. .. .. .. .. .. .. ..
     .:   0092 .. .. .. .. 04 08 .. ..
     .
```

5. Type 'M 0800,0807' and RETURN. The following should appear:

```
                  0  1  2  3  4  5  6  7
     .:   0800 24 24 24 24 24 24 24 24
     .
```

6. Change to:

```
                  0  1  2  3  4  5  6  7
     .:   0800 00 00 00 24 24 24 24 24
     .
```

PET has now been fooled into thinking that BASIC memory space now starts at 0800 instead of 0400. This protects the machine language monitor from being clobbered when extra BASIC is entered.

NOTE: Of course all this is unnecessary for 16/32k users as the M.L.M. is in ROM and can't be trampled on by anything. Now, however, any further insructions will require one set of addresses for 8k's and another for 16/32k's as the results of this exercise will end up in different areas of memory for the two machines. Therefore, the 16/32k user will use the addresses or parameters placed in brackets.

Exit the monitor and enter and RUN the following BASIC:

```
20 A  = 2.5               <
30 B% = 9                 <No spaces
40 C$ = "XXX"             <
```

After RUNning, re-enter the monitor with SYS1039 (SYS4 for 16/32k). Then do the following memory display:

```
.:   M 0800,0840    (0400,0440)
.:              0  1  2  3  4  5  6  7
.:   0800 00 0B 08 14 00 41 B2 32
.:   0808 2E 35 00 14 08 1E 00 42
.:   0810 25 B2 39 00 21 08 28 00
.:   0818 43 24 B2 22 58 58 58 22
.:   0820 00 00 00 41 00 82 20 00
.:   0828 00 00 C2 80 00 09 00 00
.:   0830 00 43 80 03 1C 08 00 00
.:   0838 24 24 24 24 24.......
```

Figure 1.

16k users will of course see "04's" rather than "08's" and the "24's" at the bottom will be "AA's" indicating "empty space".

Notice the first 4 rows is our BASIC program followed by three "00's" indicating 'end of BASIC'. Following this is the variable table which we'll get into right now.

Floating Point Variables

Floating point implies a numeric value with a fractional component. In our case it will be A = 2.5 . This value is stored along with its corresponding variable in the 7 memory locations following 0823 (0423) inclusive:

```
0823 41 00 82 20 00 00 00
```

Variations of the above 7 locations is all that is required to store any floating point number within the upper and lower limits of PETs floating point range.

The first two bytes are used to store the variable. 41 is "A" in hexadecimal. The next byte is set aside for double character variables (e.g. AA=2.5). Since ours is only a single character, location 0824 will be 00 as shown.

The remaining 5 bytes are for the actual value itself. The 82 is the exponent of the value. This is offset by 80 (half of FF) such that negative exponents can also be obtained. In our case 2 is added to indicate that the decimal point is 2 places to the right of the most signifigant bit. As you know, binary 2 = ...0*4 + 1*2 + 0*1....:

$$\ldots\ 2^{5}\ \ 2^{4}\ \ 2^{3}\ \ 2^{2}\ \ 2^{1}\ \ 2^{0}\ .\ 2^{-1}\ \ 2^{-2}\ \ 2^{-3}\ \ 2^{-4}\ \ 2^{-5}\ \ 2^{-6}\ \ldots$$

```
0   0   0   0   1   0 .  x   x   x   x   x   x
```

That covers the integer part...now the fractional part. We have 2 so far. We need to represent .5 more. Therefore a "1" is required in the 2 column. This is contained in the next byte following the exponent. 0826 contains 20 which, in binary, is:

```
0 0 1 0  0 0 0 0
```

This is "OR'd" into the above such that the leftmost bit is beneath the most signifigant bit of the integer part of the number:

$$\ldots\ 2^{5}\ \ 2^{4}\ \ 2^{3}\ \ 2^{2}\ \ 2^{1}\ \ 2^{0}\ .\ 2^{-1}\ \ 2^{-2}\ \ 2^{-3}\ \ 2^{-4}\ \ 2^{-5}\ \ 2^{-6}\ \ldots$$

```
              0   0   0   0   1   0 .  x   x   x   x   x   x
   OR'd with                  0   0    1   0   0   0   0   0
              ________________________________________________
           =  0   0   0   0   1   0 .  1   0   0   0   0   0
```

...which is 1*2 + 1*.5 = 2.5 !

Lastly is the sign of the value. If you study the theory of this method of deriving numbers, you'll notice that the leftmost bit of the "OR'd with" number never has to be a 1 for determining the magnitude of the number. Therefore it is used as the sign bit and is set to 1 for negative numbers. Examples of this and more floating point numbers at the end of this article.

Integer Variables

Integers are those with no fractional component and are stored by PET in a much simpler fashion. In our case, B% is stored in the 7 bytes immediately following A . But how does PET know that this variable is any different from the last. Notice the first two bytes of B% as compared to A :

```
0823 41 00
082A C2 80 00 03 00 00 00
```

Since A is represented as 41 in hex, you might expect that B is 42. Well you're right; B is 42 in hex but when B (or any other letter) is employed as an integer variable, bit 8 is set to 1 such that PET can make the distinction. Looking at the table on the last page of the last Transactor, you'll see that the letters stop at decimal 90 and therefore never use the 8th bit. Expanding...

```
"A"  = 41 = 0 1 0 0  0 0 0 1
"B"  = 42 = 0 1 0 0  0 0 1 0
"B%" = C2 = 1 1 0 0  0 0 1 0
   -        -
```

Bit 8 of the second byte of an integer variable is also set even if a double variable name is not used.

The next two bytes of the seven are the only ones used to represent the value. The remaining three are never used. Integers take no less space than floating point except in arrays. This simplifies the search process.

The first byte used in representing the value, 082C (042C), is the high order byte and the second 082D (042D), is the low order. The two are of course the hex representation of the value in decimal. Recall that the maximium integer value possible is 32767 or 8000 hex. The remaining possibilities are used for negative integers. Some examples:

```
B% =      9 = 00 09
B% =    256 = 01 00
B% =    257 = 01 01
B% =      0 = 00 00
B% =     -1 = FF FF
B% =   -256 = FE 00
B% = -32767 = 80 01
```

String Variables

For every string variable created, another 7 bytes are used up by PET but of course the string itself is not stored there. Our string variable, C$, is stored beginning at 0831 (0431). PET distinguishes string variables by setting bit 8 over the second byte only. "C" is 43 in hex:

```
0831 43 80 03 1C 08 00 00
```

Location 0833 (0433) holds the length of the string (Recall...40 C$ = "XXX"). The following two bytes are the low and high order bytes of the address of the string. Inotherwords, why store the string again when it already exists in the text area. Instead simply store a pointer which points at the first character of the string and call up X number of characters following where X equals the 'length' byte (03 in our case).

This procedure is fine for strings which are defined in text, but what about those that are not. Take for example the following:

```
100 INPUT " YOUR NAME ";A$
200 D$ = RIGHT$ ( A$ , 1 )
300 C$ = D$ + "*" + A$    .
```

In cases like these, PET stores the strings at the end of available RAM moving down and creates a pointer in the variable table to the beginning of the string.

The Search Process

Each time a variable is defined, 7 bytes of memory are used up. When a variable is called by BASIC in lines such as:

```
400 IF A = 1 THEN A = A + 5
500 PRINT B% , C$           .
600 ON A GOTO 1020 , 1030 .
700 X = X - 3               .
```

...PET starts at the beginning of the variable table, determined by the pointer at 007C & 007D (002A, 002B), and examines the first pair of bytes. If an exact match is not made, PET jumps 7 locations to the next variable. The search continues until the variable is found and if not found is assumed to be zero or null for strings.

Once established, PET loads the value or string into a work area and performs the desired operation. In a situation such as line 700, PET must find X (zero or otherwise), load it into the accumulator, find X again, subtract 3 and re-asign X. Of course all this takes time and if X resides down at the end of the table, PET must scan through all the variables ahead of X before it finds X. Therefore, if a variable is known to be used more often than others, time can be saved by "setting up" the variable table at the beginning of the program:

```
10 X = 0 : A$ = " " : B% = 0 : Y = 0
```

This can speed things up considerably especially if X is called upon each pass of a long FOR-NEXT loop.

What You Can Do

Assuming you still have the monitor running with the display as in Figure 1, change the following (do not exit

```
.   M 0800,0840    (0400,0440)
.:         0  1  2  3  4  5  6  7
.:  0800 .. .. .. .. .. .. .. ..
.:  0808 .. .. .. .. .. .. .. ..
.:  0810 .. .. .. .. .. .. .. ..
.:  0818 .. .. .. 22 59 59 59 22
.:  0820 .. .. .. .. .. 83 20 ..
.:  0828 .. .. .. .. .. 0F .. ..
.:  0830 .. .. .. 05 1C 07 .. ..
.:  0838 .. .. .. .. .. .. .. ..
.
```

Now type "X" and RETURN to exit the monitor and execute the following line directly on the screen:

```
? A , B% , C$
```

A is now 5 because the exponent of A was incremented by 1. This means that everything was shifted left one position putting the most signifigant bit (MSB) in the 2^2 column and Least signifigant bit (LSB) in the 2^0 column. 1*4 + 1*1 = 5 .

B% equals 15 now since the low order byte of B% was changed to 0F.

If you've ever tried programming this, you know it's impossible:

```
40 C$ = ""YYY""
```

PET interprets this as null string followed by the variable 'YYY' followed by null string. But now C$ prints out as '"YYY"' because the address of the string was changed as well as the length.

Floating Point Examples

The magnitude of a floating point value is always stored in 5 bytes. The other two are reserved for the variable name and will be ignored here so that we can concentrate on the format of the value.

Floating point is handled by PET in this format ('M' = Mantissa):

```
EXP  M1  M2  M3  M4  SIGN
```

The sign is contained in M1 but is "extracted" on its way into the accumulator and placed in a 'sign register'.

```
Eg 1.   EXP   M1   M2   M3   M4
         85   22   40   00   00
```

Since the EXP is 85, the decimal point will be 5 positions to the right of the MSB (Most Signifigant Bit):

EXP = _ _ _ 1 0 0 0 0 . 0 0 _ _ _

So far the magnitude is 16.

```
M1 = 22 = 0 0 1 0  0 0 1 0

M2 = 40 = 0 1 0 0  0 0 0 0

M3 = M4 = 0
```

To complete the operation, M1 and M2 are concatenated...

```
M1 + M2 = 0 0 1 0  0 0 1 0  0 1 0 0  0 0 0 0
```

...and OR'd with the EXP such that the leftmost bit of M1 + M2 is under the MSB of the value:

EXP = _ _ _ 1 0 0 0 0 . 0 0 _ _

OR'd with: M1 + M2 = _ _ _ 0 0 1 0 0 . 0 1 0 0 1 0 0 0 0 0 0

Equals: _ _ _ 1 0 1 0 0 . 0 1 0 0 1 0 _ _ _ _ _

This is still the binary representation. The decimal value is now:

$1*2^{4} + 0*2^{3} + 1*2^{2} + 0*2^{1} + 0*2^{0} + 0*2^{-1} + 1*2^{-2} + 0*2^{-3} + 0*2^{-4} + 1*2^{-5} + 0*2^{-6} + ...$

..which equals...

1*16 + 1*4 + 1*.25 + 1*.03125 = 20.28125

```
                      EXP M1 M2 M3 M4
Therefore 20.28125 =  85 22 40 00 00
```

```
Eg 2.   EXP M1 M2 M3 M4
         8A FF E7 80 00
```

Since the EXP is 8A, the decimal point will be 10 positions to the right of the MSB.

EXP = _ _ _ 1 0 0 0 0 0 0 0 0 0 . 0 0 _ _

Notice that bit 8 of Mantissa 1 is set. Therefore, the sign of the value will be negative. Now M1, M2, M3 and M4 must be concatenated:

```
M1 = FF = 1111 1111
M2 = E7 = 1110 0111
M3 = 80 = 1000 0000
M4 = 00
```

M(1+2+3) = 1111 1111 1110 0111 1000 0000

...and OR'd with the EXP...

EXP = _ _ 1 0 0 0 0 0 0 0 0 0 . 0 0

OR: M = _ _ 1 1 1 1 1 1 1 1 1 1 . 1 0 0 1 1 1 1 0 0 0 0 0

Equals = _ _ 1 1 1 1 1 1 1 1 1 1 . 1 0 0 1 1 1 1 0 _ _ _ _

..which equals...:

$2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-4} + 2^{-5}$
$+ 2^{-6} + 2^{-7}$

In decimal:

512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 + .5 + .0625 +
.03125 + .015625 + .0078125 = 1023.6171875

However, the sign is negative...therefore:

```
               EXP M1 M2 M3 M4
-1023.6171875 =  8A FF E7 80 00
```

Eg 3. EXP M1 M2 M3 M4
7E E0 00 00 00

The EXP is less than 80 indicating the result will be less than 1. Now the decimal point will be 2 positions to the left of the MSB because 7E is 2 less than 80:

EXP = _ _ _ 0 0 . 0 1 0 0 _ _ _ _ _ _ _

M1 = E0 = 1110 0000
M2 = 00
M3 = 00
M4 = 00

EXP = _ _ _ . 0 1 0 _ _ _ _ _ _
OR: M = _ _ _ . _ 1 1 1 0 0 0 0 0

Equals _ _ _ . 0 1 1 1 0 0 _ _ _

In decimal = -(.25 + .125 + .0625)
= -.4375

Nothing to it, right? Try these:

Eg 4. EXP M1 M2 M3 M4
84 48 00 00 00 = _ _ _ _ _ _ . _ _ _

Eg 5. EXP M1 M2 M3 M4
7F C0 00 00 00 = _ _ . _ _ _ _ _ _ _

Eg 6. EXP M1 M2 M3 M4
___ ___ ___ 00 00 = 2 9 . 5

IEEE BUS HANDSHAKE ROUTINE IN MACHINE LANGUAGE

To use the IEEE-488 bus on the PET at maximum speed it is necessary to use machine language rather than BASIC 'INPUT' and 'PRINT'. The routine given here has been used with an HP3437A systems voltmeter to reach data transfer speeds of over 5000 bytes per second, corresponding to 2500 voltage readings in 2-byte packed binary format or 625 readings in 8-byte ASCII format. The best speed attained in BASIC is 75 readings per second transferred as character strings.

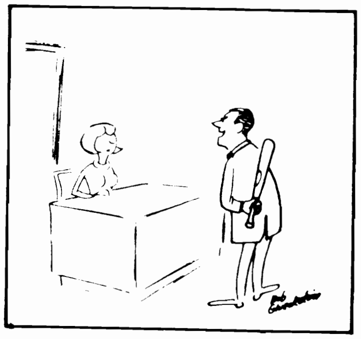
The IEEE bus

Details of the IEEE-488 bus are given in the PET Users Handbook, but some clarification of the register addresses on page 120 of the handbook is helpful. These are:

HEX	DECIMAL	BITS	IEEE	DIRECTION
E820	59424	0-7	DIO 1-8	from bus
E822	59426	0-7	DIO 1-8	to bus; 'PET'controlled
E821	59425	3	NDAC	'PET' controlled
E823	59427	3	DAV	'PET' controlled
E840	59456	0	NDAC	from bus
		1	NRFD	'PET' controlled
		2	ATN	'PET' controlled
		6	NRFD	from bus
		7	DAV	from bus

Note that on the IEEE bus, 'high' is logic false and 'low' is logic true; and that the data bus must be left with all bits 'high' when PET has finished to avoid confusion of data put on to the bus by other devices.

Cont'd →



'I'd Like to Reason With Your Computer.'

'A Sudden Reduction of Personnel Is Indicated.'

The program controls a given number of data transfers, each of 8 bytes, from the HP3437A to the PET. Each one is preceded by a trigger (GET - group execute trigger) on the IEEE bus, and the HP3437A must be correctly addressed as a 'talker' or a 'listener' at all times by sending MTA (my talk address) or MLA (my listen address) before transfers as appropriate. The sending of messages (GET, MTA, MLA, etc.) or data is controlled by the ATN line; ATN is true when messages are being sent.

The program and returned data are held in the top 2K of memory; this is hidden from BASIC using POKE 134,255 : POKE 135,23 as the first line of the BASIC control program. The number of readings required is POKEd into 6400_{10}, then control passed to the machine language program by SYS(6144). The data bytes coming in on the IEEE bus are stored in locations 6401_{10} onwards; these are PEEKed out on return to BASIC, and converted into numbers using the function VAL. As the index register is used for counting, only 256 bytes can be transferred using this program, but it would be easy to modify the program to perform more transfers.

Disassembled listings with comments and a separate listing (for ease of copying into BASIC DATA statements!) are given.

This program was prepared using a machine language handler written by the author, and the listings produced by this handler and by a modified version of the 'disassemble' part of the PETSOFT©ASSEMBLER 'EXEC' program.

IEEE bus handshake routine - main program

```
1800 A200    LDX #00     prepare index register
1802 A9FB    LDA #FB     set ATN low
1804 2D40E8  AND E840
1807 8D40E8  STA E840
180A A928    LDA #28     MLA (28 for this device)
180C 8501    STA 01
180E 208018  JSR 1880    handshake into bus
1811 A908    LDA #08     GET
1813 8501    STA 01
1815 208018  JSR 1880    handshake
1818 A948    LDA #48     MTA
181A 8501    STA 01
181C 208018  JSR 1880    handshake
181F A9FD    LDA #FD     set NRFD low (ready to receive data)
1821 2D40E8  AND E840
1824 8D40E8  STA E840
1827 A9F7    LDA #F7     and NDAC low also
1829 2D21E8  AND E821
182C 8D21E8  STA E821
182F A904    LDA #04     set ATN high
1831 0D40E8  ORA E840
```

```
1834 8D40E8 STA E840
1837 A008   LDY #08      ready to count 8 bytes
1839 20B018 JSR 18B0     handshake data from bus
183C A502   LDA 02       result to A
183E 9D0119 STA 1901,X   store in  1901+X
1841 E8     INX
1842 88     DEY
1843 D0F4   BNE 1839     jump if Y not zero
1845 A9FB   LDA #FB      set ATN low
1847 2D40E8 AND E840
184A 8D40E8 STA E840
184D A902   LDA #02      set NRFD high
184F 0D40E8 ORA E840
1852 8D40E8 STA E840
1855 A908   LDA #08      set NDAC high
1857 0D21E8 ORA E821
185A 8D21E8 STA E821
185D A95F   LDA #5F      UNT
185F 8501   STA 01
1861 208018 JSR 1880     handshake to bus
1864 A904   LDA #04      set ATN high
1866 0D40E8 ORA E840
1869 8D40E8 STA E840
186C CE0019 DEC 1900     decrease counter
186F D091   BNE 1802     jump if not zero
1871 60     RTS          return to BASIC program
```

subroutine to handle handshake into bus

```
1880 AD40E8 LDA E840     NRFD ?
1883 2940   AND #40
1885 F0F9   BEQ 1880     jump back if not ready
1887 A501   LDA 01       ready:  get data byte
1889 49FF   EOR #FF      complement it
188B 8D22E8 STA E822     send to bus
188E A9F7   LDA #F7      set DAV low
1890 2D23E8 AND E823
1893 8D23E8 STA E823
1896 AD40E8 LDA E840     NDAC ?
1899 2901   AND #01
189B F0F9   BEQ 1896     jump back if not accepted
189D A908   LDA #08      accepted;  set DAV high
189F 0D23E8 ORA E823
18A2 8D23E8 STA E823
18A5 A9FF   LDA #FF      $255_{10}$ into bus
18A7 8D22E8 STA E822
18AA 60     RTS          return to main
```

subroutine to handle handshake from bus

```
18B0 A902   LDA #02      set NRFD high
18B2 0D40E8 ORA E840
1885 8D40E8 STA E840
18B8 AD40E8 LDA E840     DAV  ?
18BB 2980   AND #80
18BD D0F9   BNE 18B8     jump back if not valid
18BF AD20E8 LDA E820     get data byte from bus
18C2 49FF   EOR #FF      complement
18C4 8502   STA 02       store in $ 0002
```

```
18C8 2D40E8 AND E840
18CB 8D40E8 STA E840
18CE A908   LDA #08      set NDAC high
18DO OD21E8 ORA E821
18D3 8D21E8 STA E821
18D6 AD40E8 LDA E840     DAV high ?
18D9 2980   AND #80
18DB FOF9   BEQ 18D6     jump back if not
18DD A9F7   LDA #F7      set NDAC low
18DF 2D21E8 AND E821
18E2 8D21E8 STA E821
18E5 A9FF   LDA #FF      255_10 into bus
18E7 8D22E8 STA E822
18EA 60     RTS          return to main
```

IEEE bus handshake routine listing

```
1800 A2 00 A9 FB 2D 40 E8 8D
1808 40 E8 A9 28 85 01 20 80
1810 18 A9 08 85 01 20 80 18
1818 A9 48 85 01 20 80 18 A9
1820 FD 2D 40 E8 8D 40 E8 A9
1828 F7 2D 21 E8 8D 21 E8 A9
1830 04 OD 40 E8 8D 40 E8 AO
1838 08 20 BO 18 A5 02 9D 01
1840 19 E8 88 DO F4 A9 FB 2D
1848 40 E8 8D 40 E8 A9 02 OD
1850 40 E8 8D 40 E8 A9 08 OD
1858 21 E8 8D 21 E8 A9 5F 85
1860 01 20 80 18 A9 04 OD 40
1868 E8 8D 40 E8 CE 00 19 DO
1870 91 60 EA EA EA EA EA EA
1878 EA EA EA EA EA EA EA EA
1880 AD 40 E8 29 40 FO F9 A5
1888 01 49 FF 8D 22 E8 A9 F7
1890 2D 23 E8 8D 23 E8 AD 40
1898 E8 29 01 FO F9 A9 08 OD
18AO 23 E8 8D 23 E8 A9 FF 8D
18A8 22 E8 60 EA EA EA EA EA
18BO A9 02 OD 40 E8 8D 40 E8
18B8 AD 40 E8 29 80 DO F9 AD
18CO 20 E8 49 FF 85 02 A9 FD
18C8 2D 40 E8 8D 40 E8 A9 08
18DO OD 21 E8 8D 21 E8 AD 40
18D8 E8 29 80 FO F9 A9 F7 2D
18EO 21 E8 8D 21 E8 A9 FF 8D
18E8 22 E8 60
```

0001 data to go into bus
0002 data from bus

1900 counter for number of data transfers

1901 start of results area

John A. Cooke

Department of Astronomy
University of Edinburgh
Royal Observatory
Edinburgh EH9 3HJ

commodore

comments and bulletins
concerning your
COMMODORE PET™

The Transactor

Vol. 2
BULLETIN # 4
August 31, 1979

PET™ is a registered Trademark of Commodore Inc.

Computed GOTO

Ever wanted to program a GOTO followed by an expression such as:

```
120 IF ST GOTO (ST * 10)
```

Normally PET does not allow this but Brad Templeton of Mississauga has submitted a machine language routine that will handle a computed GOTO. The program fits in only 12 twelve bytes and can be placed in any part of memory where it won't get clobbered by BASIC. It accesses code in ROM and therefore has two versions, one for original ROM and another for upgrade ROM:

```
Original ROM:    JSR CE11     checks for comma
                              else SYNTAX ERROR
                 JSR CCA4     evaluates expression
                 JSR D6D0     integer? >=0 and <=63999
                 JMP C7A0     jump to GOTO routine with result


Upgrade  ROM:    JSR CDF8
                 JSR CC8B     same as above
                 JSR D6D2
                 JMP C7B0
```

Because the program has no reference to itself, it can be placed anywhere, but for now we'll put it in the 2nd cassette buffer starting at 826 or hex 033A. Syntax for using the routine will be:

```
                       SYS826,expression
    or...        G%=826 : SYSG%, expression

    e.g.         IF ST THEN SYS G%, ST * 10
```

BASIC Loader

With the following modification, both of the above routines can be loaded into the 2nd cassette buffer and PET will decide which to use. This way, programs using the computed GOTO can be run with either ROMs.

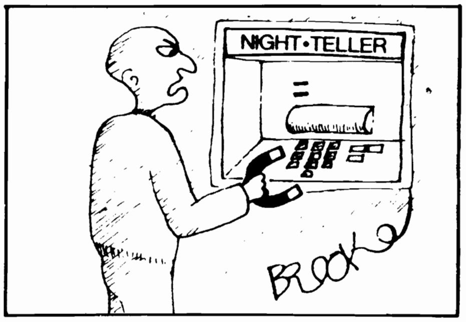
```
                    LDA  $F202
                    BMI  $0D
N.ROMs:             JSR   CDF8
                    JSR   CC8B
                    JSR   D6D2
                    JMP   C7B0
O.ROMs:             JSR   CE11
                    JSR   CCA4
                    JSR   D6D0
                    JMP   C7A0
```

The following BASIC program will load the above:

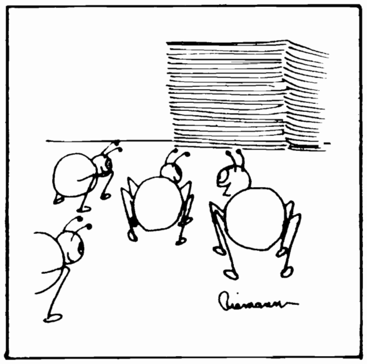
```
100 FOR J = 826 TO 854
110 READ X
120 POKE J , X
130 NEXT
200 DATA 173 , 02 , 242 , 48 , 13
210 DATA 32 , 248 , 205 , 32 , 139 , 204 ,
     32 , 210 , 214 , 76 , 176 , 199
220 DATA 32 , 17 , 206 , 32 , 164 , 204 ,
     32 , 208 , 214 , 76 , 160 , 199
```

Test with the following:

```
10 G% = 826
20 ?"TEST"
30 SYS G%, 2 * 10
```



'Give Me All Your $10s, $20s and $50s.'



'Looks Like a Good Program. Climb In, Everybody.'

from Computerworld Magazine

More PET "Quirks"   Rick Ellis, Toronto, Canada.

Clear The Screen on your 8K PET and type in the following lines:

```
POKE 59468,14

1ØOp1,3:?"cs":X=63:Y=192

2ØFoI=ØTOX:I$=Ri(STr(I),1):?"chrv"TaI)Ch(I+Y)"cl";:
Ge#1,A$:?:?"chcdcdcd"TaX-I)A$:NeI

Li
```

(cs = Clear Screen)
(ch = Cursor Home)
(rv = Reverse)
(cl = Cursor Left)
(cd = Cursor Down)

(Type line #20 above as one continuous line).

Surprise ! Line 20 is over 100 characters long. Before you try to run the above program, check that your listed version reads as follows. If not, correct it now by moving the cursor up and correcting the version you typed in to match the above:

```
1Ø OPEN1,3:PRINT"cs":X=63:Y=192

2Ø FOR I=Ø TO X:

      I$=RIGHT$(STR$(I),1):

      PRINT"chrv" TAB(I) CHR$(I+Y) "cl";:

      GET#1,A$:

      PRINT:

      PRINT"chcdcdcd"TAB(X-I) A$:

      NEXT I
```

except of course you wont see it spaced out as above.

Now type:

```
Ru
```

The program now displays a character-string on screen lines 1 & 2, in REVERSE, and as it prints each character, reads it from the screen with the GET#1 command, and reprints it in reverse order below.

Try changing line #10 (yes, it's short enough!), so that Y = 64 and RUN again.

INDENTING PROGRAM TEXT by B. Seiler

Many programmers like to indent their FOR-NEXT loops, to enhance readability. Up until now, this has only been possible by putting a colon (:) at the start of each line to be indented or spaced. For example:

```
10 FOR I=1 TO 10
20 : FOR J=1 TO 10
30 :    FOR K=1 TO 10
40 :
50 :       PRINT I,J,K
60 :
70 :    NEXT K
80 : NEXT J
90 NEXT I
```

This helps readability greatly, but you can go even further! By substituting SHIFTED(graphic) characters instead of colons, and using ▒ ▒ (graphic space graphic) to blank a line, the listing would be typed in like this (note: any shifted character can be substituted for the ▒):

```
10 FOR I=1 TO 10
20 ▒ FOR J=1 TO 10
30 ▒    FOR K=1 TO 10
40 ▒ ▒
50 ▒       PRINT I,J,K
60 ▒ ▒
70 ▒    NEXT K
80 ▒ NEXT J
90 NEXT I
```

This would list like this:

```
10 FOR I=1 TO 10
20   FOR J=1 TO 10
30      FOR K=1 TO 10
40
50         PRINT I,J,K
60
70      NEXT K
80   NEXT J
90 NEXT I
```

The same result is achieved, but the listing is cleaner. To use the screen editor, and retain this formatting, list the problem lines, put a ▒ after the line#, and edit as usual.

IFless Decisions

99% of all computer programs contain at least one decision making statement. The fundamental decision makers in PET BASIC are of course the IF-THEN and IF-GOTO statements. However, when a program performs a lot of tests or comparisons, it can become plagued with IF-THEN statements. Following are a few techniques for making decisions without 'IF'.

1. In real-time programs where GET is used to echo keyboard input onto the screen, some keys may need to be intercepted else cause undesirable effects; keys such as RVS, DEL, INST, CLR, etc. Also, other keys might want to be used as 'control' keys for initializing functions; keys such as RETURN, RVS, shifted RETURN, HOME, etc. Below is a routine which eliminates countless 'IFs'.

```
55000 C$ = "@#$%&*+/>< :'CLR''HOME''RVS'
      'RVSOFF' " + CHR$('DEL') + CHR$('INST')
      + CHR$('RET') + CHR$('shRET')
55010 GET T$ : IF T$ = "" THEN 55010
55020 B = 0
55030 FOR J = 1 TO LEN (Z$)
55040 A$ = MID$ (Z$ , J , 1)
55050 IF A$ = T$ THEN B=J : J=LEN(Z$)
55060 NEXT
55070 IF B = 0 THEN PRINT T$;:GOTO 55010
55080 ON B GOTO 60000,60100,60200,60300,
      60400,60500,60600,60700,60800,60900,
55090 ON B-10 GOTO 61000,61100,61200,61300
      61400,61500,61600,61700,61800,61900
```

This routine will PRINT any character not included in Z$. A repeat-key could also be implemented with:

```
55070 IF B = 0 THEN PRINTT$;:POKE515,255
      :GOTO55010
```

2. See if you can determine what decisions the following two programs are making.

```
10 INPUT X , Y
20 PRINT ( X + Y - ABS( X - Y ))/2
30 GOTO 10
```

```
10 INPUT X , Y
20 PRINT ( X + Y + ABS( X - Y ))/2
30 GOTO 10
```

Modifications of the above routines (i.e. using a FOR-NEXT loop and array variables) might be useful in programs performing sorts.

```
3.  IF B = 0 THEN A = 32768
    IF B = 1 THEN A = 1.259
    IF B = 2 THEN A = 556.2
    IF B = 3 THEN A = 400 * B
```

The above could continue forever depending on the possibilities for B. Try the following in direct mode on your PET:

```
    Type: B = 2        and RET
Now type: ? B = 0
```

PET will reply with 0 because B does not equal 0.

```
    Type: ? B  = 2
     and: ? B <> 0
```

In both cases PET will return a "-1" because the statements are true. This can be used most efficiently to replace the above IF-THEN statements:

```
    A = -((B = 0)* 32768 + (B=1)* 1.259 + (B=2)* 556.2
                      + (B=3) * 400 * B)
```

Since only one will be true, the others will be multiplied by zero and added. The negative sign in front changes the result back to positive.

4. This one uses the 'IF' statement but no comparison operator is used (i.e. >,=,<,<>). Try the following program.

```
10 INPUT X
20 IF X THEN ?"DID BRANCH":GOTO 10
30 ?"DID NOT BRANCH":GOTO 10
```

"DID BRANCH" occurs if X is anything but zero. On what condition does the following program branch:

```
10 INPUT X
20 IF NOT X AND 1 THEN ?"DID BRANCH":GOTO 10
30 ?"DID NOT BRANCH":GOTO 10
```

A Fast Sort.

Jim Butterfield, Toronto

When you need to sort a large array, sorting speed becomes important. Most simple sorts become very slow, since twice as many items will take four times as long to sort.

This fast sort is called "selective replacement"; it's classified as a tree type sort. It needs an index array, called I(J) here, which is twice the size of the items to be sorted. Memory can be saved, if needed, by making it an integer type array.

```
100 DIM I(200), N$(100). A$(100)
110 REM  SIMPLE INPUT ROUTINE - WRITE YOUR OWN FOR FILES
120 INPUT "HOW MANY ITEMS"; N
130   FOR J=0 TO N-1
140   INPUT "NAME";N$(J)
150   INPUT "ADDRESS";A$(J)
160   REM    INPUT OTHER DATA HERE IF DESIRED
170   NEXT J
200 REM   SORT STARTS HERE - INITIAL SCAN FINDS FIRST NUMBER
210 B=N-1 : FOR J=0 TO B : I(J)=J : NEXT J
220   FOR J=0 TO N*2-3 STEP 2
230   B=B+1 : I1=I(J) : I2=I(J+1)
240   GOSUB 700   :    REM  PERFORM COMPARISON
250   I(B)=I : NEXT J
300 REM   MAIN LOOP - OUTPUT NEXT VALUE
310 X=X-1 : C=I(B) : IF C<0 GOTO 800
320 REM  OUTPUT ITEM TO SCREEN, PRINTER, OR FILE AS DESIRED
330 PRINT N$(C),A$(C)
340 I(C)=X
350 REM  INNER LOOP TO FIND NEXT ITEM
360 C%=C/2 : J=C%*2 : C=N+C% : IF C>B GOTO 300
370 I1=I(J) : I2=I(J+1)
380 IF I1<0 THEN I=I2 : GOTO 410
390 IF I2<0 THEN I=I1 : GOTO 410
400 GOSUB 700 :  REM     PERFORM COMPARISON
410 I(C)=I : GOTO 350
700 REM     COMPARE TWO ITEMS - MODIFY TO FIT APPLICATION
710 I=I1 : IF N$(I2)<N$(I1) THEN I=I2
720 RETURN
800 STOP :   REM END OF SORT
```

As you get the sorted item at line 320, it's best to output it (or process it) on the spot. If some reason exists for completing the sort before going on to other processing, you'll find that index array I(J) contains information about the proper order for the data.

Disabling the STOP key.

It's useful to be able to disable the STOP key, so that a program cannot be accidentally (or deliberately) stopped.

METHOD A is quick. Any cassette tape activity will reset the STOP key to its normal function, however.

METHOD A, Original ROM:
Disable the STOP key with POKE 537,136
Restore the STOP key with POKE 537,133

METHOD A, Upgrade ROM:
Disable the STOP key with POKE 144,49
Restore the STOP key with POKE 144,46

Method A also disconnects the computer's clocks (TI and TI$). If you need these for timing in your program, you should use method B.

METHOD B is slightly more lengthy, but does not disturb the clocks. This method prohibits cassette tape activity.

METHOD B, Original ROM:

```
100 R$="20>:??:9??8=09024<88>6"
110 FOR I=1 TO LEN(R$)/2
120 POKE I+900,ASC(MID$(R$,I*2-1))*16 +
         ASC(MID$(R$,I*2))-816 : NEXT I
```

After the above has run:
Disable the STOP key with POKE 538,3
Restore the STOP key with POKE 538,230

METHOD B, Upgrade ROM:

```
100 R$="20>:??:9??8=9;004<31>6"
110 FOR I=1 TO LEN(R$)/2
120 POKE I+844,ASC(MID$(R$,I*2-1))*16 +
         ASC(MID$(R$,I*2))-816 : NEXT I
```

After the above has run:
Disable the STOP key with POKE 144,77 : POKE 145,3
Restore the STOP key with POKE 145,230 : POKE 144,46

How they work: Both methods change the interrupt program which takes care of the keyboard, cursor, clocks and the stop key.

Method A simply skips the clock update and the stop key test.

Method B builds a small program into the second cassette buffer which performs the clock update and stop key test, but then nullifies the result of this test.

The little program in method B is contained in R$ in "pig hexadecimal" format. Machine language programmers would read this as: 20 EA FF (do clock update and stop key test) A9 FF 8D 9B 00 (cancel stop test result) 4C 31 E6 (continue with keyboard service, etc.)

commodore

comments and bulletins
concerning your
COMMODORE PET™

The Transactor

Vol. 2
BULLETIN # 5
Oct. 31, 1979

PET™ is a registered Trademark of Commodore Inc.

Bits and Pieces

Chuan Chee of St. Catherines, Ontario, has written the Transactor with a few items of interest:

1. When a variable is assigned the value zero with " A = 0 ", it can be substituted with " A = . ". The decimal point in this case is equivalent to zero and is 600 microseconds faster than zero. This does not mean that " 1000 " can be replaced by " 1... " since the latter is interpreted as "1" followed by a decimal point and two zeros.

2. "LIST 0" lists the whole program instead of just statement 0.

3. "(shift)RETURN" acts only as a simple CRLF instead of entering it into BASIC to be interpreted.

4. Statements such as " 2*-3 " and " 2/-3 " are possible on the PET whereas other computers require " 2*(-3) " and " 2/(-3) ". In fact, you can have up to 14 "-" signs and any number of "+" preceeding a numeric. Any more than 14 "-" will result in an ?OUT OF MEMORY ERROR as the stack used by BASIC is overflowed.

5. When trying ? "Y" < "YES", PET replies with -1 which is correct. Now try; A$ = "Y" : ? A$ < "YES" and PET returns a 0 which is wrong. If this is entered as a program as follows:

```
10 A$ = "Y" : ? A$ < "YES"
```

....and RUN, PET replies with -1. So why does it work in program mode but not immediate mode?

Answer anyone?

'Billy, As Soon As You Finish Your Homework Could You Help Mommy And Me Balance the Checkbook?'

BIRTHD
WEDDING
WELL

'Do You Have Any "Sorry Your Program Bombed" Cards?'

Memory Expansion. Cost: $0.00

Ever been stuck for those few extra bytes needed to complete a program? 8K users probably know the feeling. Well now there is a consolation. If your program does not use tape file access with the second cassette, then the RAM memory devoted to the 2nd cassette buffer can be added to the memory used for BASIC.

The procedure is somewhat different for old ROMs and new but the concept is the same. Every byte of RAM in PET is physically and electronically identical. PET splits up RAM using pointers. Since these pointers are stored in RAM they can therefore be changed. Let's take a look at these pointers individually:

Old ROM:

In PETs with old ROMs, there are basically 4 pointers used to create partitions within RAM. Pointers use two bytes and are stored low order first, high order second.

1. Start of BASIC Pointer

   The start of BASIC pointer does exactly what you might think it would do; point at the start of BASIC. It is stored in locations $007A and $007B or decimal 122 and 123 and on power-up it is set to $0401 or decimal 1025. PET calls on this pointer to determine where to begin executing a RUN.

2. End of BASIC / Start of Variables Pointer

   As BASIC statements such as A=0 and X%=10 are executed, a variable table is set up immediately following the BASIC program. The variables and their corresponding values are stored in the table and consume 7 bytes each. When called, in statements such as IF A=0 THEN... , PET jumps to the location according to the value of this pointer and begins searching. When an exact match between the variable in the current statement and one stored in the table is made, PET fetches the corresponding value and moves it to a work area and BASIC continues.

   This pointer is stored at $007C and $007D or decimal 124 and 125 and on power-up is set to $0404 or 1028 decimal. It's value, however, will constantly be changing as BASIC code is inserted or deleted. This is why the values of all variables become zero when a program change is made; if code is inserted, program text is written over the first variables in the table. If code is deleted, the bytes used by the variable table are untouched but the end of BASIC is changed and this pointer is no longer set to the start of variables.

3. End of Variables / Start of Arrays Pointer

Stored at $007E-7F or decimal 126-127, this pointer works much the same way as the previous one when array variables are called. It is also set to $0404 on power-up. As DIM statements are executed, arrays are set up starting at the location determined by this pointer. This will be the first byte following the last byte of the variable table. But what happens when a value is assigned to a new variable? If no arrays exist, the new variable and its value are simply stored in the 7 bytes following the location pointed at by the End of Variables pointer inclusive. The pointer is then updated to await the next new variable.

However, if arrays are present, a space must be created such that the new entry can be inserted as part of the variable table. This means that the arrays must first be moved up 7 bytes. Try the following:

```
          Power-up PET

       Type: ?TI : A=0 : ?TI
   Note the time difference

   Now type: DIM A (4,255)
        and: ?TI : B=0 : ?TI
   Notice how much longer it takes
```

The extra time is spent transfering each byte of the arrays ahead by 7 bytes. Of course PET must start with the last byte of the arrays which brings us to...

4. End of Arrays / Start of Available Space Pointer

When PET must open up a space for a new variable by moving the arrays up, it calls on this pointer to determine where to start transfering bytes. PET continues this byte by byte transfer until the byte pointed at by the start of arrays pointer is also moved. The new entry is then inserted...process complete.

The End of Arrays pointer lies at $0080-81 or decimal 128-129 and also contains $0404 after power-up.

<u>New ROM:</u>

In new ROM PETs there are also basically 4 pointers used to section off RAM and are used the same way as old ROM PETs. However, they are stored in different places.

Pointers:	Decimal Locations:	Old ROM	New ROM
Start of BASIC		122-123	40-41
End of BASIC / Start of Variables		124-125	42-43
End of variables / Start of Arrays		126-127	44-45
End of Arrays or Start of Available Space		128-129	46-47

Moving Pointers

Now that we know where these pointers are and what they do, some experimenting can be done. Recall that on power-up the Start of BASIC Pointer is set to hex 0401 or decimal 1025. However, location 1024 is also important. It has the value zero and represents a "dummy end-of-line".

The 2nd cassette buffer starts at hex 033A or decimal 826. If this is to be included as part of BASIC memory space, the Start of BASIC Pointer must be moved DOWN. Since location 826 will have to serve as the dummy end-of-line character, the new start of BASIC will be 827 or $033B. The procedure is as follows:

```
POKE 826 , 0    :Dummy end-of-line
POKE 122 , 59   :low order byte of pointer = $3B (3*16+11)
POKE 123 , 3    :high order byte = $03
(New ROM users will substitute the otherPOKE locations.)
```

That takes care of the Start of BASIC Pointer but all those other pointers are still up where they used to be when BASIC started at $0401. They must also be moved down. We could use POKE to accomplish this however a NEW command will do them all at once. Therefore execute a NEW and then print FRE(0). You should be returned 7362 bytes free, an increase of 195 bytes! This may not seem like much but when those few extra bytes are needed to add those finishing touches it could come in very handy.

Now that the BASIC memory space has been increased does not mean that your program will automatically fill up this space. Besides, the NEW command removes your program anyways. One way to effectively use this modification is the following:

1. Power-up and LOAD your program.
2. Using UNLIST (described in Transactor #2, Vol. 2), record the program.
3. Increase memory using the steps outlined above, and...
4. Using the Merge procedure, also described in Transactor #2, bring the program back in by essentially merging it with empty space.

Now the first 195 bytes of your program will be resident in what used to be the second cassette buffer. Remember, you no longer have a second cassette buffer until you either reset the machine or re-adjust the pointers so don't try to use it or your program will be clobbered!

Sooner or later you will need to SAVE the program. However, this can no longer be done in the conventional manner. Take a look at the method used on the next page. Execute lines 100 through 220 directly on the screen exactly as shown. This is a modified SAVE. The SYS63153 accesses the tape write routines in ROM.

Now that a recording has been made there is one last problem. When the program is LOADed back into the PET, the Start of BASIC Pointer is not automatically set. It stays at 0400 but our program starts at 033A. POKE 122,59 and POKE 123,3 will fix this up.

A Short Note on Tapes

When a program is recorded on tape, the start and end addresses of that program are also recorded as part of the tape header. Therefore, when the program is LOADed, PET first looks at the start address and begins transfering bytes from tape into RAM. The first byte is transfered to the location specified by the start address. Increasing your memory using this method does NOT mean that your programs will LOAD to this extra space. However, they can be modified to do so. The information needed is in the article by Jim Butterfield on the first page of the first Transactor in Vol. 2.

```
100 POKE241,1
110 POKE247,58:POKE248,3
120 B=PEEK(124):POKE229,B
130 B=PEEK(125):POKE230,B
140 REM *** FIND SAVE NAME ***
150 A$=""
160 A$=STR$(PEEK(150)+256*PEEK(151))
170 A=VAL(A$)
180 A$="APPEND WEDGE"
190 B=PEEK(A):POKE238,B
200 B=PEEK(A+1):POKE249,B
210 B=PEEK(A+2):POKE250,B
220 SYS63153
230 END
```

Frans VanDuinen
Toronto
30Sep79

?LOAD ERROR

This note deals with program load errors on the 8K PET (Release 1), and how to recover from them.

Within two days after getting my PET (Nov78), I discovered the merits of back-up copies of programs and data files.
All I did was press PLAY and RECORD when the message said to press PLAY! It was only some twenty seconds, but it was sufficient to wipe out the file header and make the file inaccessible.

Ever since I've made sure to keep multiple copies, on the same tape for programs under development, on a dedicated back-up tape for programs that are more or less static.
So also the Journal program that I was developing back in July.
The only thing was, I was also working on another program, which that I accidently saved on the wrong tape. Scratch Journal version 0.6.

No real harm done, since I still had version 0.5, right? Wrong!
It just so happened that good old 0.5 had a load error.
I tried just about every thing, demagnetize & clean heads, both tape drives on my PET, LOAD vs STOP/shift, freeze cassette, rewind tape evenly, loosen screws in cassette housing and play on several other PETs. About the only thing I did not play with was head allignment (since the tape had been written with this allignment, it ought to be optimal for reading).

All to no avail. A load error I got, and load errors I kept on getting.
Yet I knew the data was there! There were some 3500 characters on that tape, most of which loaded correctly, but could not LIST, RUN or SAVE.

Since I still needed the Journal program, my choice was simple: salvage or re-develop and re-enter from memory.
So, with an ingenuity born of laziness (that being one of the prime qualifications for all programmers), I salvaged!

From Jim Butterfield's memory map (see The Transactor 9 vol 1 -or The Best of Transactor vol 1, pp 149-155 - and vol 2 #3) and my own disassembled listing of ROM, I had since acquired essential information on pointer fields and routines.

First let me introduce the cast of characters:

- .the program, it starts at loc 1024
- .the file header, at loc 634 for tape 1, loc 826 for tape 2
- .the load start point in the file header at offset 41
- .the load end point in the header at offset 43
- .the start of BASIC pointer at loc 122
- .the end of BASIC/start of variables pointer at loc 124
- .the end of variables pointer at 126
- .the start of available space pointer at loc 128
- .the Next Instruction Pointer (NIP) that precedes ever BASIC program instruction

- the BASIC Line Number (BLN) that is part of every statement
- the zero byte that identifies the end of each BASIC statement
- the End Of Program (EOP) marker, which is a dummy NIP of which at least the second byte contains zero.

After a normal load PET updates the end of BASIC pointer, the end of variables and the start of available space pointers , based on the end of load address from the file header.
Not so on a load error, the end of BASIC/start of variables pointer remains at 1024 (the start of BASIC pointer to be exact).
However, if variables are used they will be stored starting location 1024, i.e. smack on top of the program.
The following code will fix that (assuming LOAD from tape #1):

```
  ?Pe(637);Pe(638)  - which results in the values being printed
                      (remember, no variables may be used yet
   237  17            example (237+256*17=4589)

  Po124,237:Po125,17- set end of BASIC/start of variables
  Po126,237:Po127,17- end of variables
  Po128,237:Po129,17- start of available space.
```

Whew! Now we can use variables, since they will now be stored starting at 4589.

Next step is to rebuild the NIP pointer chain, where the NIP preceding every BASIC statement points to the NIP before the next statement, until we get to the dummy NIP that marks end of program.
SYS 50224 is an operating system routine that does just that.
However, it does that based on zero bytes. It assumes that every zero byte it encounters represents either the end of a statement or the end of the entire program. Thus if the load error introduces spurious zeroes, they may throw SYS 50224 for a loop, and the routine would store NIPs on top of valid data. If it does work, however, it's the by far easier method. If it does not work just reset the system and try the other possible approach.

The alternative is to write a one-line immediate routine that will follow the existing chain as far as possible, fix and continue.
The following routine will print a list of NIPs in ascending order, with line numbers (BLN), also in ascending order.
Any irregularity in either list indicates a load error.

```
  I=1025     initialize pointer to first NIP

  FoK=1TO900:J=Pe(I)+256*Pe(I+1):B=Pe(I+2)+256*Pe(I+3):
    ?I,J,B:I=J:Ne
```

This results in a list such as:

```
  1025   1052   10
  1052   1066   20
  1066   1099   21
  1099   1120   50
  1120   1156   60
  1156    585     70
  585     126652 12445
 BRK
```

Clearly 1156,1157 do not contain a valid NIP.
In this specific instance it appears that 1156,1157 are indeed the NIP (since the BLN looks to be correct), but the NIP has been clobbered due to the load error.
Frequently load errors are a result of timing errors.
This is where the read routine cannot handle the variations in tape speed that it perceives.
The result is commonly that the read routine reads more bits than were actually written to the tape. Conversely the routine may actually read fewer.

In my case the errors occasionally were wrong characters, or in some instances one or more characters missing or extra. Yet subsequent characters would still by and large be correct. In other words, it would appear that the read routine can recognize and synchronize with byte boundaries as recorded on tape.

The important thing here is that frequently a NIP address would be out by plus or minus one or two bytes, but so would the next one and the next.

To view what the internal representation of the program looks like, an immediate routine such as the following may be used:

```
  I=1155    -loc of last valid(?) NIP, minus 1 to check
              for presence of preceding zero

  FoK=ITOI+60:?Pe(K);:Ne - would result in

  0   132   4   70   0   145   137   32   49   48   48   44   50  48 ..
        NIP      BLN      ON   GOTO          1    0    0    ,    2   0 ..
```

(sorry, not the interpretation shown on the second line)

An other approach is to print the location number as well as its content. That makes it much easier to see what is going on:

```
  FoK=ITOI+60:?'R'K'r'Pe(K);:Ne

       'R'  - Reverse video on
       'r'  - Reverse video off
```

This would show alternately a location address (in reverse video), followed by it content:

```
1055   0   1056   132   1057   4   1058   4 ....... 1072   0
     1073   156   1074   4 ...
```

This facilitates checking the NIP actual location against the expected one (as contained in the preceding NIP).

A further variation on this to include two cursor-left characters:

```
  FoK=ITOI+60:?'R'K'rcl'Pe(K)'cl';:Ne

        cl - a single cursor-left character
```

This gets rid of the cursor-right the PET inserts after all numbers. Not only does it compress the listing, it also allows reuse of the statement (such as after a POKE, or for a different area) without occasional digits from the previous data showing through.

If an individual NIP is wrong, the most expedient solution is to POKE in a new value.
If, however, several subsequent NIPs are all out by the same amount, moving over the rest of the program may be indicated.
Visual inspection will have to indicate which bytes to suppress, or where to open it up.
Remember the main concern right now is to get the program in such shape that it can be LISTed and updated normally.

On compression, as in the following routine, bytes are copied into lower numbered locations. Thus if location 1112 is stored in 1111, 1113 in 1112, 1114 in 1113, etc., location 1112 has already been used by the time 1113 is stored into it, and thus may be safely clobbered
For example:

```
  FoI=1111TO4589:J=Pe(I+2):PoI,J:Ne
```

The +2 in the PEEK command causes everything to be moved over ('to the left') by two bytes.

Note that merely changing the +2 to -2 will not move everything two positions to the right.
Instead the leftmost two characters will be propagated through the entire section being moved. In the above example (with the +2 changed to -2) byte 1111 would be picked up first, and stored in 1113. Then 1112 would be stored in 1114. Next 1113 would be picked up to be stored in 1115. But 1113 contains the value from 1111 by now, and that is what would be deposited in 1115. Thus 1111 ends up in 1113, 1115, 1117, etc., with 1112 ending up in all the inbetween locations.

To handle such a shift right properly, the move has to start from the right, e.g.:

```
  FoI=4589TO1111STEP-1:J=Pe(I-2):PoI,J:Ne
```

That essentially sums up the totality of this technique for salvaging programs from load errors.
I do, however, sincerely hope that you'll never have to use it.

The IEEE-488 Bus

Jim Butterfield, Toronto

A parallel interface designed to exchange data with selected devices connected to the bus.

Many devices may be connected at the same time, but only the one that has been selected will send or receive data. For example, two printers and a disk unit could be connected to a bus; the Basic program would arrange to send to or receive from the various devices as desired.

Selection works by means of a "calling" system. Before sending data, the computer first sends a selection character, which commands the appropriate device to "listen". If the device is connected, it will acknowledge the command. Now the data is sent; each byte is acknowledged by the receiving device. Finally, the device is disconnected by an "unlisten" command. To receive data, the computer instructs the appropriate device to "talk". It then accepts data until the device signals "end of data", at which time the computer sends an "untalk" command.

Commands are distinguished from data by using a special line called ATN (attention). If the ATN signal is low (meaning true), the information being sent is a command: talk, untalk, listen, or unlisten. If the ATN signal is high (meaning false), the information being sent or received is data. In this system, only one direction is used: the computer sends ATN and the devices receive it. When ATN is low, all devices receive the commands, to see if they are being selected. When ATN is high, only the selected device will accept data.

Another line, called EOI (end or identify) is used to signal the last byte of data. It works in both directions: if the computer is sending, it signals EOI low (meaning true) with its last character; if the device is sending, it signals EOI low if it has no more data after the character it is sending.

When a device sends to the computer, it delivers each character only when invited by the computer. Similarly, the sending computer delivers characters only as fast as the device is ready for them. This flow is controlled by a "handshake" procedure.

An example of selection: When Basic executes OPEN 3,4, the IEEE-488 bus sets the ATN signal low and transmits hexadecimal 24 to the data lines, instructing device #4 to listen. If the device does not answer, Basic will return either DEVICE NOT PRESENT (ST=128 decimal) or WRITE TIMEOUT (ST=1). Subsequently, when the command PRINT#3,"HELLO" is given, the ATN signal is again set low and hex 24 transmitted to instruct #4 to listen; then ATN is set high, and the characters H, E, L, L and O are sent, with EOI set low during the transmission of the O character; finally, the ATN is set low and hex 3F is sent to cause the device to unlisten. Note that we haven't closed the file yet; but we have (temporarily) disconnected the device.

Using CMD on the IEEE-488 Bus

CMD does two things:

--it opens the appropriate device to "listen";
--it will divert output, normally directed to the screen,
to the IEEE-488 bus.

Both CMD activities are cancelled in any of three ways:

--preferred: when the bus is addressed with a normal PRINT# command;
--when any INPUT or GET is performed;
--when a Basic error is encountered.

It is best to avoid CMD within Basic programs, since any use of INPUT or GET will cancel it, and the programmer will have to arrange to repeat the CMD as necessary. Use PRINT# wherever possible. CMD is most useful in obtaining program listings. The preferred method:

```
OPEN 4,4        (identify the printer as device # 4)
CMD 4           (open the printer to listen & redirect output)
LIST            (do the listing)
PRINT#4         (cancel the CMD functions)
CLOSE 4         (close the file)
```

Never close a file until you have first cancelled the CMD command.

IEEE-488 Handshake: a brief technical description

The same handshake procedure is used for both command and data transmission.

The talker uses the DAV (Data available) line to indicate that valid data has now been placed on the bus. The listener uses two lines: NRFD (Not ready for data) to indicate that it is not yet willing to receive data; and NDAC (Data not accepted) to indicate that it has not yet taken data from the bus.

Transfer of data takes place in the following manner:

1. The talker initially places DAV high (meaning false) to indicate that data is not being sent yet. The listener will have NDAC low (meaning true) to indicate that no data is being received.
   If the listener is still working on something (say, printing the previous character) and can't accept data yet, it will set NRFD to low (true), meaning it's not ready.

2. The talker checks the NRFD and NDAC lines for both high (meaning false). If they are both high, something is wrong. If the computer is the talker, it will send DEVICE NOT PRESENT.

3. The talker places its data on the bus, but doesn't signal DAV low for data available until it sees the listener's NRFD is high, which signals that the listener is ready to receive data.
   The talker will wait forever - there is no timeout.

4. The data is ready, so the listener accepts and stores it. Then the listener sets NRFD low (true) and NDAC high (false) to acknowledge its receipt. The listener has a time limit on this activity: if it doesn't complete in 64 milliseconds, the talker will flag TIMEOUT ON WRITE.

5. The talker responds to the acknowledgement by setting DAV high, meaning that the data is no longer offered, and then clearing the data bus.

6. The listener detects the change in DAV, and realizes that its acknowledgement has been seen. It returns NDAC to low, completing the character exchange cycle. There is a time limit here: if the listener doesn't see DAV go high within 64 milliseconds, it will flag TIMEOUT ON READ.

Screen Print Routine

The following is a machine language subroutine that will copy the contents of the screen onto 2022/23 printers. It resides in the second cassette buffer and could be incorporated very neatly into any BASIC program where a hard copy of the screen might be required.

```
                       ; SCREEN PRINT ROUTINE
                       ; CALL WITH SYS 826
033A                            *=     $033A
033A             POINT          =      $1F
033A             RFLAG          =      $21
033A             COUNT          =      $22
033A             CR             =      $0D
033A             DEVICE         =      $D4
033A             CMD            =      $B0
033A             PRINT          =      $FFD2
033A             SLISTN         =      $F0BA                  ;LISTEN TO IEEE
033A             ATNOFF         =      $F12D
033A             BUSOFF         =      $FFCC
033A             SCREEN         =      $8000
033A             CASE           =      $E84C                  ;GRAPHICS OR LC
033A A9 80       SCPRT          LDA    #>SCREEN
033C 85 20                      STA    POINT+1
033E A9 00                      LDA    #<SCREEN               ;SET POINTER TO
0340 85 1F                      STA    POINT                  ;START OF SCREEN
0342 A9 04                      LDA    #4
0344 85 B0                      STA    CMD
0346 85 D4                      STA    DEVICE
0348 20 BA F0                   JSR    SLISTN
034B 20 2D F1                   JSR    ATNOFF                 ;OPEN PRINTER
034E A9 19                      LDA    #25                    ;25 LINES
0350 85 22                      STA    COUNT
0352 A9 0D       LINE           LDA    #CR                    ;START NEW LINE
0354 85 21                      STA    RFLAG                  ;RVS-OFF
0356 20 D2 FF                   JSR    PRINT
0359 A9 11                      LDA    #$11                   ;SHIFT FOR L/C
035B AE 4C E8                   LDX    CASE
035E E0 0C                      CPX    #12
0360 D0 02                      BNE    LOWER
0362 A9 91                      LDA    #$91                   ;SHIFT FOR GRAPHICS
0364 20 D2 FF    LOWER          JSR    PRINT
0367 A0 00                      LDY    #0
0369 B1 1F       MORE           LDA    (POINT),Y              ;SCREEN CHAR
036B 29 7F                      AND    #$7F                   ;STRIP RVS
036D AA                         TAX                           ;STORE
036E B1 1F                      LDA    (POINT),Y              ;CHECK RVS
0370 45 21                      EOR    RFLAG                  ;SAME AS LAST CHRPRINT
0372 10 0B                      BPL    SAME
0374 B1 1F                      LDA    (POINT),Y
0376 85 21                      STA    RFLAG                  ;LOG NEW RVS STATUS
0378 29 80                      AND    #$80
037A 49 92                      EOR    #$92                   ;BUILD RVS ON/OFF
037C 20 D2 FF                   JSR    PRINT
037F 8A          SAME           TXA                           ;RECALL PRINT CHAR
0380 C9 20                      CMP    #$20
0382 B0 04                      BCS    NOTALF
0384 09 40                      ORA    #$40                   ;CHANGE ALPHA ZONE
0386 D0 0E                      BNE    SEND                   ;BRANCH ALWAYS
```

```
0388 C9 40     NOTALF   CMP  #$40
038A 90 0A              BCC  SEND
038C C9 60              CMP  #$60
038E B0 04              BCS  GRAPH
0390 09 80              ORA  #$80
0392 D0 02              BNE  SEND                   ;BRANCH ALWAYS
0394 49 C0     GRAPH    EOR  #$C0
0396 20 D2 FF  SEND     JSR  PRINT                  ;PRINT CHAR
0399 C8                 INY
039A C0 28              CPY  #40                    ;LINE FINISHEDPRINT
039C 90 CB              BCC  MORE                   ;NO, DO IT AGAIN
039E A5 1F              LDA  POINT
03A0 69 27              ADC  #39                    ;YES, MOVE SCREEN POINTEF
03A2 85 1F              STA  POINT                  ;TO NEXT LINE
03A4 90 02              BCC  *+4
03A6 E6 20              INC  POINT+1
03A8 C6 22              DEC  COUNT                  ;ONE LESS LINE
03AA D0 A6              BNE  LINE                   ;BACK FOR ANOTHER
03AC A9 0D              LDA  #CR
03AE 20 D2 FF           JSR  PRINT
03B1 4C CC FF           JMP  $FFCC                  ;CLEAR BUS & QUIT
```

```
90 REM BASIC LOADER FOR SCREEN PRINT ROUTINE
100 FOR J = 826 TO 947
110 READ A : POKE J , A
120 NEXT
200 DATA 169,128,133,32,169,0,133,31
210 DATA 169,4,133,176,133,212,32,186
220 DATA 240,32,45,241,169,25,133,34
230 DATA 169,13,133,33,32,210,255,169
240 DATA 17,174,76,232,224,12,208,2
250 DATA 169,145,32,210,255,160,0,177
260 DATA 31,41,127,170 177,31,63,33,16
270 DATA 11,177,31,133,33,41,128,73
280 DATA 146,32,210,255,138,201,32
290 DATA 176,4,9,64,208,14 201,64,144
300 DATA 10,201,96,176,4,9,128,208,2
310 DATA 73,192,32,210,255,200,192,40
320 DATA 144,203,165,31,105,39,133,31
330 DATA 144,2,230,32,198,34,208,166
340 DATA 169,13,32,210,255,76,204,255
```

Frans VanTuinen
Toronto
27Sep79

Delete Rest of Instructions in Program

One of the more exciting, albeit undocumented, instructions on the PET is the 'Delete Rest of Instructionsin Program' or DRIP instruction.
If you haven't yet had occasion to use it, consider yourself lucky.

Under certain conditions the updating and replacing of a BASIC program instruction results in the dissapearance of that and all subsequent instructions in the program. As this seems to happen only after extensive (and not as yet saved) program changes have been made, the result is a lot of excitement.

This note describes what happens, when, how to recover from it, and covers a technique that seems to prevent it, but since I'm not sure how or why I can't be certain that the preventative measure always works.
The content of the note applies to Release 1 of the PET 8K system, the 'old ROM'.

The only cause that I am certain about is an interupt of a program occurs that is using the PRINT# to write to the IEEE bus. (Where my printer sits as device no 4.)
Any subsequent attempt to change the program frequently results in a 'DRIP'.
However, if I enter a 'CLR' command in between or cause an error, such as a RUN command with an invalid operand, a DRIP does not arize.

The symptoms are as follows. BASIC does somehow not recognize that the newly entered (updated) statement matches an existing number. BASIC therefore treats the updated instruction as a new one, and moves over the rest of the program to make room to insert this 'new' instruction.

However, BASIC makes other errors, that are even more severe. It inserts a zero in the high-order (second) position of the Next Instruction Pointer (NIP) of the first occurrence of the updated instruction, thus signalling the end of program. The part of the program that has been moved to allow for the insert of the 'new' instruction, has not had its pointers updated.

Fortunately, BASIC leaves the 'end of BASIC/start of variables' pointer intact, so variables can be used.

The solution of this problem is actually quite simple:

. remove the spurious zero
. rebuild the pointerchain.

I had visions of sophisticated program logic to reconstruct pointers based on the minimum and maximum number of bytes per instruction, zero bytes, relationships between statement numbers and visual inspection.
But once more, Jim Butterfield to the rescue! His list of routines identifies one called 'Corrects the chaining

between BASIC lines after insert/delete'!!!

As it turns out, it is very simple: if the address pointed by the current NIP, which itself is a NIP, contains a zero in the second byte, it is considered to be the end of program All other zeros starting at NIP+4 (to make allowance for the BASIC line number)are considered to represent the end of an instruction.

Thus by removing the zero that eroneously flags End Of Program, the pointer chain can be rebuilt by invoking this routine (SYS 50224).

Theoretically SYS 50224 could also be used to find the location on the End OF Program zero byte, as it leaves the address of the last NIP in locations 113,114.
Unfortunately, however, this is not a closed subroutine. It terminates by branching (JMP) into the PET's main command processing logic, rather than returning to the caller.
Locations 113,114 have been clobbered by the time control is returned to the keyboard

What can be used is an immediate command,such as:

```
I=1025:FoK=1TO1000:J=Pe(I)+256*Pe(I+1):?I,J:I=J:Ne
```

which will print a list of NIPs, that is in ascending order, upto and including the address of the faulty NIP, e.g.:

```
  3255      3272
  3272      3301
  3301      3356
  3356      55
  55        12356
BRK (stop as soon as dip occurs)
```

In this example locations 3356,3357 contain the faulty NIP. (These bytes contain 55 and 0 respectively.)
Now all that is required is the following:

```
   POKE 3357,1
   SYS  50224
```

The POKE instruction eradicates the value zero, and the SYS rebuilds the pointer chain.

In above example byte 3356 would originally have contained 13 (13*256+55=3383), however that is immaterial as the instruction that was there has been moved, while the SYS 50224 only makes the distinction zero or non-zero.

I hope this will allow others to deal with the DRIP instruction, however, the approach of frequent saving of program updates is still preferable!

Technical Comment on FOR/NEXT Loop Structures

Jim Butterfield,
Toronto

Recent remarks on popular BASIC implementations indicate that difficulties may be encountered if the programmer jumps out of a FOR/NEXT loop.

This would be very serious if true. The programmer doing a table search would be required to continue scanning the table even after finding the item he wants; or to use questionable practices such as meddling with the loop variable while still within the loop.

Fortunately, it's true only for a few complex situations - and these are easy to fix if you understand how the dynamic FOR/NEXT loop works. (Dynamic loops are those set up during an actual program run, as contrasted to pre-compiled loops which are checked out before the actual run starts).

When a dynamic interpreter, such as Microsoft BASIC, encounters a statement such as FOR J= ... it sets up internal tables to manage the loop. These internal tables contain such things as: where to return if a NEXT J is encountered; the identity of the loop variable (in this case, J); whether the loop is counting up or down, etc.

These tables will remain until one of three things happens. If the loop goes through its complete range (by encountering a suitable number of NEXT J statements); if a new FOR J statement is found; or if a higher priority loop is terminated for either of the previous reasons.

The last rule is very sensible, and it's worth a closer look. Suppose we have set up a sequence of commands such as:

```
FOR I= ...   : FOR J= ...   : FOR K= ...
```

and suppose the computer, while dealing with these three loops finds a new FOR I= ... statement. It very wisely says, in its own computerese, "OK - looks like the big loop is being restarted; so the little ones are finished too". And it promptly terminates the J and K loops, removing the tables from its memory.

Exactly what we want - but there are a couple of hidden gotchas that the user must know about when he gets into tricky coding routines.

The easiest one to spot is the situation where every loop has a different variable name. The first loop is, say, FOR A ... the next on, FOR B ... and the programmer continues through the alphabet with each loop. His idea is good: he can analyze how each loop has behaved, for each variable remains untouched for his examination. But each time he jumps out of a loop, the loop tables remain in memory, using up valuable text space. He'd be much better off to give at least his outer loops the same variable name, and reclaim that space.

The second problem spot is a little more subtle, and an example would best illustrate it.

Here's a simple program to input a string, extract the individual words (eliminating single or multiple spaces), and print them:

```
100 INPUT S$               get the string
110 K=1                    mark start of string
120 FOR J=K TO LEN(S$)
130 IF MID$ (S$,J,1) <> " " GOTO 150   skip spaces
140 NEXT J
150 IF J > LEN(S$) GOTO 900
160 FOR K= J TO LEN(S$)
170 IF MID$ (S$,K,1) = " " GOTO 200   scan to space or end
180 NEXT K
200 PRINT MID$ (S$,J,J-K)
800 IF K <= LEN(S$) GOTO 120
900 END
```

The program works quite well and isn't hard to follow. It should be noted that if either the J or K loops run to completion, the variable will have a value of LEN(S$)+1; this is intended and allowed for in lines 150 and 800.

Before we extend this program into catastrophe, let's note one thing: by the time the program reaches line 200, both the J and K loops will still be open most of the time - we "jumped out" of both of them. No real problem; when we go back to 120, the new FOR J ... will cancel them both.

Now let's get into trouble. We may be writing a little ELIZA here, and we want to check the word we've found against a table of keywords so as to pick a suitable reply. We'll assume a table of twenty keywords, and start to build a search loop. Replacing line 200, we start a new loop:

```
200 X$ = MID$ (S$,J,K-J)     get word
210 FOR I= 1 TO 20
  ......
```

Our loop is now three deep - J and K are still considered active, remember? No problem with three-level loops; we're still OK.

But here's where we might get clever and wreck everything. We need to preserve K - that's where our search for the next word will start. But J has served its purpose, and could be used again, right? Well .. let's see.

This table of 20 words is really a double table. It contains pairs of words such as "I", "YOU", or "MY", "YOUR". To make our computer talk we must spot a word from either column, and switch in the word from the opposite column (so that "I HAVE FLEAS"). So we need one more loop to search over the two columns.

Let's be clever and use J, since we have decided that it isn't needed any more at this point. We code:

```
        220 FOR J=1 TO 2
        230 IF X$=T$(I,J) THEN X$=T$(I,3-J):GOTO 400   swap word
        240 NEXT J
        250 NEXT I
        400 PRINT X$;" ";               repeat word
```

Suddenly everything stops working, and the whole world tumbles down around our program. What happened?

Let's stop and analyze. Just before executing line 220, the computer had three active loops, with variables J, K, and I. Now it reaches line 220, and what does it see? A loop Based on J, the "biggest" loop! So what does it do? It cancels the K and I loops, of course, and starts a new J loop.

When we reach line 250, the computer sees NEXT I - but it no longer has an active FOR I= loop, and you get a NEXT WITHOUT FOR error notice.

The rule here is slightly more complex, but not too tough. If you use J as an "outer" loop variable, never use it for an inner loop. If we reversed I and J in the coding from 210 to 250, we'd have no problem. Try to think in terms of the hierarchy of loops, and you can make sure that a given variable is used only at its proper hierarchy level.

Let's try to put the rules together and create a tiny ELIZA, polishing up some of the coding as we go. You'll have fun adding your own features to it.

```
        100 DIM T$(1,4)          two by five array
        110 DATA ME,YOU,I,YOU,MY,YOUR,AM,ARE,MYSELF,YOURSELF
        120 FOR J=0 TO 4
        130 FOR K=0 TO 1
        140 READ T$(J,K)
        150 NEXT K
        160 NEXT J
        170 INPUT S$
        180 K1=1
        190 FOR J= K1 TO LEN(S$)
        200 IF MID$ (S$,J,1) = " " THEN NEXT J
        210 J1=J
        220 IF J > LEN(S$) GOTO 900
        230 FOR J= J1 TO LEN(S$)
        240 IF MID$ (S$,J,1) <> " " THEN NEXT J
        250 K1=J
        260 X$ = MID$ (S$,J1,K1-J1)
        270 FOR J=0 TO 4
        280 FOR K=0 TO 1
        290 IF T$(K,J) = X$ THEN X$ = T$(1-K,J):GOTO 320
        300 NEXT K
        310 NEXT J
        320 PRINT " ";X$;
        330 IF K1 <= LEN(S$) GOTO 190
        340 PRINT "?"
        900 GOTO 170
```

Note that the outer most loop is now always called J, the next down always K. I've tightened up the array to use the zero rows and columns to save memory; and the search loops are a little faster.

Even though the program is riddled with premature loop exits, there are no problems. Just observe a few simple rules, and you'll have efficient and trouble-free loops.

Attention Multi-Peripheral Users

It has been found that when more than one peripheral is connected to the IEEE-488 buss, a slight problem may occur should one device be ON and the other OFF. Take for example the following sequence of events:

PET	ON
Printer	OFF
Disk	OFF

```
          Type: OPEN 1 , 8 , 4 , " 0:DISKFILE , S , W "

 PET responds: ?DEVICE NOT PRESENT ERROR
```

This is of course what you would expect. Now power up the Printer, leaving the disk unit OFF.

```
          Type: OPEN 1 , 8 , 4 , " 0:DISKFILE , S , W "

 PET responds: READY.
```

But the disk is OFF or essentially "NOT PRESENT". Therefore:

```
PRINT#1,"FILE DATA"
```

...will result in lost data.

There is, however, a test that can be made to protect against lost info. The status word, ST, is set to -128 whenever the above situation occurs. Therefore the following test could be included immediately after the OPEN statement:

```
IF ST < 0 THEN PRINT "DEVICE NOT PRESENT"
```

Don't be alarmed since any programs using disk file access are usually loaded from the disk, the disk will be turned ON anyways and the above situation will probably never be encounterd.

commodore

comments and bulletins
concerning your
COMMODORE PET™

The Transactor

Vol. 2
BULLETIN #6

PET™ is a registered Trademark of Commodore Inc.

November 30, 1979

Inside the 2040 Disk Drive

Jim Butterfield, Toronto

Yes, you can look at the programs inside the 2040. But unless you're strong in machine language - and have a bit of hardware background - it won't make much sense.

There are two processors in there. One looks out toward the PET .. I'll call it the IEEE processor; the other looks in toward the disk mechanics .. this one I'll call the disk processor. Each processor has a completely different set of programs. The two processors talk to each other by sharing a little memory space: about 4K of RAM is common to both micfoprocessors.

The IEEE processor is relatively easy to look into. You have the M-R, or memory read, command which allows you to look at the whole 64K memory space of this processor. Not all of this is actually fitted with memory, of course. As far as I can tell, ROM occupies hex locations E000 to FFFF. There's RAM in zero page; and the RAM which is shared with the disk microprocessor is in hex 1000 to 1FFF. The 6532 PIA chips seem to be in ~~the~~ addresses $0200 to $03FF.

To analyze a completely unknown 650X program, you must start by inspecting ~~locatio~~ locations $FFFA to $FFFF. This gives you the three main vectors, for NMI, Reset, and INT. As far as I can tell, NMI isn't used - the vector points at non-existent memory. Reset is of course used; in my 2040 it points at F480, and that's where the main code for initialization begins. It looks to me as if the interrupt line must be kicked by the IEEE ATN (attention) line: when I follow the vector (FDDE) in my machine, it looks like an IEEE handshake is taking place.

That's all vefy well for the IEEE processor, but how can you get a look at the inner, disk processor? I had trouble with this one. until one day I discovered that the IEEE processor can download the disk processor - via the shared RAM - and make it execute this new code! So all that's needed is a little program to tell the disk processor to copy part of its memory to the shared RAM space. where it can be examined by using the M-R command.

I couldn't get this to work, however, until I discovered the vital missing link. The shared RAM, which is seen at locations 1000 to 1FFF by the IEEE processor, is seen in a completely different location by the disk processor! .. in this case, hex 0400 to 13FF. The hardware just "maps" the memory into a different location. I might never have spotted this if the memories had not overlapped; but a little rummaging around and tearing of hair showed that my early programs seemed to be putting data into the wrong buffer. Eventually, the penny dropped, and the system became clear.

I'm far from being able to give details about the inner secrets of the 2040. But with the enclosed DISK PEEK program, you too can rummage around in there - in either processor's memory space - and come up with interesting data.

```
100 PRINT"■DISK MEMORY DISPLAY    JIM BUTTERFIELD"
110 DATA77,45,87,0,18,16,162,0,189
120 DATA157,64,06,232,224,16,208,245,76,193,254
130 FORJ=1TO9:READX:C$=C$+CHR$(X):NEXTJ
140 FORJ=1TO11:READX:D$=D$+CHR$(X):NEXTJ
150 PRINT"■THERE ARE TWO PROCESSORS:"
160 PRINT"  1) THE IEEE PROCESSOR;"
170 PRINT"  2) THE DISK PROCESSOR;"
180 INPUT"WHICH DO YOU WANT TO PEEK (1 OR 2)";D
190 PRINT"INPUT MEMORY ADDRESS"
200 PRINT"IN HEXADECIMAL:":OPEN1,8,15
210 PRINT"  ▮▮▮▮▮                               ⌝"
220 INPUTZ$
230 PRINT"⌝";:IFLEN(Z$)<>4THENGOTO210
240 FORJ=1TO4:Y=ASC(MID$(Z$,J))
250 IFY<58THENY=Y-48
260 IFY>64THENY=Y-55
270 IFY<0ORY>16GOTO210
280 Y(J)=Y:NEXTJ:K=0:PRINT"▮▮▮▮▮▮";
290 ONDGOTO300,320:GOTO180
300 U=Y(3)*16+Y(4):V=Y(1)*16+Y(2)
310 GOSUB360:GOTO210
320 PRINT#1,C$;CHR$(Y(3)*16+Y(4));CHR$(Y(1)*16+Y(2));D$
330 PRINT#1,"M-W";CHR$(4);CHR$(16);CHR$(1);CHR$(224)
340 PRINT#1,"M-R";CHR$(4);CHR$(16):GET#1,X$:IFX$=CHR$(224)GOTO340
350 U=64:V=18:GOSUB360:GOTO210
360 PRINT#1,"M-R";CHR$(U);CHR$(V)
370 GET#1,X$:IFX$=""THENX$=CHR$(0)
380 PRINT" ";:X=ASC(X$)/16
390 FORJ=1TO2:X%=X:X=(X-X%)*16:IFX%>9THENX%=X%+7
400 PRINTCHR$(X%+48);:NEXTJ
410 U=U+1:IFU=256THENU=0:V=V+1
420 K=K+1:IFK<8GOTO360
430 Y(0)=0:Y(4)=Y(4)+8:J=4
440 IFY(J)>15THENY(J)=Y(J)-16:J=J-1:Y(J)=Y(J)+1:GOTO440
450 PRINT:PRINT"  ";:FORJ=1TO4:Y=Y(J):IFY>9THENY=Y+7
460 PRINTCHR$(Y+48);:NEXTJ:PRINT"⌝":RETURN
```

```
**** THE LAST THREE ITEMS IN LINE 120 (76,193,254) MAY BE CHANGED
    IF NECESSARY TO A RESET SEQUENCE OF 108,252,255    ****
```

Printer Formatting

There has been a bug detected with the formatting feature of the 2022 and 2023 Printers but fortunately, Kim Lantz of North Sydney, Nova Scotia, has found the fix.

It seemed that setting up the first format was no problem, but changing to a second format was. When PRINTing to the printer, the last character to be sent to a line is a CRLF. This is done for obvious reasons but, the Carriage Return is printed on the current line and the Line Feed is printed on the next line. The Line Feed character is of course not printed on the paper but the printer "sees" it as the first character of the new line and when the printer is anywhere but the absolute beginning of a line, it doesn't like changing the format.

Therefore, anything that is output to secondary address 1 or the printer should be followed by...

```
;CHR$(13);
```

```
For e.g.        OPEN 1,4,1
                PRINT #1, X;CHR$(13);
                PRINT #1, "PET";CHR$(13);
```

...especially when the format string is about to be changed. This is also true for secondary address 0.

The above can of course be shortened by first equating R$ to CHR$(13) and using R$ in place of CHR$(13). Also the first semi-colon is not necessary when preceeded by a closing quote or another string variable but is necessary when following numeric variables.

However, the general idea is to keep the printer in the 0'th position after a carriage return when the format string is to be changed.

Bits and Pieces

The IF..THEN statement can be very useful in avoiding certain unexpected hazards. Two in particular are 1) argument outside range and 2) dividing by zero.

The ON..GOTO statement has a limited range on its argument; 1 to 255. Zero causes execution to drop through to the next line but values negative or over 255 will cause an error and a forced break. Protecting against this is easy and often a good idea.

```
500 IF X > -1 AND X < 256 THEN ON X GOTO...   (GOSUB)
501 REM -CODE FOR X = 0
```

Executing a 'THEN' causes PET to interpret the code following as a "new line". A 'THEN' can therefore be followed by any BASIC statement including another 'IF..THEN'.

Dividing by zero will fail for obvious reasons. Preceeding a possible trouble spot with a denominator test will protect against ?DIVISION BY ZERO ERROR.

```
600 IF D <> 0 THEN IF N/D <> 0 THEN
    IF N2/(N/D) > 1 GOTO 880
```

Another hidden gotcha that has been known to cause bald spots is the peculiar behavior of the FOR..NEXT loop. Code within a FOR..NEXT loop will always execute at least once regardless of the initial loop counter values.

```
700 IF J > 0 THEN FOR X = 1 TO J:...: NEXT
```

...will guard against unwanted looping. Only one problem; the entire loop must be squeezed into one line otherwise GOTOs must be used.

One further note; a STEP size of zero will cause endless looping. Depending on the extent of STEP use, testing of STEP variables might be advisable.

Bullet-Proof INPUT

As you know, INPUT allows the cursor control characters to be typed which can really foul up a program especially when user infallibility is of importance. The following subroutine could substitute for INPUT:

```
5000 POKE  167 , 0
5010 A$ = ""
5020 GET B$ : IF B$ = "" THEN 5020
5030 IF ( ASC ( B$ ) AND 127 ) > 31 THEN
     PRINT B$; : A$ = A$ + B$
5040 IF B$ = CHR$( 13 ) THEN POKE 167 , 1 : RETURN
5050 GOTO 5020
```

Line	Explanation
5000	The only drawback using GET over INPUT was that a simulated cursor was required. POKE 167 , 0 (548 in old ROM) conveniently turns the PETs cursor on.
5010	Sets A$ (the input string) to null string.
5020	Standard "GET loop".
5030	This test masks out all of the cursor control keys, allowing only numeric, alpha and graphics to PRINT.
5040	Test for 'RETURN' key, yes...turn cursor off, exit.

Extra tests could be inserted between 5030 and 5040 to include cursor left/right and/or delete. Also, a character counter might be incorporated to limit the input string length.

Floating Binary

The following program by Jim Butterfield shows the true value of a decimal floating point number as stored by PET in floating binary. The program illustrates how some decimal values cannot be represented in binary exactly. Try values of 1.1, 1.2 and 1.7

```
100 PRINT : INPUT V
110 PRINT INT(V);".";
120 V = (V - INT(V)) * 10 : IF V=0 GOTO 100
130 PRINT CHR$ (V+48);
140 GOTO 120
```

THE WALL STREET JOURNAL

"No! I don't want any middlemen, put me right through to your computer."

Infinitely Long PET Programs

Henry Troup, Toronto

Even with a 32K PET, it is sometimes desirable to handle programs in sections, loading as necessary. Loading a program from a program does not change any pointers so variables are preserved. However, any new program must be the same length or shorter than the first one loaded!

In order to make certain details such as filenames and the disk commands transparent to the user, you may want a small front end loader or menu program to call in subsequent code.

However, if the program coming in is longer than the menu driver, the variable pointers will be pointing right into the middle of your program. As soon as any new variables are created, the program is disturbed, and a machine crash may result. Certainly this will cause a non-recoverable error. This may be avoided by including this line as the first of the program:

```
POKE 42, PEEK (201) : POKE 43, PEEK (202) : CLR
```

This resets the bottom of text pointer and CLR cleans up all the other pointers. The program will now run safely.

If a program containing this line at the beginning is RUN and then STOPped, and modifications are made, DO NOT re-run without branching around this line. If you do, the end of text pointer will be improperly set by the POKEs and you might be in for trouble.

Of course using this method does not allow passing data between the programs. Should this be required, you could set up a disk file with the necessary data and then call it back in, or simply exclude the use of the above line and make sure the first program is the biggest!

Screen I/O

Some of you may have experienced problems PRINTing characters to the screen over top of characters that are already there. Try, for example, the following program:

```
100 ?"home";
110 FOR J = 1 TO 10
120 ?"+++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++"  (approx 60)
130 NEXT
140 ?"home";
150 FOR J = 1 TO 10
160 ?"******************"
170 NEXT
180 END
```

So why the extra line feeds? PET maintains a "line wrap" table in RAM which determines whether the line is a single or a double line or more precisely, over or under 40 characters. This is done for things like INPUT and for entering BASIC.

For upgrade ROMs the wrap table is kept in RAM from 00E0 to 00F8 (decimal 224 - 248), 0229 to 0241 (dec 553 - 577) for old ROMs.

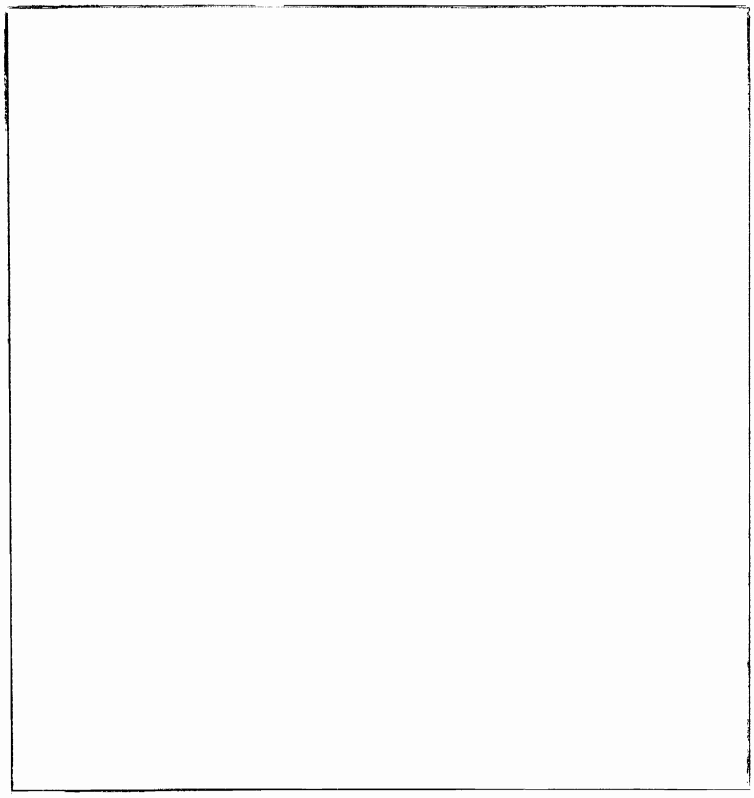
So how do we eliminate these dastardly line feeds? You could play with "cursor ups" but if some lines are double and others single this can be somewhat cumbersome especially if your PRINT strings end at column 40. The alternative is to alter the information held in the line wrap table.

The table consumes 25 bytes of RAM; one byte for each line on the screen. These bytes will contain the lines high order memory address. As you know, screen memory starts at hex 8000 and continues to hex 8FFF (see memory map). The home position of the screen is therefore at hex 8000. Since the address of a line is taken from the beginning of that line, the address of the top line will be $8000 ($ = hex). The high order address is simply $80 and the decimal equivalent of $80 is 128. The PEEK of the first location of the wrap table will return a 128 which is of course decimal.

The following relates wrap table decimal values (PEEK values) to the hex address of the first character space of each screen line. Remember, only the high order part of the address is of any concern to the wrap table. Also, the table resides in different locations for old and new ROMs so for now we'll call them locations 1 through 25.

Wrap Table	Hex addr. of
1: 128	8000
2: 128	8028
3: 128	8050
4: 128	8078
5: 128	80A0
6: 128	80C8
7: 128	80F0
8: 129	8118
9: 129	8140
10: 129	8168
11: 129	8190
12: 129	81B8
13: 129	81E0
14: 130	8208
15: 130	8230
16: 130	8258
17: 130	8280
18: 130	82A8
19: 130	82D0
20: 130	82F8
21: 131	8320
22: 131	8348
23: 131	8370
24: 131	8398
25: 131	83C0

Blank Screen (single lines)



If the wrap table PEEK values were represented in binary, the eighth bit would be set to 1 in each case:

```
128 = 1 0 0 0  0 0 0 0
131 = 1 0 0 0  0 0 1 1
```

This means that the corresponding line is single or has less than 40 characters on it.

When characters outputing to the screen wrap around the right side, PET considers these characters as part of the above line. Take, for example, the top two lines (lines 1 & 2). The screen is cleared and a string of 52 characters are PRINTed from the home position, past column 40 and onto line 2. Line 2 is now considered part of a double line but more importantly, line 1 is considered a single line of double length. The wrap table records this by setting the eighth bit of the value corresponding to line 2 to zero. The top two lines are now treated by PET as a single line hence the extra line feeds. This is most noticeable when using the screen editor on program lines of length greater than 40.

The wrap table values for the example program would be:

Wrap Table	Hex addr. of	Program Example
1: 128	8000	******************++++++++++++++++++++++
2: 0	8028	+++++++++++++++++++++++
3: 128	8050	******************++++++++++++++++++++++
4: 0	8078	+++++++++++++++++++++++
5: 128	80A0	******************++++++++++++++++++++++
6: 0	80C8	+++++++++++++++++++++++
7: 128	80F0	******************++++++++++++++++++++++
8: 0	8118	+++++++++++++++++++++++
9: 129	8140	******************++++++++++++++++++++++
10: 1	8168	+++++++++++++++++++++++
11: 129	8190	******************++++++++++++++++++++++
12: 1	81B8	+++++++++++++++++++++++
13: 129	81E0	******************++++++++++++++++++++++
14: 2	8208	+++++++++++++++++++++++
15: 130	8230	******************++++++++++++++++++++++
16: 2	8258	+++++++++++++++++++++++
17: 130	8280	******************++++++++++++++++++++++
18: 2	82A8	+++++++++++++++++++++++
19: 130	82D0	******************++++++++++++++++++++++
20: 2	82F8	+++++++++++++++++++++++
21: 131	8320	
22: 131	8348	
23: 131	8370	
24: 131	8398	
25: 131	83C0	

The Solution

If PRINTing on double lines has thrown a wrench into your program, the easiest solution is make all lines single. Insert the following lines into the example program and RUN it:

```
New ROM:    143 FOR J = 224 TO 248 : X = PEEK (J)
            145 POKE J, X OR 128 : NEXT

Old ROM:    143 FOR J = 553 TO 577 : X = PEEK (J)
            145 POKE J, X OR 128 : NEXT
```

The "OR" function in line 145 is used to set the eighth bit to 1, thus altering the wrap table such that PET considers all lines as single.

Random Access File Indexing

For those writing programs that have random access record handling, a routine has been developed by Jim Hindson of Burlington, Ontario. The routine is basically an algorithm that will convert a record number into the location of the record within the file.

2040 Disk — Jim Hindson

Index and Main Record locations for:

a) Index file of records at 10 records per sector
b) Main file of records at 3 records per sector

Task A - Divide available sectors into sectors to be used as the index file and sectors to be used for the main file and to obtain an equal number of each record type (index and main) on a diskette.

For 10 index records/sector and 3 main records/sector, one plan would be as follows:

Index Records

Record No.	Track No.	Sector No.
1 - 200	1	1 - 20
201 - 400	2	1 - 20
401 - 600	3	1 - 20
601 - 800	4	1 - 20
801 - 1000	5	1 - 20
1001 - 1200	6	1 - 20
1201 - 1400	7	1 - 20
1401 - 1500	8	1 - 10

Main Records

Record No.	Track No.	Sector No.
1 - 567	9 - 17	0 - 20
Track 18 reserved for directory		
568 - 927	19 - 24	0 - 19
928 - 1251	25 - 30	0 - 17
1252 - 1500	31 - 35	0 - 16

Each of the four Main Record areas will be known as track zones.

Note (1) Although sector 0 is available on tracks 1 - 8, it is not used in this example.
(2) Sector 15 & 16 of track 35 not used

Task B - Write a subroutine to convert any record number (say NR) to the track, sector and record number within the sector.

Variable Identification

```
   NR : Number of the Record, the location of which is
        required
TR(1) : Index file track number for NR
TR(2) :  Main file track number for NR
SN(1) : Index file sector number for NR
SN(2) :  Main file sector number for NR
SR(1) : Index file record number for NR (1-10)
SR(2) :  Main file record number for NR (1-3)

Z(1) - Z(4) : delimiters for the track zones which have a
              different number of available sectors
         B1 : number of records per track ( within a track
              zone )
          A : B1-1
          C : 1 less than the lowest track number in a
              track zone
```

By using this subroutine it is not necessay to carry any information on the index file about where the record is located on the main file.

Subroutine Convert

Fed NR, this subroutine will return TR(1), SN(1), SR(1) and TR(2), SN(2), SR(2) for a 1500 record file of 1500 index records at 10 records/sector and 1500 main records at 3 records/sector.

```
40500 REM *** SUBROUTINE CONVERT ***
40501 REM +++ FIND INDEX FILE LOCATION +++
40502 Z = (NR + 199)/200
40505 TR(1) = INT(Z)
40510 Z1 = NR - ((TR(1) - 1)*200)
40515 Z2 = (Z1 + 9)/10
40520 SN(1) = INT(Z2)
40525 Z3 = Z1 - ((SN(1) - 1)*10)
40530 SR(1) = INT(Z3)

40550 REM +++ FIND MAIN FILE LOCATION +++
40549 Z(1) = 567 : Z(2) = 927
40552 Z(3) = 1251 : Z(4) = 1506
40560 FOR J = 1 TO 4                        :find track
40565 IF NR - Z(J) <= 0 THEN 40576           zone
40575 NEXT J
40576 NZ = NR
40578 IF J > 1 THEN NZ = NR - Z(J-1)        :convert to number
                                             within track zone
40580 ON J GOTO 40591,40592,40593,40594
40591 A=62 : B1=63 : C=8  : GOTO 40600      :define
40592 A=59 : B1=60 : C=18 : GOTO 40600       zone
40593 A=53 : B1=54 : C=24 : GOTO 40600       parameters
40594 A=50 : B1=51 : C=30
```

```
40600 Z =(NZ + A)/B1                     :find
40605 TR(2) = INT(Z)                      track,
40610 Z1 = NZ - ((TR(2) - 1)*B1)
40615 Z2 = (Z1 + 2)/3
40620 SN(2) = INT(Z2)                     sector,
40625 Z3 = Z1 - ((SN(2) - 1)*3)
40630 SR(2) = INT(Z3)                     record
40640 TR(2) = TR(2) + C                  :compensate for # of
40650 SN(2) = SN(2) - 1                   tracks in lower and
                                          availabilty of
                                          sector 0.

40660 RETURN
```

Editor's Note

You may be asking, "Why an index file routine and a main file routine when the whole purpose is to do away with the index ?". The index file really doesn't do any indexing and might have been called a 'sub-main' file. Jim developed the program for his own use and found it more efficient to split each entry into 2 files: an "index" file for name and Social Insurance Number and a main file for any remaining info (address, phone #, etc.). It was anticipated that 110 characters would be required for each entry. With 255 byte sectors, this would impose a restriction of 2 entries per sector, wasting 35 bytes. The maximum would also be restricted to 2*670 (blank disk has 670 sectors) or 1340. By splitting up the entries into 25 and 85, each sector or block can filled to capacity allowing 1500 entries. This figure could also be increased as some blocks are unused.

This method of indexing has only one drawback: NR. That is, each item in the file must have a number (1, 2, 3...etc.) that may be irrelevant to the data being recorded. Therefore, access to a record requires entry of the corresponding 'NR' and in the above example NR has a range of 1 to 1500.

This would be ideal for applications such as a mailing list where each subscriber has a number, but for a inventory it becomes somewhat impracticle since 'NR' will probably not be your part number. However, Jim's method is still simpler than recording disk co-ordinates. Consider this; have PET assign "NR's" to the record element that will be primarily used for record recall. For example:

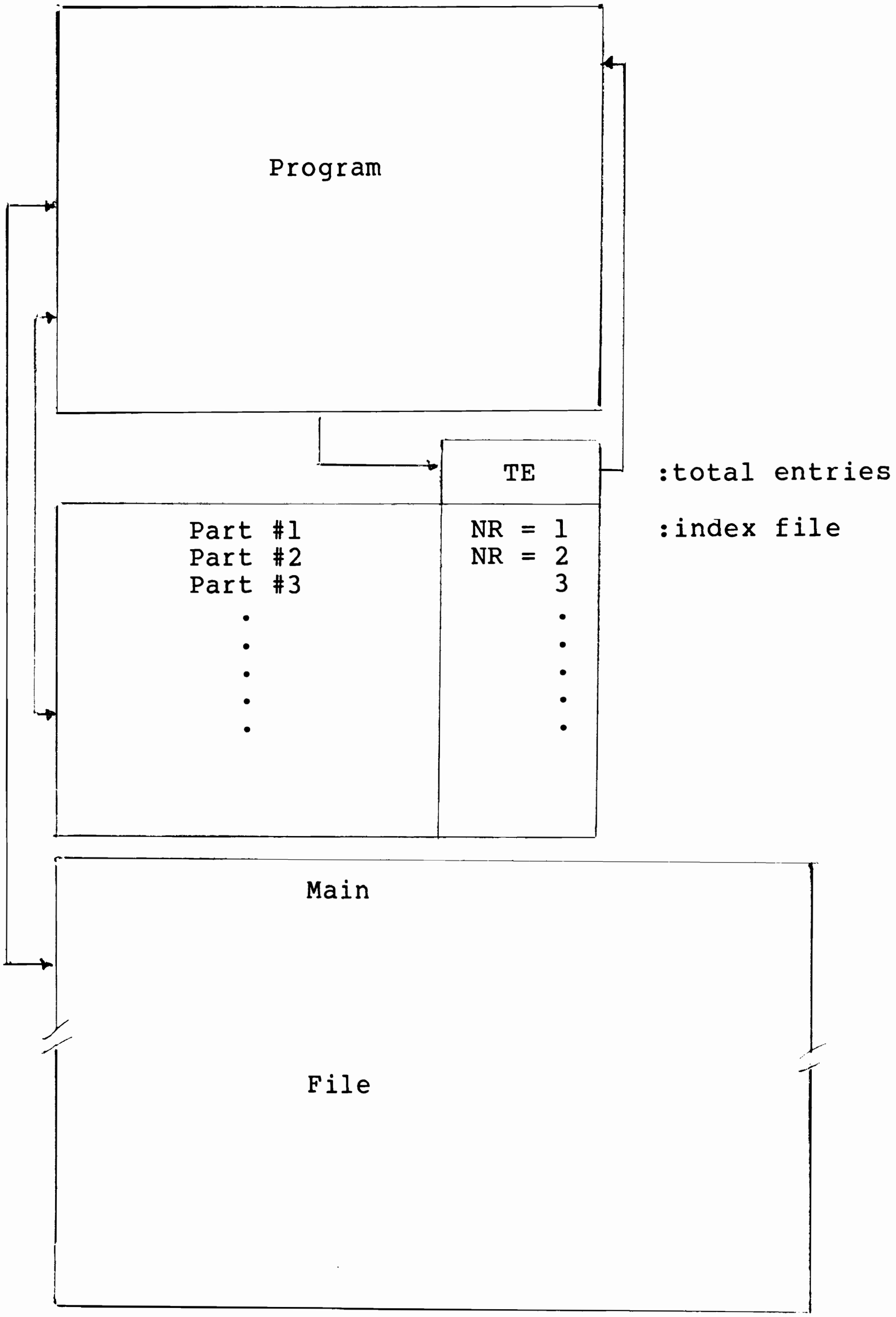
```
(Part #1) , X
(Part #2) , X+1
(Part #3) , X+2
```

...and so on. This information could be stored in an random index file along with the total number of entries (TE) so that PET would know where to start assigning new NR's to new entries.

With the desired Part # entered, the index file could be searched, NR extracted and passed into Jim's main file subroutine.

Once the track and sector co-ordinates are determined (TR(2) and SN(2)), they can now be inserted in the Block-Read command and SN(2) in the Buffer-Pointer command for rapid record access. You might also consider using Bill Maclean's Block Get routine for transfering data from disk to PET.

System layout for above:



commodore

comments and bulletins
concerning your
COMMODORE PET

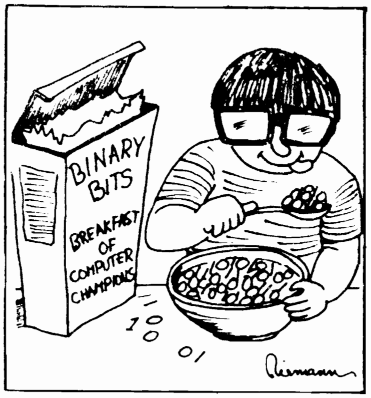
The Transactor

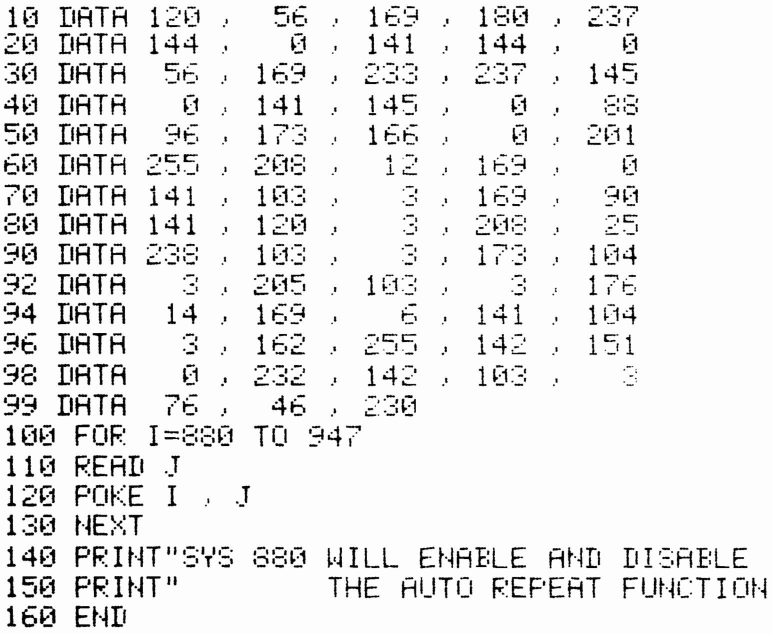
Vol. 2
BULLETIN # 7
Dec. 31, 79

PET^{tm} is a registered trademark of Commodore Inc.

This months Transactor is a collection of all the charts and tables concerning PET and computers in general. Some have appeared in previous Transactors but flipping and finding can be a chore. Therefore a handy reference was thought to be in order.

For The Best of The Transactor Volume 2, this reference material has been moved to the back for quick access.




```
10 DATA 120 ,  56 , 169 , 180 , 237
20 DATA 144 ,   0 , 141 , 144 ,   0
30 DATA  56 , 169 , 233 , 237 , 145
40 DATA   0 , 141 , 145 ,   0 ,  88
50 DATA  96 , 173 , 166 ,   0 , 201
60 DATA 255 , 208 ,  12 , 169 ,   0
70 DATA 141 , 103 ,   3 , 169 ,  90
80 DATA 141 , 120 ,   3 , 208 ,  25
90 DATA 238 , 103 ,   3 , 173 , 104
92 DATA   3 , 205 , 103 ,   3 , 176
94 DATA  14 , 169 ,   6 , 141 , 104
96 DATA   3 , 162 , 255 , 142 , 151
98 DATA   0 , 232 , 142 , 103 ,   3
99 DATA  76 ,  46 , 230
100 FOR I=880 TO 947
110 READ J
120 POKE I , J
130 NEXT
140 PRINT"SYS 880 WILL ENABLE AND DISABLE
150 PRINT"         THE AUTO REPEAT FUNCTION
160 END
```

Block Get Routine

Bill MacLean
BMB Compuscience, Canada

The machine language routine below can be used with direct access routines to transfer the contents of a disk buffer into PET memory. BASIC 2.0 only.

```
100 FOR J = 826 TO 914
110 READ A
120 POKE J , A
130 NEXT
200 DATA 169 ,   0 , 133 ,  52 , 169 , 127
210 DATA 133 ,  53 ,  32 , 248 , 205 ,  32
220 DATA 159 , 204 ,  32 , 210 , 214 , 165
230 DATA  18 , 240 ,   3 ,  76 ,   3 , 206
240 DATA 165 ,  17 , 133 , 210 , 169 ,   0
250 DATA 133 ,   1 , 169 , 127 , 133 ,   2
260 DATA 166 , 210 ,  32 , 198 , 255 ,  32
270 DATA 207 , 255 , 201 ,  10 , 240 , 249
280 DATA 201 ,  13 , 240 ,   8 , 160 ,   0
290 DATA 145 ,   1 , 230 ,   1 , 208 , 237
300 DATA 165 , 210 ,  32 , 204 , 255 ,  32
310 DATA 248 , 205 ,  32 , 159 , 204 , 160
320 DATA   0 , 165 ,   1 , 145 ,  68 , 200
330 DATA 169 ,   0 , 145 ,  68 , 200 , 169
340 DATA 127 , 145 ,  68 ,  96 ,  66
```

Note: For 16K machines, change the three 127's to 63's.

The command to use this program is:

```
SYS 826 , LF# , A$
```

It replaces:

```
INPUT# (LF#), A$
```

It works with ASCII string files only. Any string variable can be used but must be initialized before calling.

```
eg. A$ = "" : SYS 826 , 2 , A$
```

A$ only has to be initialized once.

Since the string is transfered from disk into a dummy input buffer ($7F00 to $8000 on 32K machines, $3F00 to 4000 on 16K), it is necessary to move the record into BASIC string storage. This can be accomplished by:

```
A$ = A$ + ""    or    Y$ = A$ + ""    or    A$(J) = A$ + ""
```

This routine has two advantages over INPUT#. It permits inputting strings up to 255 characters and it strips the Line Feeds placed on disk by PRINT#. Also it is much faster than using GET# in a loop.

Block Get can also be used for sequential file access to recover strings of length greater than 80. Even 255 character strings can be retrieved with Block Get. Essentially, Block Get is the same as INPUT# but with a 255 character input buffer....which brings us to point 2.

This 255 byte buffer is set up in the very top page of RAM; $7F00 to $7FFF on 32Ks or $3F00 to $3FFF for 16Ks. This space must be sealed off before INPUTing or INPUT#ing strings, defining strings as the result of a concatenation, LOADing DOS Support or anything else that resides in this memory space. Otherwise when Block Get is called, the data will be transfered from the disk into the buffer and clobber your DOS Support, strings or whatever happens to be there.

```
                    POKE 53 , PEEK (53) - 1 : CLR
```

Location 53 ($35) is the high order byte of the Top Of BASIC Pointer. Decrementing 53 by 1 brings the pointer down by 256 thus "sealing off" the top page of memory. PET will then ignore this memory as though it's not even there (try ?FRE(0)). You may want to use absolute values rather than PEEK (53) - 1 since each time this is executed, the pointer will decrement another 256 bytes.

```
                    32K :   POKE 53 , 127 : CLR
                    16K :   POKE 53 , 63  : CLR
```

The CLR command equates some other pointers to the new value of the Top of BASIC Pointer i.e. the Bottom of Strings Pointer and the Utility String Pointer. These could also be POKEd, but CLR does the job quite nicely.

If DOS Support is to be used with Block Get, this statement should be executed prior to RUNning DOS. However Block Get contains one gotcha that will leave DOS open for certain destruction.

When DOS Support is LOADed and RUN, it sets itself up just below the Top of BASIC Pointer (TBP). After executing the above command, the TBP will now be 256 bytes lower.... but that's ok since DOS can live anywhere. Once set up though, DOS lowers this pointer again to protect itself. But each time Block Get is called, the pointer is moved back to 256 bytes lower than the TBP at power up. Now DOS is sitting in memory that is available to BASIC. Re-RUN your program and whammo!...DOS Support gets clobbered by strings. Hit ">" and PET JSRs to where DOS used to be which is now ASCII characters....crash!

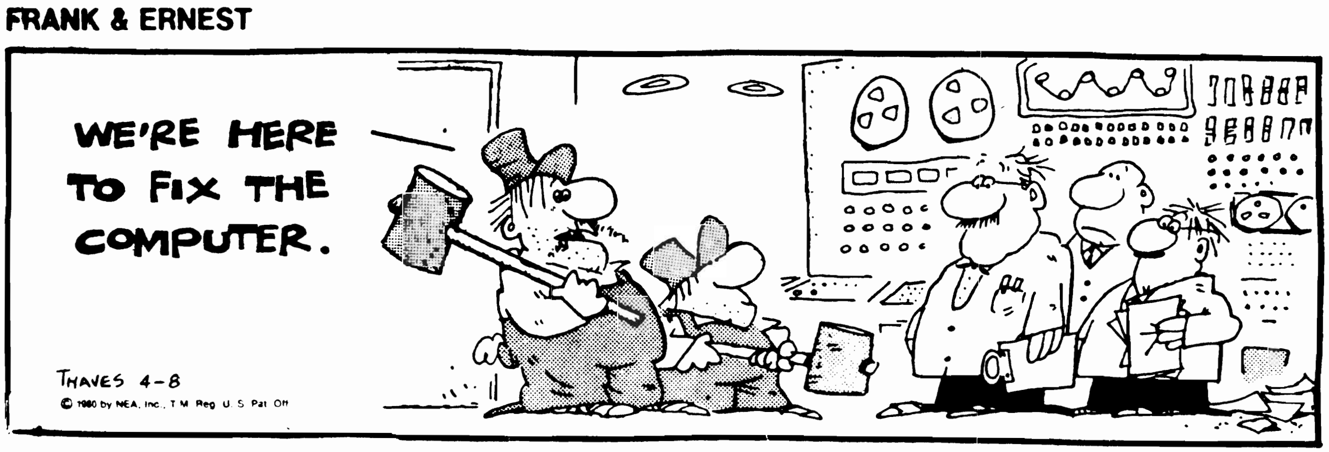
Fortunately this can be avoided. The first 8 bytes of Block Get sets the TBP every time it's called:

```
                          033A  LDA #$00
                          033C  STA  $34
                          033E  LDA #$7F   (3F for 16K)
                          0340  STA  $35
```

Therefore when using DOS Support and Block Get, SYS past these bytes with:

```
                          SYS 834 , LF# , A$

                Instead of: SYS 826 , LF# , A$
```


FRANK & ERNEST
WE'RE HERE TO FIX THE COMPUTER.
THAVES 4-8
© 1980 by NEA, Inc., T.M. Reg. U.S. Pat. Off.

commodore

comments and bulletins
concerning your
COMMODORE PET™

The Transactor

Vol. 2
BULLETIN # 8

PET™ is a registered Trademark of Commodore Inc.

Jan/Feb 1980

BITS AND PIECES

Re-DIMensioning Arrays

You all know what happens when you attempt to re-define an array that has already been defined. PET returns a ?REDIM'D ARRAY ERROR. But maybe you had a good reason to re-dimension. And now you must perform a CLR which clobbers all your variables, or else work around it. No longer! By manipulating some pointers down in zero page, arrays can be REDIM'D with no problem at all. Try the following example:

```
100 DIM A$ (1000)
110 GOSUB 2000
120 DIM A$ (2000)
130 GOSUB 2000
140 DIM A$ (126)
150 END

2000 POKE 46 , PEEK (44)
2010 POKE 47 , PEEK (45)
2020 Z9 = FRE (0)
2030 RETURN
```

The subroutine at 2000 "squeezes" the array out by making the End of Arrays Pointer equal to the Start of Arrays Pointer. PET now believes that there are no arrays of any name so DIMensioning is ok. The new array(s) is "built" in the same memory space.

Line 2000 forces a "garbage collection" so that any strings associated with Array A$ are thrown away. This wouldn't really be necessary with floating point or integer arrays since the values are stored in the array itself. With string arrays, only the string lengths and pointers to the strings are stored in the array. The strings lie elsewhere in RAM; in high memory if they were the result of a concatenation or INPUT from the keyboard, disk, etc. and directly in text if that's where they were defined (why store it twice?). This is also true for non-array type string variables. Of course strings residing in text are not thrown away by a garbage collection.

The Transactor is produced by WordPro III in conjunction with the NEC Spinwriter 5530

This trick can be played especially well when the sizes of your arrays are maintained in a disk file along with the file information.

Sometimes clearing all the arrays may not always be desirable. In that case, the order in which the arrays are defined becomes important. The 'permanent' arrays must be DIMensioned first, 'temporary' arrays last. However, if the value of the End of Arrays Pointer is stored immediately after defining the last 'permanent' array, the 'temporary' arrays can be squeezed out by POKing the End of Arrays Pointer with this value later on. For example:

```
   100 DIM A(1000) , B%(1500) , C$(1450)
   110 PL% = PEEK (46) : PH% = PEEK (47)

...1000 INPUT #8, I% , J% , K%
   1010 GOSUB 2100...

   2110 POKE 46, PL% : POKE 47, PH%
   2120 DIM X (I%) , Y% (J%) , Z$ (K%)
   2130 RETURN
```

The subroutine at 2100 would allow Arrays X, Y% and Z$ to be redimensioned any number of times without destroying Arrays A, B% and C$.

Dynamic LOADing

Steve Punter of Mississauga has a note for those performing LOADs from within programs. If strings are defined in text and are to be passed between programs they must be placed in high memory before the LOAD is executed.

As mentioned earlier, a string variable is set up with only the length and a pointer to the location of the first character of that string. When strings are defined in part of a line of BASIC, this pointer points right into that part of text. A dynamic LOAD replaces that text with new text and although the variable remains intact, the string itself is lost. Inotherwords, the pointer doesn't change but what lies in that location and the locations following is not what it used to be. In fact, it could be virtually anything; BASIC command or keyword tokens, line numbers or even another (or part of another) string.

About the easiest way to avoid this is to define strings in text as a concatenation. For example:

```
50 SP$ = "" + "          "
60 NO$ = "" + "0123456789"
```

When a concatenation of any kind is performed, PET automatically rebuilds the string up in high RAM area thus protecting them from dynamic LOADs.

Cursor Positioning

The following subroutines will remember the position of the cursor at a given time and restore the cursor to that position at a later time. This is often handy for displaying prompts or status messages in an area of the screen set aside for that purpose. Once the message is PRINTed, the cursor can be "brought back" to its former position to await user input, etc.

Another application would be to re-position the cursor for re-input of data that may have been unsuitable or unrelated to the previous prompt.

```
30049 REM + REMEMBER CURSOR POSITION +
30050 W% = PEEK (196)      :Old ROM 224
30060 X% = PEEK (197)      :Old ROM 225
30070 Y% = POS (0)
30080 Z% = PEEK (216)      :Old ROM 245
30090 RETURN

30149 REM + RESTORE CURSOR POSITION +
30150 POKE 196, W%
30160 POKE 197, X%
30170 POKE 216, Z%
30180 POKE 198, Y%         :Old ROM 226
30190 RETURN
```

BASIC and the Machine Language Monitor

Want to look at parts of your BASIC code with the monitor? Easy! Simply place a STOP command just before the code to be examined and execute it with a GOTO or a RUN followed by the appropriate line number. Now enter the monitor with SYS 4 and type:

```
.M 003A 003B
```

(Note: In the Machine Language Monitor, a space can be used as well as a comma for delimiting parameters.)

In memory locations 003A and 3B is a pointer which is mainly used by the CONTinue command. When a line containing STOP or END is executed, the hex address of that line is stored in 3A and 3B so that PET can pick up where it left off.

The address will appear low order first, high order second. Now a second ".M" command can be given using this address and some higher address to display the BASIC code in the general vicinity of the inserted STOP.

SAVing With The Monitor

Many BASIC programs are set up to access a machine language subroutine (Screen Print, Block GET, etc.)(also see F. VanDuinen's article PROGRAM PLUS). This code usually resides in the second cassette buffer. But the contents of the second cassette buffer are not recorded with a BASIC SAVE command. Including a loader routine as part of your program avoids this problem entirely as the machine code would be set up in the buffer on each RUN. However the loader will probably contain DATA statements which must be accounted for if other DATA statements are read and re-read later in the program (RESTORE brings the data pointer back to the first DATA element). Working around this can be cumbersome.

The solution is to ".S" the program with the Machine Language Monitor. Syntax for a Monitor SAVE is:

```
    .S "PROGRAM NAME",Dv#,start addrs,end addrs  (RETURN)
```

If the machine code is placed at the beginning of the 2nd cassette buffer, the start address will be 033A. But where does the program end ? This can be determined by first doing a memory display of the End of BASIC Pointer:

```
              .M 002A 002b   (RETURN)
```

The above might return something like:

```
              .002A 87 2C 16 2D 4F 2F 45 7A
```

The first two bytes indicate the end address (again, low order first, high order second) and in this case is 2C87. The Monitor SAVE command for this example would therefore be:

```
              .S "0:PROGRAM NAME",08,033A,2C87
```

The above is of course for disk users but 08 could also be 01 for cassette #1. Cassette #2 could <u>not</u> be used in this case since the recording process would wipe out the code in the 2nd cassette buffer.

Now when the program is LOADed, it will start loading with your machine code subroutine directly into the second cassette buffer.

Careful though! Any updates to this sort of program must be recorded using this same procedure. Additions or deletions will also cause the End of BASIC Pointer to change.

TRANSACTOR - A Philosophy

The January/February, 1980 issue marks the beginning of the third year of The Transactor and the beginning of an new decade. Starting with this issue you will be noticing changes to the Transactor format and content which we hope will benefit you - the dedicated PET user. It is safe to say that the dream of a computer in every home, which you the reader are pioneering, is well under way. This trend will no doubt accelerate geometrically in the early 1980's. The Transactor will evolve as necessary to keep pace (or slightly ahead of that pace).

Naturally the life blood of any non-profit publication such as The Transactor is your input. The potential of the PET system is so vast that no one or a small group of humans can hope to know all there is to know about the PET system. Each of us approach the PET with different needs, desires and applications. However in the process we discover answers or maybe as important raise questions which can be of incalculable use to the PET (and the greater 6502) community. This SYNERGISTIC process, where one plus one equals more than two, is the major function of The Transactor!

To make it easier for you to participate, and as an inducement, we will issue a free one year subscription (or extend your present subscription) for any original article submitted to and published in The Transactor. The publishing decision wil remain with COMMODORE so be patient if you do not see your article published at once. No doubt there will be a backlog of good articles.

We will experiment with annual BEST FEATURE ARTICLE and MOST CREATIVE APPLICATION awards. Beginning with Volume 2, bulletin #12 will contain a ballot. For best feature article, the winning author will receive a Commodore software product of their choice to a maximum value of $125.00; for most creative application, a Commodore calculator (max. $50.00). If reader response warrants it, we will issue runner-up awards also.

We will continue to welcome your many letters and telephone calls. We will try to answer all, either individually (if we can) or through calls for help in the The Transactor . If your question proves particullarly widespread we will publish a general answer in The Transactor.

With this and future issues we will include an index. For this issue we include an outline of articles we would like to cover in future issues. We welcome your comments particularly those articles which are of most interest to you. Of course such an objective will require considerable dedication from our readership. As readership increases (it presently numbers 800+) we may be able to provide a modest honarium.

If all the above sounds like an attempt to create another slick, glossy magazine please be assured this is not the case. Only by maintaining our present non-commercial, non-profit status will we be able to continue to provide and improve the support for the PET system.

Karl J. Hildon
Editor

POP a RETURN and Your Stack Will Feel Better

Ever wanted to 'POP' out of a subroutine ? The POP function, available in some forms of BASIC, allows you to jump out of a subroutine using GOTO without leaving the RETURN information on the stack. But what if this information is left on the stack ? Try the following "bad" example:

```
100 GOSUB 200
110 END
200 PRINT"SUBROUTINE ENTRY"
210 GOTO 100
220 PRINT"SUBROUTINE EXIT" : RETURN
```

Of course line 220 will never execute but is the proper way to terminate a subroutine. Instead, execution is re-directed back to line 100 where another GOSUB is performed and more RETURN information is pushed onto the stack. Soon the stack fills to capacity and PET displays the ?OUT OF MEMORY ERROR IN 200.

Now change line 210 to:

```
210 SYS 50583 : GOTO 100
```

With this modification the RETURN information will be artificially POPed off the stack before jumping out of the subroutine. (SYS 50568 for Old ROM)

This POP resets the entire stack. That is all RETURNs are POPed (eg. subroutines called by subroutines). A single POP can be accomplished by doing a SYS to 7 PLA's followed by an RTS.

Jumping out of subroutines is bad programming practice and should be avoided at all cost. But these simulated POPs have their applications. Consider an INPUT subroutine that handles an escape key (eg. the "@" symbol). This escape key takes the program back to a "warm start", for instance the Main Menu. You could test for the "@" and RUN if true, but RUN also CLRs all variables. Another method would be to RETURN from the INPUT subroutine upon detecting the "@" but a second "@" key test would be necessary upon RETURNing. This second test would also have to be repeated for every GOSUB to the INPUT subroutine which might consume considerable memory depending on the number of times the INPUT subroutine is used. The third method, and probrably the best for handling an escape key, is to use POP:

```
20000 +++ INPUT SUBROUTINE +++
20010 GET A$ : IF A$ = "" THEN 20010
20020 IF A$ = "@" THEN SYS 50583 : GOTO (Menu)
20030 See Transactor #6, Bullet Proof INPUT
```

The POP SYS for BASIC 2.0 also has an equivalent BASIC 4.0 entry point:

```
BASIC 2.0:   SYS 50583
BASIC 4.0:   SYS 46610
```

Disk Merge

The following program uses disk in much the same fashion as the existing tape merge to merge one program with another in new ROM PETs.

First LOAD the sub-program or subroutine that you wish to merge with your main program. Make sure that this code doesn't use line 0 as the merge routine makes use of this line. Now type directly on the screen:

```
  OPEN 8,8,8, " 0 : MERGE FILE NAME , S , W " : CMD 8 : LIST
```

Of course 'MERGE FILE NAME' can be any filename and any part of the program can be 'LISTed' by following the LIST command with parameters.

Now type:

```
  PRINT #8 : CLOSE 8
```

The merge file is now complete and can be merged with any program at any time. LOAD the main program into RAM and enter the following line of BASIC without the spaces. Abbreviations must be used so that Disk Merge will fit on one line.

```
  0 INPUT#8,A$ : PRINT "cs"A$ : PRINT "POKE 174,1 : POKE
    593,8 : GOTO 0 " : POKE 158,3 : POKE 623,19 : POKE
    624,13 : POKE 625,13 : END
```

With Abbreviations:

```
  0 iN8,A$ : ? "cs"A$ : ? "pO 174,1 : pO593,8 : gO 0" : pO
    158,3 : pO 623,19 : pO 624,13 : pO 625,13 : eN
```

Now type:

```
  OPEN 8,8,8,"0:MERGE FILE NAME,S,R" : GOTO 0  (Return)
```

and watch it go. One glitch...any lines in the merge file that span greater than two lines (>80 characters) such as those originally entered using abreviations, will cause the process to halt. Since Disk Merge makes use of the PET screen editor, these lines cannot be properly entered anyways as the BASIC input buffer is only 80 bytes long (see upgrade ROM memory map locations 512 to 592 decimal). If this happens you can fix up the line with the appropriate abreviations, enter it with a 'RETURN', and continue the merge by executing the command line underneath (Po 174,1 : Po 593,8 : Go 0).

As with tape merge (Transactor #2, Vol 2), a ?SYNTAX ERROR or ?OUT OF DATA ERROR will appear when the merge is complete.

Supermon 1.0

Supermon is a machine language program which seals itself off in RAM and links itself to the built-in ROM Monitor. Once initialized, Supermon provides extended machine language monitor (M.L.M.) commands in much the same way that the Programmers Toolkit adds extra direct commands to BASIC. It is the ideal machine language programmers tool.

SUPERMON 1.0

COMMANDS - USER INPUT IN **REVERSE**

GO RUN

```
.G
```

GO TO THE ADDRESS IN THE PC REGISTER DISPLAY AND BEGIN RUN CODE. ALL THE REGISTERS WILL BE REPLACED WITH THE DISPLAYED VALUES.

```
.G 1000
```

GO TO ADDRESS 1000 HEX AND BEGIN RUNNING CODE.

LOAD FROM TAPE

```
.L
```

LOAD ANY PROGRAM FROM CASSETTE #1.

```
.L "RAM TEST"
```

LOAD FROM CASSETTE #1 THE PROGRAM NAMED **RAM TEST**.

```
.L "RAM TEST",02
```

LOAD FROM CASSETTE #2 THE PROGRAM NAMED **RAM TEST**.

COMMANDS - USER INPUT IN REVERSE

MEMORY DISPLAY

```
.M 0000 0080

.:  0000 00 01 02 03 04 05 06 07
.:  0008 08 09 0A 0B 0C 0D 0E 0F
```

DISPLAY MEMORY FROM 0000 HEX TO 0080 HEX. THE BYTES FOLLOWING THE ADDRESS MAY BE MODIFIED BY EDITING AND THEN TYPING A RETURN.

SAVE TO TAPE

```
.S "PROGRAM NAME",01,0800,0C80
```

SAVE TO CASSETTE #1 MEMORY FROM 0800 HEX UP TO BUT NOT INCLUDING 0C80 HEX AND NAME IT PROGRAM NAME.

HUNT MEMORY

```
.H C000 D000 'READ
```

HUNT THRU MEMORY FROM C000 HEX TO D000 HEX FOR THE ASCII STRING READ AND PRINT THE ADDRESS WHERE IT IS FOUND. A MAXIMUM OF 32 CHARACTERS MAY BE USED.

```
.H C000 D000 20 D2 FF
```

HUNT MEMORY FROM C000 HEX TO D000 HEX FOR THE SEQUENCE OF BYTES 20 D2 FF AND PRINT THE ADDRESS. A MAXIMUM OF 32 BYTES MAY BE USED.

REGISTER DISPLAY

```
.R

     PC   IRQ  SR AC XR YR SP
.;  0000 E62E 01 02 03 04 05
```

DISPLAYS THE REGISTER VALUES SAVED WHEN SUPERMON WAS ENTERED. THE VALUES MAY BE CHANGED WITH THE EDIT FOLLOWED BY A RETURN.

USE THIS INSTRUCTION TO SET UP THE PC VALUE BEFORE SINGLE STEPPING WITH .I

EXIT TO BASIC

```
.X
```

RETURN TO BASIC READY MODE. THE STACK VALUE SAVED WHEN ENTERED WILL BE RESTORED. CARE SHOULD BE TAKEN THAT THIS VALUE IS THE SAME AS WHEN THE MONITOR WAS ENTERED. A CLR IN BASIC WILL FIX ANY STACK PROBLEMS.

FILL MEMORY

```
.F 1000 1100 FF
```

FILLS THE MEMORY FROM 1000 HEX TO 1100 HEX WITH THE BYTE FF HEX.

TRANSFER MEMORY

```
.T 1000 1100 5000
```

TRANSFER MEMORY IN THE RANGE 1000 HEX TO 1100 HEX AND START STORING IT AT ADDRESS 5000 HEX.

SIMPLE ASSEMBLER

```
.A 2000 LDA #$12
.A 2002 STA $8000,X
.A 2005 (RETURN)
.
```

IN THE ABOVE EXAMPLE THE USER STARTED ASSEMBLY AT 1000 HEX. THE FIRST INSTRUCTION WAS LOAD A REGISTER WITH IMMEDIATE 12 HEX. IN THE SECOND LINE THE USER DID NOT NEED TO TYPE THE A AND ADDRESS. THE SIMPLE ASSEMBLER PROMPTS WITH THE NEXT ADDRESS. TO EXIT THE ASSEMBLER TYPE A RETURN AFTER THE THE ADDRESS PROMPT. SYNTAX IS THE SAME AS THE DISASSEMBLER OUTPUT.

SINGLE STEP

```
.I
```

ALLOWS A MACHINE LANGUAGE PROGRAM TO BE RUN STEP BY STEP.

CALL REGISTER DISPLAY WITH .R AND SET THE PC ADDRESS TO THE DESIRED FIRST INSTRUCTION FOR SINGLE STEPPING. THE .I WILL CAUSE A SINGLE STEP TO EXECUTE AND WILL DISASSEMBLE THE NEXT.

CONTROLS:
◀ FOR SINGLE STEP;
RVS FOR SLOW STEP;
SPACE FOR FAST STEPPING;
STOP TO RETURN TO MONITOR.

CALCULATE BRANCH

```
.C 1000 1010 0E
```

THE EXAMPLE CALCULATES THE SECOND BYTE OF A BRANCH INSTRUCTION. THE BRANCH OP-CODE IS AT 1000 HEX AND THE TARGET ADDRESS IS 1010 HEX. SUPERMON RESPONDED WITH THE 0E HEX OFFSET.

DISASSEMBLER

```
.D 2000

(SCREEN CLEARS)
.,  2000 A9 12        LDA #$12
.,  2002 9D 00 80     STA $8000,X
.,  2005 AA           TAX
.,  2006 AA           TAX
(FULL PAGE OF INSTRUCTIONS)
```

DISASSEMBLES 22 INSTRUCTIONS STARTING AT 1000 HEX. THE THREE BYTES FOLLOWING THE ADDRESS MAY BE MODIFIED. USE THE CRSR KEYS TO MOVE TO AND MODIFY THE BYTES. HIT RETURN AND THE BYTES IN MEMORY WILL BE CHANGED. SUPERMON WILL THEN DISASSEMBLE THAT PAGE AGAIN.

SUPERMON 1.0

SUMMARY

COMMODORE MONITOR INSTRUCTIONS:

G GO RUN
L LOAD FROM TAPE
M MEMORY DISPLAY
R REGISTER DISPLAY
S SAVE TO TAPE
X EXIT TO BASIC

SUPERMON ADDITIONAL INSTRUCTIONS:

A SIMPLE ASSEMBLER
C CALCULATE BRANCH
D DISASSEMBLER
F FILL MEMORY
H HUNT MEMORY
I SINGLE STEP
T TRANSFER MEMORY

Supermon 1.0 : Set up

The procedure to follow is about the simplest paper to PET transcription for obtaining a fully operational Supermon. The time spent here will be saved ten fold by dedicated machine code programmers and for those just getting started in machine language, Supermon is the perfect launch to more sophisticated assemblers and programs.

Step 1.

The two programs below are, respectively, the loader/relocator and checksum programs for the Supermon machine code to be entered later. Enter them into PET, double check, and SAVE seperately. Tape users should use seperate cassettes. Note: the two letter mnemonics within square brackets designate PET cursor control characters and should be entered as such.

CAUTION: These programs should be entered <u>exactly</u> as they appear. Spaces can be omitted but anything that will cause the programs to be larger than shown (i.e. added commands, cursor control, spaces or characters, indenting, REMarks, etc.) must be avoided. Immediately before SAVing, check that FRE(0) is less than or equal to 31052 (14668 for 16k machines and 6476 for 8k). If not, LIST and edit out any text that doesn't belong. Otherwise I predict extreme exasperation in your future.

```
100 PRINT"[CS DN DN RV] SUPERMON! "
110 PRINT"[DN]     DISSASSEMBLER [RV]D[RO] BY WOZNIAK/BAUM
120 PRINT"       SINGLE STEP [RV]I[RO] BY JIM RUSSO
130 PRINT"MOST OTHER STUFF [RV],CHAFT[RO] BY BILL SEILER
140 PRINT"[DN]BLENDED & PUT IN RELOCATABLE FORM"
150 PRINT"    BY JIM BUTTERFIELD"
155 POKE42,182:POKE43,6:CLR
160 L=PEEK(52)+PEEK(53)*256
170 N=L-1466:P=3391:FORJ=L-1TONSTEP-1
180 X=PEEK(P):IFX>0GOTO190
185 P=P-2:X=PEEK(P+1)+PEEK(P)*256:IFX=0GOTO190
186 X=X+L-65536:X%=X/256:X=X-X%*256:POKEJ,X%:J=J-1
190 POKEJ,X:P=P-1:PRINT"[HM]";X;"[CL]  ":NEXTJ
200 X%=N/256:Y=N-X%*256:POKE52,Y:POKE53,X%:POKE48,Y:POKE49,X%
210 PRINT"[CS DN]LINK TO MONITOR -- SYS";N
220 PRINT:PRINT"SAVE WITH MLM:"
230 PRINT".S ";CHR$(34);"SUPERMON";CHR$(34);",01";:X=N/4096:GOSUB250
240 X=L/4096:GOSUB250:END
250 PRINT",";:FORJ=1TO4:X%=X:X=(X-X%)*16:IFX%>9THENX%=X%+7
260 PRINTCHR$(X%+48);:NEXTJ:RETURN
```

```
100 PRINT"SUPERMON CHECKSUM":CH=0
110 FOR J = 1718 TO 3397 STEP 40
120 FOR I = 0 TO 39
130 CH = CH + PEEK(J + I)
140 NEXT I
150 READ CK : IF CK <> CH THEN 180
160 CH = 0 : NEXT J
170 PRINT" NO ERRORS !!" : END
180 PRINT" DATA ENTRY ERROR IN BLOCK ";(J - 1718 + I)/40
190 PRINT" ENTER M.L.M. WITH SYS 4 AND VERIFY":END
200 DATA 5428, 5429, 5348, 5125, 6141, 5576, 5622, 5845, 4883, 5703
210 DATA 4966, 5273, 5006, 5594, 5091, 5266, 5066, 4152, 4942, 4180
220 DATA 5697, 4801, 5690, 5363, 3398, 4556, 4639, 5236, 4843, 5232
230 DATA 5359, 4924, 5653, 5717, 2711, 2631, 1965, 2874, 3707, 4148
240 DATA 2832, 5392.
```

Step 2.

On the pages to follow is the machine code data for Supermon 1.0. This data will be read by the loader/relocater program and packed into the top of memory, wherever that happens to be on your machine*. Note: this is not the actual machine language for Supermon but rather the machine code data in relocatable form.

To enter this data, first (pour yourself a fresh tea or coffee or open another pint and) enter:

```
SYS 64715
```

This is power-on reset or the equivalent of power down-power up. Now enter the machine language monitor with:

```
SYS 4
```

To make it easier, the code has been sectioned off into groups of ten lines, each displaying 8 bytes in hex. The first section (see next page) starts at $06B6 and continues down to $06FE+8 or $0705. However, the monitor will complete the line regardless of where in the line the contents of the last address specified will be printed. Therefore, enter the monitor command "M", for memory display, followed by these two addresses:

```
M 06B6,06FE
```

On hitting 'RETURN', the screen displays 10 hex addresses and the 8 hex bytes following that address inclusively. Since what is displayed is "empty space", all bytes should be the same. In most cases they will be hex "AA's".

* Supermon relocates according to PET's Top of Memory Pointer. Therefore any programs already residing in the very top of user RAM (e.g. DOS Support, TRACE, etc.) will not be touched by Supermon.

Now move the cursor up to the first AA (beside .: 06B6) and, using the screen editor just as in BASIC, begin entering the data as shown in the first section. Use spaces between each byte and hit 'RETURN' at the end of each line. This enters all 8 bytes of the line simultaneously into their respective addresses in RAM. Don't worry too much about mistakes...the checksum program will help you find them later on.

Upon completing a section entry, execute another "M"emory display using the first and last addresses shown for the next section (as above). Continue entering bytes as before until all sections have been completed. (The 5 "AA's" at the end need not be re-entered but should be there for the checksum to work.)

Once finished, SAVE it! Type:

```
S "Ø:MON DATA Ø",08,06B6,0D45
```

This is of course for disk users; tape users can omit the drive number in the file name and substitute 08 with the appropriate cassette number.

Step 3.

Exit the monitor (X and 'RETURN') but do not reset PET. Instead, LOAD the checksum program (recorded earlier) and RUN. This checks a block at a time by summing consecutive bytes and comparing against a checksum. A block is half of a section so if a " DATA ENTRY ERROR IN BLOCK x " occurs, count two blocks for each section. An odd number will indicate an error in the first half of the section and of vice versa. Fix any and all errors using the monitor, each time eXiting and re-RUNning the checksum program until a " NO ERRORS !! " is returned. If there were no errors on the first RUN, there's no need to re-SAVE. Otherwise do a second SAVE using the same monitor command as above but of course with a different file name.

Step 4.

Once again, eXit the monitor but do not reset. LOAD the relocator program and RUN. Assuming all goes well, the program will end with instructions for initializing Supermon and SAVing just the relocated machine language. However, SAVE the relocator and the byte data together for later use (in case Supermon is to be relocated into a different size machine or along with other relocatable utilities e.g. TRACE :see Compute Issue #1). Enter the monitor with SYS4 and Type:

```
S "Ø:SUPERMON.REL",08,0400,0D46
```

...for SUPERMON Point RELocatable.

```
.:   06B6 AD FF FE 00 85 34 AD FF
.:   06BE FF 00 85 35 AD FF FC 00
.:   06C6 8D FA 03 AD FF FD 00 8D
.:   06CE FB 03 00 00 00 A2 08 DD
.:   06D6 FF DE 00 D0 0E 86 B4 8A
.:   06DE 0A AA BD FF E9 00 48 BD
.:   06E6 FF AD FF FE 00 85 34 AD
.:   06EE FF FF 00 85 35 AD FF FC
.:   06F6 00 8D FA 03 AD FF FD 00
.:   06FE 8D FB 03 00 00 00 A2 08
.:   0706 DD FF DE 00 D0 0E 86 B4
.:   070E 8A 0A AA BD FF E9 00 48
.:   0716 BD FF E8 00 48 60 CA 10
.:   071E EA 4C F7 E7 A2 02 2C A2
.:   0726 00 00 00 B4 FB D0 08 B4
.:   072E FC D0 02 E6 DE D6 FC D6
.:   0736 FB 60 20 EB E7 C9 20 F0
.:   073E F9 60 A9 00 00 00 8D 00
.:   0746 00 00 01 20 FA 8C 00 20
.:   074E BE E7 20 AA E7 90 09 60
.:   0756 20 EB E7 20 A7 E7 B0 DE
.:   075E 4C F7 E7 20 CD FD CA D0
.:   0766 FA 60 E6 FD D0 02 E6 FE
.:   076E 60 A2 02 B5 FA 48 BD 0A
.:   0776 02 95 FA 68 9D 0A 02 CA
.:   077E D0 F1 60 AD 0B 02 AC 0C
.:   0786 02 4C FA DD 00 A5 FD A4
.:   078E FE 38 E5 FB 85 CF 98 E5
.:   0796 FC A8 05 CF 60 20 FA 94
.:   079E 00 20 97 E7 20 FA A5 00
.:   07A6 20 FA BE 00 20 FA A5 00
.:   07AE 20 FA D9 00 20 97 E7 90
.:   07B6 15 A6 DE D0 64 20 FA D0
.:   07BE 00 90 5F A1 FB 81 FD 20
.:   07C6 FA B7 00 20 D5 FD D0 EB
.:   07CE 20 FA D0 00 18 A5 CF 65
.:   07D6 FD 85 FD 98 65 FE 85 FE
.:   07DE 20 FA BE 00 A6 DE D0 3D
.:   07E6 A1 FB 81 FD 20 FA D0 00
.:   07EE B0 34 20 FA 78 00 20 FA
.:   07F6 7B 00 4C FB 27 00 20 FA
.:   07FE 94 00 20 97 E7 20 FA A5
.:   0806 00 20 97 E7 20 EB E7 20
.:   080E B6 E7 90 14 85 B5 A6 DE
.:   0816 D0 11 20 FA D9 00 90 0C
.:   081E A5 B5 81 FB 20 D5 FD D0
.:   0826 EE 4C F7 E7 4C 56 FD 20
.:   082E FA 94 00 20 97 E7 20 FA
.:   0836 A5 00 20 97 E7 20 EB E7
.:   083E A2 00 00 00 20 EB E7 C9
.:   0846 27 D0 14 20 EB E7 9D 10
.:   084E 02 E8 20 CF FF C9 0D F0
.:   0856 22 E0 20 D0 F1 F0 1C 8E
.:   085E 00 00 00 01 20 BE E7 90
.:   0866 C6 9D 10 02 E8 20 CF FF
.:   086E C9 0D F0 09 20 B6 E7 90
.:   0876 B6 E0 20 D0 EC 86 B4 20
```

```
.:   087E D0 FD A2 00 00 00 A0 00
.:   0886 00 00 B1 FB DD 10 02 D0
.:   088E 0C C8 E8 E4 B4 D0 F3 20
.:   0896 6A E7 20 CD FD 20 D5 FD
.:   089E A6 DE D0 92 20 FA D9 00
.:   08A6 B0 DD 4C 56 FD 20 FA 94
.:   08AE 00 8D 0D 02 A5 FC 8D 0E
.:   08B6 02 A9 04 A2 00 00 00 85
.:   08BE B8 86 B9 A9 93 20 D2 FF
.:   08C6 A9 16 85 B5 20 FC 10 00
.:   08CE 20 FC 6D 00 85 FB 84 FC
.:   08D6 C6 B5 D0 F2 A9 91 20 D2
.:   08DE FF 4C 56 FD A0 2C 20 15
.:   08E6 FE 20 6A E7 20 CD FD A2
.:   08EE 00 00 00 A1 FB 20 FC 7C
.:   08F6 00 48 20 FC C2 00 68 20
.:   08FE FC D8 00 A2 06 E0 03 D0
.:   0906 12 A4 B6 F0 0E A5 FF C9
.:   090E E8 B1 FB B0 1C 20 FC 65
.:   0916 00 88 D0 F2 06 FF 90 0E
.:   091E BD FF 4A 00 20 FD 4D 00
.:   0926 BD FF 50 00 F0 03 20 FD
.:   092E 4D 00 CA D0 D5 60 20 FC
.:   0936 70 00 AA E8 D0 01 C8 98
.:   093E 20 FC 65 00 8A 86 B4 20
.:   0946 75 E7 A6 B4 60 A5 B6 38
.:   094E A4 FC AA 10 01 88 65 FB
.:   0956 90 01 C8 60 A8 4A 90 0B
.:   095E 4A B0 17 C9 22 F0 13 29
.:   0966 07 09 80 4A AA BD FE F9
.:   096E 00 B0 04 4A 4A 4A 4A 29
.:   0976 0F D0 04 A0 80 A9 00 00
.:   097E 00 AA BD FF 3D 00 85 FF
.:   0986 29 03 85 B6 98 29 8F AA
.:   098E 98 A0 03 E0 8A F0 0B 4A
.:   0996 90 08 4A 4A 09 20 88 D0
.:   099E FA C8 88 D0 F2 60 B1 FB
.:   09A6 20 FC 65 00 A2 01 20 FA
.:   09AE B0 00 C4 B6 C8 90 F1 A2
.:   09B6 03 C4 B8 90 F2 60 A8 B9
.:   09BE FF 57 00 8D 0B 02 B9 FF
.:   09C6 97 00 8D 0C 02 A9 00 00
.:   09CE 00 A0 05 0E 0C 02 2E 0B
.:   09D6 02 2A 88 D0 F6 69 3F 20
.:   09DE D2 FF CA D0 EA 4C CD FD
.:   09E6 20 FA 94 00 20 D5 FD 20
.:   09EE D5 FD 20 97 E7 20 FA A5
.:   09F6 00 20 97 E7 20 CA FD 20
.:   09FE FA D9 00 90 09 98 D0 13
.:   0A06 A5 CF 30 0F 10 07 C8 D0
.:   0A0E 0A A5 CF 10 06 20 75 E7
.:   0A16 4C 56 FD 4C F7 E7 20 FA
.:   0A1E 94 00 A9 03 85 B5 20 EB
.:   0A26 E7 20 A7 FD D0 F8 AD 0D
.:   0A2E 02 85 FB AD 0E 02 85 FC
.:   0A36 4C FB F1 00 C5 B9 F0 03
.:   0A3E 20 D2 FF 60 A9 03 A2 24
.:  0A46 85 B8 86 B9 20 D0 FD 78
.:  0A4E AD FF FA 00 85 90 AD FF
.:  0A56 FB 00 85 91 A9 A0 8D 4E
.:  0A5E E8 CE 13 E8 A9 2E 8D 48
.:  0A66 E8 A9 00 00 00 8D 49 E8
.:  0A6E AE 06 02 9A 4C F1 FE 20
.:  0A76 7B FC 68 8D 05 02 68 8D
.:  0A7E 04 02 68 8D 03 02 68 8D
.:  0A86 02 02 68 8D 01 02 68 8D
.:  0A8E 00 00 00 02 BA 8E 06 02
.:  0A96 58 20 D0 FD 20 BF FD 85
.:  0A9E B5 A0 00 00 00 20 9A FD
.:  0AA6 20 CD FD AD 00 00 00 02
.:  0AAE 85 FC AD 01 02 85 FB 20
.:  0AB6 6A E7 20 FC 18 00 20 01
.:  0ABE F3 C9 F7 F0 F9 20 01 F3
.:  0AC6 D0 03 4C 56 FD C9 FF F0
.:  0ACE F4 4C FD 60 00 00 00 00
.:  0AD6 20 FA 94 00 20 97 E7 8E
.:  0ADE 11 02 A2 03 20 FA 8C 00
.:  0AE6 48 CA D0 F9 A2 03 68 38
.:  0AEE E9 3F A0 05 4A 6E 11 02
.:  0AF6 6E 10 02 88 D0 F6 CA D0
.:  0AFE ED A2 02 20 CF FF C9 0D
.:  0B06 F0 1E C9 20 F0 F5 20 FE
.:  0B0E F0 00 B0 0F 20 CB E7 A4
.:  0B16 FB 84 FC 85 FB A9 30 9D
.:  0B1E 10 02 E8 9D 10 02 E8 D0
.:  0B26 DB 8E 0B 02 A2 00 00 00
.:  0B2E 86 DE A2 00 00 00 86 B5
.:  0B36 A5 DE 20 FC 7C 00 A6 FF
.:  0B3E 8E 0C 02 AA BD FF 97 00
.:  0B46 20 FE D5 00 BD FF 57 00
.:  0B4E 20 FE D5 00 A2 06 E0 03
.:  0B56 D0 12 A4 B6 F0 0E A5 FF
.:  0B5E C9 E8 A9 30 B0 1D 20 FE
.:  0B66 D2 00 88 D0 F2 06 FF 90
.:  0B6E 0E BD FF 4A 00 20 FE D5
.:  0B76 00 BD FF 50 00 F0 03 20
.:  0B7E FE D5 00 CA D0 D5 F0 06
.:  0B86 20 FE D2 00 20 FE D2 00
.:  0B8E AD 0B 02 C5 B5 D0 59 20
.:  0B96 97 E7 A4 B6 F0 2B AD 0C
.:  0B9E 02 C9 9D D0 1C 20 FA D9
.:  0BA6 00 90 09 98 D0 4A A6 CF
.:  0BAE 30 46 10 07 C8 D0 41 A6
.:  0BB6 CF 10 3D CA CA 8A A4 B6
.:  0BBE D0 03 B9 FC 00 00 00 91
.:  0BC6 FB 88 D0 F8 A5 DE 91 FB
.:  0BCE 20 FC 6D 00 85 FB 84 FC
.:  0BD6 A0 41 20 15 FE 20 6A E7
.:  0BDE 20 CD FD 4C FD DE 00 20
.:  0BE6 FE D5 00 86 B4 A6 B5 DD
.:  0BEE 10 02 F0 0C 68 68 E6 DE
.:  0BF6 F0 03 4C FE 30 00 4C F7
.:  0BFE E7 E8 86 B5 A6 B4 60 C9
```

```
.:  0C06 30 90 03 C9 47 60 38 60
.:  0C0E 40 02 45 03 D0 08 40 09
.:  0C16 30 22 45 33 D0 08 40 09
.:  0C1E 40 02 45 33 D0 08 40 09
.:  0C26 40 02 45 B3 D0 08 40 09
.:  0C2E 00 00 00 22 44 33 D0 8C
.:  0C36 44 00 00 00 11 22 44 33
.:  0C3E D0 8C 44 9A 10 22 44 33
.:  0C46 D0 08 40 09 10 22 44 33
.:  0C4E D0 08 40 09 62 13 78 A9
.:  0C56 00 00 00 21 81 82 00 00
.:  0C5E 00 00 00 00 59 4D 91 92
.:  0C66 86 4A 85 9D 2C 29 2C 23
.:  0C6E 28 24 59 00 00 00 58 24
.:  0C76 24 00 00 00 1C 8A 1C 23
.:  0C7E 5D 8B 1B A1 9D 8A 1D 23
.:  0C86 9D 8B 1D A1 00 00 00 29
.:  0C8E 19 AE 69 A8 19 23 24 53
.:  0C96 1B 23 24 53 19 A1 00 00
.:  0C9E 00 1A 5B 5B A5 69 24 24
.:  0CA6 AE AE A8 AD 29 00 00 00
.:  0CAE 7C 00 00 00 15 9C 6D 9C
.:  0CB6 A5 69 29 53 84 13 34 11
.:  0CBE A5 69 23 A0 D8 62 5A 48
.:  0CC6 26 62 94 88 54 44 C8 54
.:  0CCE 68 44 E8 94 00 00 00 B4
.:  0CD6 08 84 74 B4 28 6E 74 F4
.:  0CDE CC 4A 72 F2 A4 8A 00 00
.:  0CE6 00 AA A2 A2 74 74 74 72
.:  0CEE 44 68 B2 32 B2 00 00 00
.:  0CF6 22 00 00 00 1A 1A 26 26
.:  0CFE 72 72 88 C8 C4 CA 26 48
.:  0D06 44 44 A2 C8 04 22 10 20
.:  0D0E 2D 2F 33 54 46 48 44 43
.:  0D16 2C 41 49 4E 00 00 00 FA
.:  0D1E E8 00 FB 3C 00 FB 6A 00
.:  0D25 FB DD 00 FC FD 00 FD 30
.:  0D2E 00 FD DA 00 FD 54 00 55
.:  0D36 FD FD 84 00 FA 5D 00 FA
.:  0D3E 46 00 AA AA AA AA AA AA
```

RS-232C: AN OVERVIEW

W.T. Garbutt
Mississauga
Ontario, L5L 1K3

Sooner or later the PET owner requires greater memory storage or printed copy. For the former he can purchase a CBM disc, connect the cable, sit back and compute; for the later he can purchase a CBM printer. If the user needs a more esoteric peripheral say photometric analysis, current measurement etc. they will likely use the IEEE bus, so thoughtfully provided by the folks at Commodore. In a previous issue of The TRANSACTOR, Jim Butterfield talked about the IEEE buss. At the end of this article we provide a brief bibliographpy for further exploration.

The IEEE port is not the only means a PET owner has to access the real world. As a matter of fact the most common peripheral interfacing technique in use is not the IEEE port. It is of course RS-232C.

A brief digression to review the differences between PARALLEL and SERIAL data transfer will prove useful.

As we may recall PARALLEL data transfer involves sending out eight bits of data simultaneously over eight hard wires to define a byte or character. In addition a number of additional wires are needed to provide processor control and translation. While this method has the advantage of speed (a byte is available at one time) it requires complex circuitry to interface to analog terminals as well as multi-conductor cable. The IEEE interface is a special example of the PARALLEL method.

SERIAL data transmission, on the other hand is the method of sending data one bit at a time over a single wire. While inherently slower than the PARALLEL method it is ideally suited to the slow, single line analog interconnections such as phone lines, cassette tapes, radio or human operated printers or teletypes.

Essentially RS-232C is the title for a standard formulated by the Electronic Industries Association (EIA). As a standard it decribes a set of parameters that must exist to provide the housekeeping necessary to interface a peripheral and transmit data to a computer.

During the early 1960's the EIA formulated a set of standards to allow for an orderly interconnection and communication of peripherals to the then newly developing mini-computers. Prior to EIA's RS-232C standard what communication did take place was, in the vast majority of cases, handled by the 60 or 20 ma current loop teletypes.

Let's take a close look at the standard. The EIA Standard RS-232C is entitled "Interface Between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange". For the compulsive reader the standard comprises a 29 page document covering "Electrical Signal Characteristics", "Interface Circuits and Mechanical Interface", and "Standard Interface for Selected Communication System Configuration".

The standard has gained widespread use not only in the original area of intent, communication between terminal and modems, but also for the interconnection of computer peripherals such as printers, plotters, etc.

Electrical Signal Characteristics

The RS-232C standard as we indicated previously is based on SERIAL data transmission eg. a bit at a time over a single wire (as opposed to PARALLEL, in which different bits travel over seperate wires at the same time). Electrically, a logic zero is represented by a voltage between +5 and +15 V; a logic one by a voltage between -5 and -15 V (see FIGURE 1). The RS-232C standard also prescribes electrical impedence; drive capabilities, and signal voltage rate-of-change limits etc.

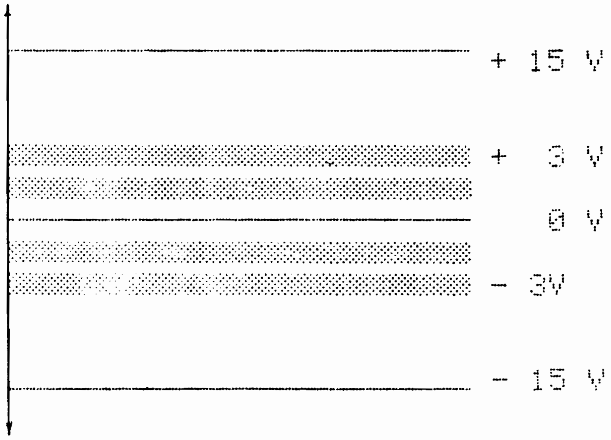


FIGURE 1

BIT REPRESENTATION

The transmission can be synchronous or asynchronous. Synchronous transmission requires that a clock signal be present (usually transmitted on a seperate line) to mark the start of each bit of information. Optionally, special data patterns are used to define the start of a message. Data must of course follow uninterrupted in sychronization with the clock signal. With asynchronous transmission a clock signal is not transmitted with data. Instead the synchronizing information is incorporated into the data itself as a single logic zero at the start of a character and a logic one at the end of the character (see FIGURE 2). The receiver contains an internal clock that examines the data triggered by the logic one and zero bit and locates the character bit.

The advantages of using asynchronous transmission are clearly obvious;

1.The transmission need not be continuous (desirable when entering data to a terminal manually)

2.Less complex (no clock) and hence less prone to error.

3.Capable of moderately high transmission speeds.

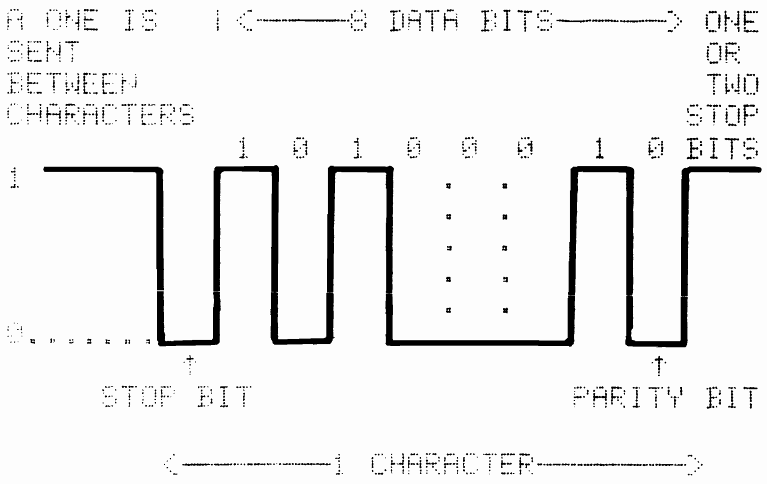


FIGURE 2

ASYNCHRONOUS ASCII
CHARACTER REPRESENTATION

Interchange Circuits

The signal interchange circuits defined by RS-232C fall into four groups: ground, data, control, and timing. We have already mentioned timing (e.g. synchronous and asynchronous transmission). Grounding is, of course, obvious. Let's examine data and control.

Data

Within an RS-232C interface are two seperate bi-directional data channels. The primary channel is the main data channel. The secondary channel is intended to serve as a low speed channel or as an auxilliary channel to convey status information.

Control

Associated with each of the two data channels are three control signals; Request to Send to the Data Communication Equipment (DCE); Clear to Send (from DCE) and Received Line Signal Detector (from DCE). Six additional signals are associated with the interface: Data Set Ready (from DCE), Data Terminal Ready (to DCE), Ring Indicator (from DCE), Signal Quality Detector (from DCE), and Data Signal Rate Selectors for both Data Terminal Equipment (DTE) and DCE.

These control lines serve several major functions:

1.OPERATIONAL STATUS: Data Terminal Ready (pin 20) is set by the DTE to indicate that it is functional (often a power-on indicator). Data Set Ready (pin 6) is the complimentary function performed by the DCE.

2.INITIATION OF DATA TRANSFER: Request to Send (pin 4) is activated by the DTE when it wishes to transmit data to the DCE; Clear to Send (pin 5) is the signal by which the DCE indicates that it is capalbe of receiving data from the DTE for transmission.

3.STATUS CHECKING: Signal Detect (pin 8) is set by the DCE to indicate that a carrier of sufficient amplitude is present. Signal Quality Detector (pin 21) is set by the DCE to indicate that the quality of communication is acceptable.

4.INITIATION OF LINK: Ring Indicator (pin 22) is set by the DCE to indicate that an incoming call is being initiated. While the majority of these signals are intended for interconnection of a terminal to a modem the user is free to assign them other functions, provided they are common to the interconnected devices.

Mechanical Interface

The RS-232C specification calls for a 25 pin connector, with the male part tied to the DTE and the female to the DCE. Consult Table 1 for RS-232C pin assignments.

NOTE: The reader is reminded that the RS-232C was initially designed as a communication interface standard hence the numerous pinouts. The simplest configurations can operate with a combination of 3 or 4 pins (the most common are *'d).

RS-232C PIN-OUT	FUNCTION
1	Protective ground
* 2	Transmitted Data
* 3	Received Data
* 4	Request to Send
* 5	Clear to Send
* 6	Data Set Ready
* 7	Signal Ground
8	Received Line Signal Detector
9	(Reserved for Data Set Testing)
10	(Reserved for Data Set Testing)
11	Unassigned
12	Secondary Rec'd Line Signal Detector
13	Secondary Clear to Send
14	Secondary Transmitted Data
15	Transmission Signal Element Timing
16	Secondary Received Data
17	Receiver Signal Element Timing
18	Unassigned
19	Secondary request to Send
20	Data Terminal Ready
21	Signal Quality Detector
22	Ring Indicator
23	Data Signal Rate Selector: DTE/DCE
24	Transmitter Signal Timing Element
25	Unassigned

TABLE 1

RS-232C PIN
ASSIGNMENTS

Foot-note

In the mid 1970's with increased peripheral sophistication made possible by integrated circuits new standards were clearly needed. On the initiation of Hewlett Packard (which was manufacturing a great number of these new sophisticated peripherals) the International Electical and Electronics Engineers issued it's 488th standard in 1975. Called appropriately enough the IEEE-488-1975. (A revision was issued in 1978.) Essentially the standards were based on PARALLEL rather than SERIAL data transmission.

Commodore has provided a PARALLEL User Port as well as an IEEE Port. Numerous methods have been described in micro-computer periodicals for simple and complex RS-232C circuits using either the IEEE or PARALLEL User Port.

Bibliography

IEEE-488/RS-232C Printer Adapter Prentice Orswell
Pet User Notes (Now Compute) Vol 1 Issue 1

IEEE-488 Bus, Jim Butterfield
The Transactor Vol.2 #5 (Oct.,1979)

IEEE Bus Handshake Routine in Machine Language J.A. Cooke
The Transactor Vol.2 #3 (July,1979)

IEEE Standard 488-1975: IEEE Standard Digital Interface for
Programmable Instrumentation   Hewlett-Packard

"Interface Between Data Terminal Equipment and Data
Communication Equipment Employing Serial Binary Data
Interchange"
Electronic Industries Association   Washington DC 20006

Microprocessor Interfacing Techniques   A. Lesea, R. Zaks
2nd ed. 1978   Sybex Berkeley Ca. 94704

TV Typewriter Cookbook   Don Lancaster
Howard W. Sams and Co. 1976 Indianapolis , Indiana 46268

Kilobaud Klassroom:#14 Parallel & Serial I/O   P.Stark
Kilobaud   Nov 1978 Issue # 23 Pg. 38

PET User Port Cookbook   Greg Yob
Kilobaud Microcomputing   March 1979   Pg.62
(Portions of this article also printed in The Transactor
Volume #1)

Parallel Port to RS-232C--Inexpensively   R.Hallen
Kilobaud Microcomputing   April 1979   Pg.62

Manufacturers of PET compatible RS-232C Interface:

Computer Associates Ltd.
1107 Airport Rd.,
Ames, Iowa 50010   (515) 233-4470

Connecticut microComputer, Inc.,
150 Pocono Rd.,
Brookfield, Ct., 06804 (203) 775-9659

Electronics Systems
P.O. Box 21638,
San Jose, Ca., 95151   (408) 448-0800

TNW Corporation,
5924 Quiet Slope Dr.
San Diego, Ca., 92120 (714) 225-1040

The following letter was received from PET user/enthusiast F. VanDuinen. It precedes his third article for the Transactor and contains a most unique request....

3 February 1980

Karl J. Hildon, Editor,
The Transactor
Commodore Business Machines, Ltd.
3370 Pharmacy Ave.
Agincourt, Ont. MlW 2K4

Dear Karl:

Here is another article for your newsletter. I do hope it is suitable for publication. Should you feel that it is worthwhile to revise it, such as make it less verbose, do not hesitate to let me know and I'll gladly oblige.

I also have a question I'd like to submit to the Transactor readers. I'd appreciate if you'd include it in whatever way you deem appropriate:

Many of the advantages of emulating one machine on another (also referred to sometimes as simulation), are well known. (A good example is the article '8080 Simulation with a 6502' by Dann McCreary in Micro, September '79, pp53-56.) There is one less obvious advantage, however. Consider a 6502 emulator (or simulator) to run on the 6502. That's right, emulate a machine on itself!

Such an emulator, provided it could handle breakpoints without modifying the code to be executed, and relocation of fields operated on, would be very useful in studying the function of code in Read Only Memory.

I'm looking for just such an emulator to learn more about the exact functioning of PET system routines. So if anybody knows of just such an emulator, let's hear about it through our newsletter, The Transactor.

F. VanDuinen,
175 Westminster Ave.
Toronto, Ont. M6R 1N9

F. VanDuinen
Toronto
30 Jan 80

PROGRAM PLUS

Overview

Many BASIC programs require assembler routines that are not part of the PET system (ROM), but that must be brought into memory before the program can execute properly. This article looks at techniques for SAVing these with the BASIC program, so they will be brought in automatically when the main program is LOADed.

One of these techniques can even be used to set PET operating system fields as part of the LOAD instruction. That allows such esoteric tricks as program protection and changing LOAD to LOAD-and-RUN.

The system used in the examples is an 8K old ROM PET with only tape storage. While these techniques are directly adaptable to new ROM PET, only a few have relevance to disk-based systems.

Multiple Files

The most straightforward way would be to have the various programs, BASIC and assembler, in individual consecutive files on the same tape. That way the main program would issue in sequence a LOAD for each of the other files.

Unfortunately that does not work. After the loading of each individual program, the PET updates BASIC's program pointers. Therefore the main BASIC program must be LOADed last. Also, the first program (assembler) must be started using the SYS command.

Simpler would be if everything could be SAVed together on one single file. The following techniques all do just that.

Following BASIC program

If the assembler routine is stored immediately following the end of program marker, it must be protected from variable storage. This can easily be done by setting the End of BASIC/Start of variables pointer (loc 124/125) to follow the appended code. As an added bonus, that is all that is required to cause the appended code to be SAVed with the BASIC program on the next SAVE. On subsequent LOADs all code will be brought into memory, and the End of BASIC/Start of Variables pointer will be automatically set from the end of program pointer in the program file header.

I don't know exactly how, but when there is a discrepancy between the End of BASIC pointer and the end of program as marked by the Next Instruction Pointer(NIP) chain, the End of BASIC pointer isused for the SAVE. This is in spite of the fact that the SAVE instruction does rebuild the NIP pointer chain.

The problem with this approach, of course, lies with BASIC program updates, (Analogous to Parkinson's third law, programs tend to expand untill they fill all available memory.) Every time the program is extended, the assembler code following it will have to be moved, thus necessitating changes to all absolute references (e.s. SYS, JMP, JSR etc.). This can to some extent be accomodated by leaving some unused space between the BASIC and the assembler code, but only at the dual cost of increased load time and reduced space for variable storage.

This approach of appending can be very nicely used to reserve memory space for tables etc., that will be created only at RUN-time, i.e. where the content of these locations at LOAD-time is irrelevant. I have used this tecchnique in the case of a BASIC program (not a compiler) that creates an assembler program and then SAVes it on tape. Most of the assembler code was constant and was carried as strings of hex characters in DATA statements in the BASIC program. Variable portions of the assembler program were then tailored based on input received the BASIC program and added to the constant code.

Because of memory constraints and the size of the target assembler program, it was necessary to create the latter in the space previously occupied by the DATA. The added variable portion, however, could be so large that the DATA space might be insufficient. All DATA statements were therefore set up at the very end of the program, with additional space reserved (but not used until execution time) by adjusting PET'S End of BASIC pointer. The start of the DATA statements was determined at execution time from loc 144/145, where PET leaves the address of the next DATA statement (after at least one READ).

Within BASIC

An interesting approach is that of storing assembler code within a BASIC program. While the technique is practical only for very short assembler routines, it does handle those very neatly.

The technique involves setting up a REM statement at the beginning of the program to set aside the space required for the assembler routine, and then pokins the assembler code in. A few conditions must be met:

.the End of Instruction marker (zero) and NIP pointers must not be disturbed
.the assembler code may not contain any zeroes, e.s. LDY #0 is out (use LDY #255 & INY to effect this)

.set up a quote mark immediately before the assembler object code, to accomodate listing the funny characters
.no BASIC statements should precede this carrier REM (any updates to these would relocate the assembler code)
.the carrier REM must be clearly marked as such, as LIST will not clearly indicate the assembler code.

More than one routine could be set up by using more than one carrier REM, however one routine per REM. A good example of this is a disassembler program in BASIC that needs an assembler routine to 'PEEK' at the region occupied by the BASIC interpreter (old ROM).

The following is an example of such code, showing both the way the BASIC program would look, and the assembler source code. The example shown is for a disassembler for both old and new ROM. (PEEK(50003) will return 1 (one) for new ROM, 0 (zero) for old.)

```
 10 REM DO NOT DELETE  '......statement carrying assembler
 20 POKE 1,23 : POKE 2,4      set up USR address as 1047
   .
   .
100 REM PEEK ROUTINE
110 IF PEEK(50003) THEN S1=PEEK(S1) : RETURN  handle new ROM
120 S1 = USR(S1) : RETURN                     handle old ROM
```

The assembler routine at 1047 could be as follows:

```
    20A7D0    JSR $D0A7     convert USR parameter to fixed pt.
    A0FF      LDY #255      *clear Y index register
    C8        INY           *
    B1B3      LDA (179),Y   get contents of specified byte
    2078D2    JSR $D278     set up USR value in F.P.
    60        RTS           return
```

<u>In File Header</u>

File headers are the same length as data blocks, 192 bytes. The system recognizes the various blocks from the record type in the first position:

1 - program file header
2 - data block
4 - data file header
5 - end of volume marker (OPEN .,.,2,.)

Following, that in the program file header, are the beginning and end addresses where the program is to be loaded (two 2 byte addresses). (In data file headers similar addresses are present. Those are merely the beginning and end of the buffer from which the file was written.)

Starting in byte 6 is the file name. While the name has a maximum length of 128 bytes, typically less than a quarter of that is used.

That leaves from (192-128-5)=59 to some (192-32-5)=155 bytes that could be used to carry something else. The main problem with this approach is that it is difficult to set up the assembler code.

One method is to key in the characters corresponding to the object code as part of the name. The format and length of the name are very critical that way. Furthermore, not all 255 possible codes are present on the keyboard.

Another way is as follows:

.issue a SAVE specifying the normal name etc, and immediately press the STOP/RUN key.
.this results in a proper file header in the buffer, and all pointers properly set up
.then POKE the assembler code into this header
.write out this header by:

```
POKE 633,100      (specify length of shorts to write)
                  (195 for new ROM)
SYS 63676+8       (write block with leader length as
                  set)
                  (63622+8(?) for new ROM)
```

.set up start and end of 'buffer' pointers at 247/248 and 229/230 respectively (251/252 and 201/202 for new ROM) to beginning and end of program to be saved
.write out program by:

```
SYS 63676         (write block preceded by standard
                  leader)
                  (63622 for new ROM
```

For subsequent program update, use can be made of the fact that the header and pointers have already been set up. Using the above sequence first, the existing header and then the updated programsegment can be saved.

A few caveats are in order, however:

.if the update changes the programs lenght, the header's end of program marker (in loc 4/5 of the header (639/640 or 831/832 absolute)) has to be updated from PET's End of BASIC/Start of Variables pointer 124/125 (new ROM 42/43)
.any tape I/O on the device from which the program was LOADed will also destroy the file header copy in the buffer

The VERIFY command may be used, if need be, to obtain a fresh copy of the file header without disturbing anything else.

Preceding BASIC

It is curious to reflect, that in a way the reason I'm writing this article is because Len Lindsay in his PET-Pourri column in Kilobaud (June 79, p6) talked about program protection that changed LOAD to LOAD-and-RUN, and disabled the STOP key. That got me intrigued, trying to figure out how that was done. Until suddenly my mental block cleared: why not load operating system data along with the program. That could set the RUN in the keyboard buffer, and the modified interrupt address. That, of course, was very smart and at the same time very wrong, as there is a special interrupt routine in use during tape read, and the system resets that to the normal interrupt routine address at the end of the LOAD. But at least it got me thinking in the right direction.

Normally when a BASIC program is SAVed, the starting address used is 1024 or $400. More precisely, the SAVE command gets its starting address from loc 122/123 (new ROM 40/41), PET's Start of BASIC pointer.

Consider, however, the possibilities of lower addresses; 826 (tape 2), 634 (tape 1), or even lower. That's right, why not include system fields! Set things like the keyboard buffer, interrupt addresses (careful there) and stuff like that.

To be sure, there are complexities in setting it up and scores of ways of crashing the system, but possibilities nonetheless.

During a LOAD operation, the system first reads the program file header into the appropriate buffer (tape 1 or tape 2). Then it transfers the start and end of program from the file header (2/3 and 4/5 in header) to loc 247/248 and 229/230 respectively (new ROM 251/252 & 201/202). Thus by the time the actual program segment is read in, the header is no longer required. If the start of program address is before the end of the tape buffer, the program segment will simply be stored on top of the header.

Looking at the system fields, starting at the end and working backwards we see a lot of fields that are not really relevant during a LOAD operation. Most of these standard values will do nicely. For instance, 553-577 (new ROM 224-248) contains the 'Line Address and Screen Wrap table'. Setting these up as after a clear screen should not affect most programs.

Some fields are critical, but predictable. For instance, the Hardware Interrupt Vector at 537/538 (new ROM 144/145) is critical (I believe). Predictable, however, as it should contain the address of the Tape Read Interrupt Routine, $F95F (new ROM $F931). The Stack (267-511) is also critical, unfortunately I have not the faintest idea what it contains during the loading of a program segment. I do believe it is constant during most of this process and is the same for every direct LOAD. (It will be different for LOADs issued from a program.)

I hope someone will investigate what the Stack looks like during this time and publish it.

Locations 247/248 and 229/230 are critical (at least 229/230 is), but are known to be as per the file header fields. All other fields are essentially immaterial.

That leaves of course the SAVing of the wanted values for these fields. While they are predictable or known during a LOAD, many of them are affected by a SAVE.

The trick is to copy all relevant fields and the entire BASIC program to a location where they are out of harms way, and SAVE them from there in such a way that they will be LOADed back into their original location.

The technique is to write a file header whose start and end of program addresses specify the desired LOAD location, and then write the program segment with PET's start and end of buffer pointers (247/248 and 229/230 respectively) pointing to the program's current location. The routine at the end of this article (Relocate and SAVE) will do just that

<u>Applications</u>

The ability to set system fields has a number of interesting applications. Program protection is but one of these. Another is the use of relocated BASIC programs.

The main trick to program protection is to ensure the user can not use Immediate Mode. Thus the program must not release control. There are at least the following items to consider:

.force automatic RUN by LOADing to keyboard buffer
 (don't forget cariage return and countfield)
.disable RUN/STOP key by modifying interrupt address at
537/538 (new ROM 144/145)
 use POKE 537,136 for old ROM, POKE 144,49 for new ROM
.do not use INPUT, use GET and ignore RUN/STOP

That leaves tape I/O. I don't know if the STOP key can be disabled there. It may be necessary to include assembler code that duplicates the tape read interrupt routine at $F95F, minus the check for STOP key, and further code to simulate INPUT# and PRINT# to ensure the address for the other routine is used in 537/538.

Unfortunately all that effort still would not make it foolproof. The way around it is still quite simple (as per Jim Butterfield's article on page 1 of Transactor #1, Vol 2). Instead of LOAD use:

```
SYS  62894                   to load the header
POKE 638,... : POKE639,...   to modify the area the program
                             is to be LOADed into
```

To avoid critical system fields, inspect the code using immediate PEEK instructions, and modify to disable the code that disables the STOP key. Also correct any pointers that may have been messed up to prevent the LIST function from being used. Then copy over the program to its proper location (using immediate instructions).

In Transactor #5, Vol 2, was an article (Memory Expansion, Cost $0.00) about using the tape buffers for BASIC program storage. As indicated in the article, before programs located there could be executed, certain PET system pointers had to be changed. Well, here's the way to set those pointers automatically.

The only time I've used this technique so far was for a loader program to load the object code written by my assembler program. The assembler program I'm using is written in BASIC, and resides at address $400 and up. So, when I assembled a program that was to reside there itself (and was too large to assemble in the few bytes not used for the assembler), I had no choice but to write it out to a file (one byte at a time). The, using a simple BASIC program, I could read each byte in and POKE it into consecutive locations, provided the loader program itself was not in the way. That program was thus created in the tape 2 buffer, and because it was small, did not use any memory above $400.

```
     RELOCATE & SAVE V0.0 22JAN80      PAGE 1

1 REM RTN TO SAVE & RELOCATE
2 REM F. VANDUINEN 22JAN80
10 EL = 2000                :REM END ADDR FOR LOAD
20 SL = 525                 :REM START ADDR FOR LOAD
30 SS = 2525                :REM START ADDR FOR SAVE
40 ES = SS + EL - SL        :REM END ADDR FOR SAVE
50 DN = 241                 :REM DEVICE NO      (212)
60 DB = 243                 :REM DEVICE NO PNTR (214)
70 B = 634                  :REM BUFFER ADDR
80 R1 = 63101               :REM RTN TO SET BUFFER START & END (63082)
90 R2 = 63763               :REM WAIT FOR I/O COMPL (63718)
100 R3 = 63676              :REM WRITE BLOCK (DATA PGM)    (63622)
110 REM R3 + 8 WRITE BLOCK WITH HEADER LENGTH SET IN 633 (195)
120 LL = 633    :REM LEADER LENGTH (SEC OF SHORTS B/4 DATA)(195)
130 BS = 247    :REM START OF BUFFER TO BE WRITTEN (PNTR)  (251)
140 BE = 229    :REM END OF BUFFER TO BE WRITTEN (PNTR)    (201)
150 D = 1       :REM TAPE NUMBER
200 REM         *CONSRUCT HEADER
210 POKE DN,D:M=DB:K=B:GOSUB900:FOR I=B TO B+191:POKE I,32:NEXT
220 POKE B,1    :REM SET FILE TYPE
230 M = B + 1 : K=SL : GOSUB900 : M = B + 3 : K = EL : GOSUB900
300 REM         *WRITE HEADER
305 PRINT "305"
310 SYS R1
315 PRINT "315"
320 SYS R2
330 POKE LL,100 : SYS R3+8
335 PRINT "335"
400 REM         *MOD POINTERS
410 M = BS : K = SS : GOSUB900 : M = BE : K = ES : GOSUB900
450 REM         *WRITE PROGRAM BLOCK
460 SYS R3
500 END
900 I = INT (K/256) : J = K - 256 * I : POKE M,J : POKE M+1,I
:RETURN
```

commodore

comments and bulletins
concerning your
COMMODORE PET™

The Transactor

VOL 2
BULLETIN # 9

PET™ is a registered Trademark of Commodore Inc.

Bits and Pieces

Printer Tabbing

When using TAB to print on the screen, PET looks at the current position of the cursor first (POS(0)). If the TAB argument is less than the cursors' position on the line then the data is simply printed in the spaces immediately following the last character printed. If the argument is greater than or equal to POS(0), PET subtracts POS(0) from the argument and prints the resulting number of cursor-rights.

However, when printing to the printer, the cursor is usually in column zero and TAB acts like the SPC function (the printer has no "internal cursor"). Therefore, to make TAB work on the printer, print the data to the screen first then to the printer. This can be done with duplicate PRINT and PRINT# statements or more efficiently with one "dynamic" PRINT# statement. For example:

```
10 REM OPEN OUTPUT FILES TO
      SCREEN & PRINTER
20 OPEN 3,3,1
30 OPEN 4,4,0
40 PRINT# 3+X,"ABCDEFGHIJKLMNOPQRSTUVW";
50 X=1-X : IF X THEN 40
```

Line 50 toggles X from 1 to 0 thus repeating line 40 only twice. The semi-colon is important else the POS(0) goes back to zero. When a carriage return is required on the printer the following might be inserted between PRINT# and toggle statements:

```
45 IF X THEN PRINT#4,CHR$(13);
```

Dynamic PRINT# statements are only more efficient if the DATA being printed is within quotes. If variables are used, more bytes are probably saved by duplicating the output statements.

The Transactor is now produced on the new CBM 8032 using WordPro IV and the NEC Spinwriter.

Input and Output from PET Machine Language

Jim Butterfield
Toronto

Output to the screen is quite straightforward. Load the ASCII character into the A register; then call the routine at FFD2. Special characters, such as cursor movements, will be honoured in the usual way.

The GET activity gives no trouble, either, except for one minor situation. To do a GET, call FFE4 and the character will appear in the A register. If you don't have a character available, the subroutine will return zero in the A register. Since you can't get an ASCII zero from the keyboard, recognize this as a "no-character" situation and arrange to deal with it as desired.

INPUT is a little trickier. When you call FFCF for Input, you'll get one character back. This seems like a GET, but it's really quite different. The first time you call, it will prompt and get an input, transfering it via the screen in the usual way; then it will edit out leading and trailing spaces and quote marks. After doing all this work, it will deliver the first character to you. On subsequent calls, it will deliver following characters. When it has delivered the whole input, it will deliver a Return character to signal you've got it all. After that, it starts over.

Beginners will be happier using the GET call.

Peripheral Input/Output

Surprisingly easy, once you have the above techniques mastered.

Start by OPENing the file in BASIC, before you go to machine language. When you're ready to the actual activity, the machine language sequence is as follows:

Load X with the logical file number;
For INPUT or GET, call FFC6 to set the input channel;
For output, call FFC9 to set the output channel;

Now use you INPUT, GET, or output calls as described above;

Finally, restore the normal input/output channels with a call to FFCC. Careful! This routine changes the A register to zero.

Wind up your program in BASIC by closing all files, as usual.

When you're INPUTting or GETting from an external device, keep an eye on the status word, ST (located at 020C in original ROM, or at 96 in 2.0 ROM). It will warn you when you reach the end of a input file.

The above procedure isn't too hard, and it's likely to carry through to newer versions of ROM when they appear.

An Instring Utility for the 16/32K PET

Have you ever wanted to program something like...

MID$ (A$, 10) = "Name, Address, etc..."

...well now you can!...thanks to another fabulous routine by Bill Maclean of BMB CompuScience, Milton Ontario. The routine works only with PETs using the 2.0 ROM set.

This is a little utility to allow a programmmer to change a substring within a main string. Its primary uses are manipulating data records in disk files and setting up formatted printer or screen outputs. It is called with the following command

SYS 826,A$,B$,X

This command string will cause the string A$ to be placed within string B$, starting at the 8th character. The A$,B$,and X are all variables. Any variables can be used. The programmer is responsible for assuring that the length of the main string is not exceeded.

The machine language routine can be entered using the resident monitor and cursor editing the screen display. The code is completely relocatable and can be placed anywhere or relocated anywhere. The calling address (826 above) should be the address of the first byte of the program.

```
      PC   IRQ  SR AC XR YR SP
.;   0005 E62E 30 00 5E 04 F5
.M 033A 0382
.:   033A 20 F8 CD 20 9F CC A0 00
.:   0342 B1 44 85 00 C8 B1 44 85
.:   034A 01 C8 B1 44 85 02 20 F8
.:   0352 CD 20 9F CC A0 01 B1 44
.:   035A 85 0F C8 B1 44 85 10 20
.:   0362 F8 CD 20 9F CC 20 D2 D6
.:   036A A5 12 F0 03 4C 03 CE C6
.:   0372 11 A5 0F 18 65 11 85 0F
.:   037A 90 02 E6 10 A0 00 B1 01
.:   0382 91 0F C8 C4 00 D0 F7 60
```

PET as an IEEE-488 Logic Analyzer

Jim Butterfield, Toronto

If you'd like to see what's going on on the GPIB - and if you can borrow an extra PET and IEEE interface cable - this program will help.

It shows the current status of four of the GPIB control lines, plus a log of the last nine characters transmitted on the bus.

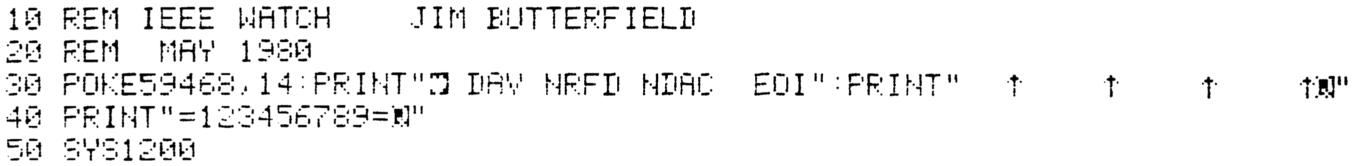
The four control lines are NRFD, NDAC, DAV and EOI. It would be nice to show ATN too, but I couldn't fit this in: it's detected in a rather odd way in the PET so that fitting it in is somewhat too tricky for this simple program.

The last nine characters are shown in "screen format". This means that you'll have to do a little translation work to sort out what some of them mean. On the other hand, it allows you to see characters that otherwise wouldn't be printed. A carriage return, for example, shows up as a lower case m; this is a little confusing at the start, but you'll quickly get used to it and it's handy to see everything that goes through. Don't forget that original model PETs may show upper and lower case reversed.

I had hoped to show which characters were accompanied by the EOI signal. It turned out that time is critical - the bus works very fast - and that adding this feature would cut down the number of displayed characters from nine to five. I opted for the bigger count and dropped the EOI log feature.

The high speed of the bus makes it difficult to watch the control lines in real time. When the "active" PET is exchanging information with disk or printer, everything is happening very fast, and the "logic analyzer" PET will show an amazing flurry of activity on the control lines. Only when the activity stops or hangs up will you be able to see the lines in their static conditions.

You may use the program to chase down real GPIB problems, or just to gain insight on how the bus works. Either way, it will come in handy if you can borrow that extra PET unit.



```
10 REM IEEE WATCH     JIM BUTTERFIELD
20 REM  MAY 1980
30 POKE59468,14:PRINT"❏ DAV NRFD NDAC  EOI":PRINT"  ↑    ↑    ↑     ↑■"
40 PRINT"=123456789=■"
50 SYS1200
```

```
                      ; IEEE WATCH   JIM BUTTERFIELD MAY/80
110:   04B0                     *=    $4B0
120:   04B0          DFLAG      =     $B1
130:   04B0          DNNSAV     =     $B2
140:   04B0          EOISAV     =     $B3
200:   04B0 46 B1    START      LSR   DFLAG
210:   04B2 78                  SEI
220:   04B3 AD 12 E8 MAIN       LDA   $E812
230:   04B6 C9 EF               CMP   #$EF
240:   04B8 D0 02               BNE   CONT
250:   04BA 58                  CLI
250:   04BB 60                  RTS
280:   04BC AC 10 E8 CONT       LDY   $E810        ;EOI
290:   04BF AD 40 E8            LDA   $E840        ;DAV, NRFD, NDAC
300:   04C2 AE 20 E8            LDX   $E820        ;DATA
310:   04C5 29 C1               AND   #$C1         ;EXTRACT BITS
320:   04C7 C5 B2               CMP   DNNSAV
330:   04C9 D0 11               BNE   DNN
340:   04CB 98                  TYA
350:   04CC 29 40               AND   #$40         ;EXTRACT EOI
360:   04CE 0A                  ASL   A
370:   04CF 49 A0               EOR   #$A0
380:   04D1 C5 B3    EOI        CMP   EOISAV
390:   04D3 F0 DE               BEQ   MAIN
400:   04D5 85 B3               STA   EOISAV
410:   04D7 8D 61 80            STA   $8061
420:   04DA D0 D7               BNE   MAIN
                      ;ACTIVITY SEEN - UPDATE SCREEN
430:   04DC 85 B2    DNN        STA   DNNSAV
440:   04DE 29 80               AND   #$80
450:   04E0 49 A0               EOR   #$A0
460:   04E2 8D 52 80            STA   $8052
470:   04E5 10 1D               BPL   NDAV         ;NO DAV SEEN
500:   04E7 A4 B1               LDY   DFLAG
510:   04E9 30 1B               BMI   DCONT        ;DAV SEEN BEFORE
520:   04EB 85 B1               STA   DFLAG
530:   04ED 85 B2               STA   DNNSAV
540:   04EF A0 00               LDY   #0
550:   04F1 B9 A2 80 SCROL      LDA   $80A2,Y
560:   04F4 99 A1 80            STA   $80A1,Y
570:   04F7 C8                  INY
580:   04F8 C0 08               CPY   #8
590:   04FA D0 F5               BNE   SCROL
600:   04FC 8A                  TXA
600:   04FD 49 FF               EOR   #$FF
600:   04FF 8D A9 80            STA   $80A9
610:   0502 B0 AF               BCS   MAIN
640:   0504 85 B1    NDAV       STA   DFLAG
650:   0506 A5 B2    DCONT      LDA   DNNSAV
660:   0508 29 40               AND   #$40         ;NRFD
670:   050A 0A                  ASL   A
680:   050B 49 A0               EOR   #$A0
690:   050D 8D 57 80            STA   $8057
700:   0510 A5 B2               LDA   DNNSAV
710:   0512 29 01               AND   #$1          ;NDAC
720:   0514 4A                  LSR   A
730:   0515 6A                  ROR   A
740:   0516 49 A0               EOR   #$A0
750:   0518 8D 5C 80            STA   $805C
760:   051B D0 96               BNE   MAIN
```

CROSS - REFERENCE

Jim Butterfield, Toronto
also in Compute #4

One of the handy things about the 2040 disk system is that it allows you to read programs - or write them, for that matter - as if they were data files.

The possibilities are endless: you can analyze or cross-reference programs; renumber them; repack them into minimum number of lines deleting spaces, comments, etc.; or even create a program-writing program that is tailor-made for a particular job.

This program does cross-referencing of a BASIC program. It's written in BASIC: that means that it won't run too fast (all those GET statements) but you can read what it's doing fairly easily.

There are two types of cross-references normally needed for a BASIC program. One is the variable cross-reference: where do I use B$? The other is a line-number cross-reference: when do I go to line 360 ? CROSS-REF does either. An example of both types is shown - the program in this case did the cross-references of itself.

```
CROSS REFERENCE - PROGRAM CROSS-REF


A        270    280    300    310    390    400
A$       180    240    260    270    300    460    490    500    510    520    560
         570    580
A$(      100    200    330    340    350    360
B        190    200    220    320    330    340    350    360
B$       180    240    480    500    520    580    590
B$(      100    120    480
C        280    370    390    410    420    430    440    450    540    550
C$       480    520    580    620
C(       100    140    150    160    310
C1       280    310    370    420    440
C2       130    150    205    280    370    380    450    565
C9       310    410    420    440    470    480
J        140    150    200    210    220    330    340    350    560    630
K        200    210    220    320    340    350    360    565    570    580
L        260    280
L$       200    206    280
M$       330    340    360    370    450    460
P$       170    550
Q$       120    510
S$       120    280    590    610    620
X        200    220    560
X$       200    205    206    210    220    560    580    590
X$(      100    210    220    560
Y        590    600    610
Z        540    600
Z$       130    530    540
```

```
100 DIM A$(15),B$(3),X$(500),C(255)
110 PRINT"CROSS-REF    JIM BUTTERFIELD"
120 Q$=CHR$(34):S$="     ":B$(1)=Q$:B$(3)=CHR$(58)
130 INPUT"VARIABLES OR LINES";Z$:C2=5:IFASC(Z$)=76THENC2=6
140 FORJ=1TO255:C(J)=4:NEXTJ:FORJ=48TO57:C(J)=6:NEXTJ
150 IFC2=5THENFORJ=65TO90:C(J)=5:NEXTJ:FORJ=36TO38:C(J)=7:NEXTJ:C(40)=8
160 C(34)=1:C(143)=2:C(131)=3
170 INPUT"PROGRAM NAME";P$:OPEN1,8,3,"0:"+P$+",P,R"
180 GET#1,A$,B$:IFASC(B$)<>4THENCLOSE1:STOP
190 IFB=0GOTO240
200 PRINTL$;:K=X:FORJ=BTO1STEP-1:PRINT" ";A$(J);:X$=A$(J)
205 IFC2=6ANDLEN(X$)<5THENX$=" "+X$:GOTO205
206 X$=X$+L$
210 IFX$(K)>=X$THENX$(K+J)=X$(K):K=K-1:GOTO210
220 X$(K+J)=X$:NEXTJ:X=X+B:PRINT:B=0
230 REM: GET NEXT LINE, TEST END
240 GET#1,A$,B$:IFLEN(A$)+LEN(B$)=0GOTO530
250 REM GET LINE NUMBER
260 GET#1,A$:L=LEN(A$):IFL=1THENL=ASC(A$)
270 GET#1,A$:A=LEN(A$):IFA=1THENA=ASC(A$)
280 C=C2:C1=-1:L=A*256+L:L$=STR$(L):IFLEN(L$)<6THENL$=LEFT$(S$,6-LEN(L$))+L$
290 REM GET BASIC STUFF
300 GET#1,A$:A=LEN(A$):IFA=1THENA=ASC(A$)
310 C9=C(A):IFC9>C1GOTO380
320 K=0:IFB=0GOTO360
330 FORJ=1TOB:IFA$(J)=M$GOTO370
340 IFA$(J)<M$THENNEXTJ:K=B:GOTO360
350 FORK=BTOJSTEP-1:A$(K+1)=A$(K):NEXTK
360 B=B+1:A$(K+1)=M$
370 C=C2:C1=-1:M$=""
380 IFC2=5GOTO420
390 IFA=137ORA=138ORA=141ORA=167THENC=6:GOTO470
400 IFA=44ORA=32GOTO470
410 IFC9<>6THENC=9:GOTO470
420 IFC9=CTHENC=-1:C1=4
430 IFC>6GOTO470
440 IFC<0ANDC9>C1ANDC9>6THENC1=C9:GOTO460
450 IFC2=5THENIFLEN(M$)>2ORC>0GOTO470
460 M$=M$+A$
470 ONC9+1GOTO190,480,480,480:GOTO300
480 B$=B$(C9):C$=""
490 GET#1,A$:IFA$=""GOTO190
500 IFA$=B$GOTO300
510 IFA$<>Q$GOTO490
520 A$=B$:B$=C$:C$=A$:GOTO490
530 CLOSE1:INPUT"PRINTER";Z$
540 C=3:Z=6:IFASC(Z$)=89THENC=4:Z=12
550 OPEN2,C:PRINT#2:PRINT#2,"CROSS REFERENCE - PROGRAM ";P$
560 X$="":FORJ=1TOX:A$=X$(J)
565 IFC2=6THENK=6:GOTO580
570 FORK=1TOLEN(A$):IFMID$(A$,K,1)<>" "THENNEXTK:STOP
580 B$=LEFT$(A$,K-1):C$=MID$(A$,K+1):IFX$=B$GOTO600
590 PRINT#2:Y=0:X$=B$:PRINT#2,X$;LEFT$(S$,5-LEN(X$));
600 Y=Y+1:IFY<ZGOTO620
610 Y=1:PRINT#2:PRINT#2,S$;
620 PRINT#2,LEFT$(S$,6-LEN(C$));C$;
630 NEXTJ:PRINT#2:CLOSE2
```

```
190    470    490
205    205
210    210
240    190
300    470    500
360    320    340
370    330
380    310
420    380
460    440
470    390    400    410    430    450
480    470
490    510    520
530    240
580    565
600    580
620    600
```

Reading a BASIC Program as a File

To read a BASIC program, you must OPEN it as a file, using type P for PRG rather than S for SEQ. Line 170 of CROSS-REF does this.

If you read a zero character from the program (that's CHR$(0), not ASCII zero which has a binary value of 48), the GET# command gives you a small problem: it will give you a null string instead of the CHR$(0) you might normally expect. You need to watch this condition and correct it where necessary: you'll see this type of coding in lines 260, 270 and 300.

The first thing to do when you OPEN the file is to get the first two bytes. These represent the program start address, and should be CHR$(1) and CHR$(4) for a normal BASIC program starting at hexadecimal 0401 (see line 180).

Now you're ready to start work on a line of BASIC. The first two bytes are the forward chain. If they are both zero (null string) we have reached the end of the BASIC program; otherwise, we don't need them for this job (see line 240).

Continuing on the BASIC line: the next pair of bytes represent the line number, coded in binary. We're likely to need this, so we calculate it as L (lines 260 and 280) and also create it's string equivalent, L$. We take an extra moment to right-justify the string by putting spaces at the front so that it will sort into proper numeric order.

From this point on we are looking at the text of the BASIC line until we reach a zero which flags end-of-line. At that time we go back and grab the next line.

Detailed Syntax Analysis

When digging out variables or line numbers, we have several jobs to do. As we look through the BASIC text, we must find out where the variable or line number starts. For a variable, that's an alphabetic character; for a line number, it's the preceeding keyword GOTO, GOSUB, THEN or RUN followed by an ASCII numeric.

Once we've "aquired" the variable or line number, we must pick up its following characters and tack them on. For line numbers it's strictly numeric digits. For variables, things are more complex. Both alphabetic and numeric digits are allowed, but we should throw away all after the first two since GRUMP and GROAN are the same variable (GR) in PET BASIC. We must also pick up a type identifier - % for integer variables or $ for strings - if present. Finally, we have to spot the left bracket that tells us we have an array variable.

To help us do this rather complex job, we construct a character type table. Each entry in the table represents an ASCII character, and classifies it according to its type. Numeric characters are type 6. If we're looking for variables, alphabetic characters are type 5, identifiers are type 7, and the left bracket is type 8.

To help us in scanning the BASIC line, we define the end-of-line character as type 0; the quotation mark as type 2; the REM token as type 3; and the DATA token as type 4.

Every time we get a new character from BASIC, we get its type from table C as variable C9. If we're looking for a new variable or line number, we see if it matches C - alphabetic for variables, numeric for line numbers. Once we find the new item, we kick C out of range and start searching based on the value of C1. This mechanism means that we can search for a variable starting with an alphabetic, and then allow the variable to continue with alphabetics, numerics or whatever.

To summarize variables in this area: A is the identity of the character we have obtained from the BASIC program, and C9 is its type. If we're searching, C is the type we are looking for; otherwise it's kicked out of range, to -1 or 9. C1 tells us we're collecting characters and what type we're allowed to collect. C2 is out variables/line numbers flag; it tells us what we're looking for. M$ is the string we've assembled.

The routine from 480 to 520 scans ahead to skip over strings in quotes and DATA and REM statements.

Collecting the Results

For each line of the BASIC program we are analyzing, we collect and sort any items we find, eliminating duplicates. They are staged in array A$ in lines 320 to 370.

When we're ready to start a new line, we add this table to our main results table, array X$, in lines 200 to 220. To save sorting time, we merge these pre-sorted values into the main table. At this point, our data has the line number stuck on the end; this way, we're handling two values within a single array.

Because the merging of the two tables must start at the top so that we can make room for the new items, the items are handled in reverse alphabetic order. We print this to the screen so that you can watch things working. At BASIC speed, this program can take quite a while to run; it's nice to confirm that the computer is doing something during this period.

Final Output

We finish the job starting at line 530. It's mostly a question of breaking the stuck-together strings apart again and then checking to see if we need to start a new line.

Do Your Own Thing

The size of array X$ determines how large a program you can handle. The given value of 500 is about right for 16K machines; with 32K you can raise it to 1500 or so.

If you're squeezed for space, change array C to an integer array C%. As you can see from the cross reference listing, you'll need to change lines 100, 140, 150, 160 and 310 - see how handy the program is ?

As mentioned before, run time is slow. A machine language version - or even a BASIC program with machine language inserts - would speed things up dramatically.

NOTE: Some ASCII printers may give double spaced output. If this is a problem the PRINT#2 statements in 590 and 610 should be changed to PRINT#2,CHR$(13);.

Better Auto Repeat

David Berezowski of ASCII Computing, Thunder Bay Ontario, has submitted another repeat key program which might be used instead of the one printed in Transactor #7.

```
0 REM RELOCATABLE AUTO-REPEAT BY...
1 REM DAVID BEREZOWSKI
2 REM ORIGINAL CODE TAKEN FROM BEST OF TRANS. VOL 1
3 REM UPDATED FOR NEW ROM AND PUT INTO RELOCATABLE FORM BY DAVID BEREZOWSKI
4 REM RELOCATABLE FORMAT TAKEN FROM J. BUTTERFIELDS TRACE ROUTINE
5 K=0
10 PRINT"■■THIS PROGRAM LOCATES ■AUTO-REPEAT■ IN"
20 PRINT"ANY SIZE MEMORY THAT IS FITTED...■"
30 IFPEEK(65000)=254THENE=52:D=0:GOTO60
40 IFPEEK(65000)<>192THENPRINT"?? I DON'T KNOW YOUR ROM ??":END
50 E=134:D=4:K=3:FORJ=1TO56:READZ:NEXTJ
60 PRINT"I SEE THAT YOU HAVE AN ";
70 IFE=134THENPRINT"ORIGINAL";
80 IFE=52THENPRINT"UPGRADE";
90 PRINT"  R O M."
95 FORZ=1TO2000:NEXT
100 DATA 162,3,181,255,157,45,3,202,208
101 DATA 248,56,169,233,229,145,133,145
102 DATA 96,165,166,201,255,208,10,169
103 DATA 0,133,15,169,48,133,16,208,19
104 DATA 230,15,165,16,197,15,176,11
105 DATA 169,6,133,16,162,255,134,151
106 DATA 232,134,15,76,46,230
150 REM***************************
160 REM*  END OF UPGRADE DATA *
170 REM***************************
200 DATA 162,3,181,255,157,122,3,202,208
201 DATA 248,56,169,233,237,25,2,141,26,2
202 DATA 96,173,35,2,201,255,208,10,169
203 DATA 0,133,60,169,48,133,61,208,20
204 DATA 230,60,165,61,197,60,176,12
205 DATA 169,6,133,61,162,255,141,3,2
206 DATA 232,134,60,76,133,230
1000 S2=PEEK(E)+PEEK(E+1)*256
1005 S1=S2-56-D
1010 FORJ=S1TOS2-1
1020 READX:POKEJ,X:NEXTJ
1030 S=INT(S1/256):T=S1-S*256
1040 POKE0,76:POKE1,T+19+D/2:POKE2,S
1050 POKEE,T:POKEE+1,S
1060 POKEE-4,T:POKEE-3,S
1130 PRINT"■■===AUTO-REPEAT==="
1140 PRINT"■TO ENABLE: SYS"S1
1150 PRINT"TO DISABLE: SYS"S1
1160 PRINT"■CHANGE SPEED WITH: POKE"S1+40+K"■,X
1170 PRINT"CHANGE DELAY WITH: POKE"S1+29+K"■,X (X>10)
1180 PRINT"■■TO EXPERIENCE THE FRUSTRATION FROM"
1190 PRINT"■KEY-BOUNCE THAT ALL TRASH-80 OWNERS"
1200 PRINT"■MUST PUT UP WITH, TRY POKING "S1+29+K
1210 PRINT"■WITH VALUES LESS THAN 5'"
1220 PRINT"■NOTE:YOU MUST DISABLE AUTO-REPEAT"
1230 PRINT"BEFORE USING THE CASSETTE"
```

MACHINE LANGUAGE CASE CONVERTER   S. Donald   Rossland, B.C.

This machine language program will convert strings to the correct upper/lower case condition for printing on CBM 2022/23 printers with an original ROM PET. It is relocatable so will operate anywhere in memory. The routine given here puts it in the second cassette buffer, but changing the location given in line 10100 will place it wherever you wish.

There are several things which must be done in order for the routine to operate correctly. These are best demonstrated by the following program.

```
0  ML$="" : GOSUB 10000
10 POKE 59468, 14
20 ML$="az123AZ"
30 PRINT ML$ : OPEN 4,4 : PRINT#4,ML$
40 SYS 826
50 PRINT#4,ML$ : CLOSE 4
60 PRINT ML$
70 LIST
10000 DATA 160,   2, 177, 124, 141, 251
10010 DATA   0, 200, 177, 124, 141, 252
10020 DATA   0, 200, 177, 124, 141, 253
10030 DATA   0, 172, 251,   0, 136, 177
10040 DATA 252, 201, 219, 176,  22, 201
10050 DATA 193, 144,   5,  56, 233, 128
10060 DATA 208,  11, 201,  65, 144,   9
10070 DATA 201,  91, 176,   5,  24, 105
10080 DATA 128, 145, 252, 192,   0, 208
10090 DATA 223,  96
10100 FOR A = 826 TO 881 : READ B
10110 POKE A, B : NEXT : RETURN
```

Note that line 20 is altered once the program is RUN. This is done by the SYS command in line 40.

Now alter line 20 to:

```
20 ML$ = ML$ + "az123AZ"
```

and reRUN from line 0. This time line 20 has not been changed in the listing. Whenever a string is formed by concatenation, the new string is stored in a location different from the original strings i.e. up in high RAM. It is this new location that has been altered. The major advantage in working on a string stored away from the program listing is that you don't have to worry if the string has been previously altered.

Now change line 0 to:

```
0  A = 0 : GOSUB 10000
```

and reRUN from line 0. Two points to note are:

1. Make sure that the variable string to be printed is #1 in the variable table, and
2. form the string to be printed by concatenation.

ASSEMBLY LANGUAGE LISTING OF UPPER/LOWER CASE CONVERTER

MOVE VARIABLE POINTERS TO ZERO PAGE

```
LDY 2        Set Y register offset.
LDA 124,Y    Load A with byte from variable table pointed to by
             124/125 + Y
STA 251,0    and move to location 251.  This byte is the
             character count.
INY          Increment offset.
LDA 124,Y
STA 252,0
INY          Shift start address of string to zero page.
LDA 124,Y
STA 253,0
```

ADJUST STRING

```
LDY 251,0    Load Y with string character count from location 251.
DEY          Decrement Y offset.  Y points to character to be
             altered next.
```

TEST FOR LOWER CASE

```
LDA 252,Y    Load A with string byte pointed to by 252/253
             and offset by Y
CMP 219      and compare to lower case 'z' and
BCS 22       if greater than, skip to COMPARE Y.
CMP 193      Compare to lower case 'a' and
BCC 5        if less than skip to TEST FOR UPPER CASE.
```

ADJUST LOWER CASE

```
SEC          Set carry flag
SBC 128      Subtract 128 from the string byte in A and
BNE 11       always skip to STORE MODIFIED CHARACTER.
```

TEST FOR UPPER CASE

```
CMP 91       Compare to upper case 'Z' and
BCS 9        skip to COMPARE Y if greater than.
CMP 65       Compare to upper case 'A' and
BCC 5        skip to COMPARE Y if less than.
```

ADJUST UPPER CASE

```
CLC          Clear carry flag.
ADC 128      Add 128 to string byte.
```

STORE MODIFIED CHARACTER

```
STA 128,Y    Store byte at location pointed to by
             252/253 and offset by Y.
```

COMPARE Y FOR STRING END

```
CPY 0       Compare Y to '0' and
BNE 223     skip to DEY in ADJUST STRING if string
            not finished.
RTS         Otherwise, return to BASIC.
```

RESTORE DATA Line Program

Paul Barnes
Deseronto, Ontario

Executing the RESTORE command causes the next READ to occur at the very first DATA element in your program. This subroutine can be used to RESTORE the DATA line pointer at a line other than the first.

It doesn't matter if you don't give the number of the line that has the "DATA" keyword in it that you want to start at, as long as it is past previous DATA statements so that the next data to be read will be the one desired.

```
      4 REM *******************************
      5 REM ***   RESTORE DATA LINE PGM   ***
      6 REM ***       BY PAUL BARNES      ***
      7 REM ***     DESERONTO, ONTARIO    ***
      8 REM *******************************
     10 DATA 166, 142, 134,   8, 166, 143
     10 DATA 166,  60, 134,  17, 166,  61
     15 DATA 134,   9,  32,  34, 197, 144
     15 DATA 134,  18,  32,  44, 197, 144
     20 DATA  11, 166, 174, 142, 132,   3
     20 DATA  11, 166,  92, 142, 132,   3
     25 DATA 166, 175, 142, 133,   3   96
     25 DATA 166,  93, 142, 133,   3   96
     30 DATA 162,   0, 142, 132,   3, 162
     35 DATA   0,  42, 133,   3,  96
     40 FOR F = 826 TO 860 : READ S : POKE
  F, S : NEXT
     50 DATA "GOOD-BYE!"
     60 DATA "ANYBODY HOME?"
     70 J = 26545 * 10 : FOR D = 1 TO 100 :
  DATA "MAYBE!"
     80 NEXT : DATA "HI!"
    100 DATA "GO HOME!"
    110 DATA "GO DIRECTLY TO JAIL!"
    120 DATA "DO NOT PASS GO!"
    130 DATA "DO NOT COLLECT $100!!!"
    200 GOSUB 1000
    210 FOR T = 1 TO 3 : READ A$ : PRINT A$
    230 NEXT : PRINT : GOTO 200
    998 REM *** SUBROUTINE TO RESTORE DATA
    999 REM ***  AT A CERTAIN LINE NUMBER
  1000 INPUT "RESTORE TO LINE"; A
  1010 H = INT (A/256) : L = A - H * 256
  1020 REM POKE CURRENT DATA LINE POINTER
  1030 POKE 142, L : POKE 143, H
  1030 POKE  60, L : POKE  61, H
  1040 SYS 826
  1050 L = PEEK(900) : H = PEEK(901)
  1060 IF L=0 AND H=0 THEN PRINT "LINE NOT
  FOUND" : GOTO 1000
  1070 REM POKE MEMORY ADDRESS OF DATA LINE
  1080 A = H * 256 + L - 1 : H = INT(A/256)
  : L = A - H * 256
  1090 POKE 144, L : POKE 145, H
  1090 POKE  62, L : POKE  63, H
  1100 RETURN
```

Editors Note: Paul has submitted the above program for the Original ROM set. The duplicate underlined statements are for BASIC 2.0 ROM.

commodore

comments and bulletins
concerning your
COMMODORE PET™

The Transactor

VOL 2
BULLETIN # 10

PET™ is a registered Trademark of Commodore Inc.

REMAINDER$

One little known use of the MID$ function is "remainder string". If the third parameter of the MID$ function is omitted the resulting string will be every character to the right of the specified start position for the string being operated on. For example:

```
1. A$ = "123456789"
2. B$ = MID$ ( A$, 2, 4 )    ;equals "2345"
3. B$ = MID$ ( A$, 2 )       ;equals "23456789"
```

This is not the same as RIGHT$ as this function returns an absolute number of characters starting from the rightmost position. This application works best when the right-hand portion of a string is wanted and the string length is not known.

BASIC 4.0 Preliminary Note

BASIC 4.0 ROMs for the 40 column PET are on their way! The main differences are:

1. Faster garbage collection
2. Disk commands included in BASIC

Of course most SYStem calls to ROM will require modification but PEEKs and POKEs should remain valid except for some locations that may have been labelled unused in BASIC 2.0. More on BASIC 4.0 in a later issue. Also see Jim Butterfield's new BASIC 4.0 memory maps, this issue.

All BASIC 2.0 programs will run on BASIC 4.0 except for one minor gotcha. BASIC 4.0 has reserved two more variables for it's own use; DS and DS$. When called, DS will contain the error number from the disk and DS$ will return the error number, description, track and sector much like hitting ">" and return with DOS Support. The same rule applies to DS and DS$ as ST, TI and TI$; they must not appear on the left of an "=" sign. If they do a ?SYNTAX ERROR will result. So if your programs use either of these two new reserved variables, it would be a good idea to change them before RUNning on BASIC 4.0. This could be easily done by running your programs through Jim Butterfield's Cross-Ref program from Transactor #9, Vol 2.

The Transactor is produced on the new CBM 8032 using WordPro IV and the NEC Spinwriter.

ID Changer

COMPUTE magazine, issue #5, published an article that allows the user to change the ID of a diskette. This can cause irreparable damage to your disks! The program changes only the the ID that gets printed with the directory. However, the ID precedes every sector on the disk and these do not get changed. An update will be published in the next COMPUTE but this early warning will be appreciated by some I'm sure.

Printer ROMs

Recent deliveries of Commodore printers have been released with the 04 ROM. Though this ROM fixes existing 03 ROM bugs, it has a tendency to lock into lower case, inhibiting upper case character printing. This happens after sending to secondary address 2 (receive data for format). Commodore has discontinued the 04 printer ROM and until the 08 ROM is released (sometime in the fall) the following software fix will prevent this bug from appearing. Lines 30 and 40 insert a 25 jiffy delay prior to OPENing the format channel:

```
10 OPEN 4, 4, 0
20 PRINT#4, "HELLO"
30 T = TI
40 IF TI - T < 25 THEN 40
50 OPEN 5, 4, 1
60 PRINT#5, "  AAA 999    ...etc.
```

This bug can also be used to your advantage i.e. for LISTing to the printer in lower case which was, in most cases, impossible on printers containing an 03 ROM. There is, however, an easier way of implementing it:

```
100 OPEN 7, 4, 7 : PRINT#7 : CLOSE 7
```

...puts the printer in lower case mode. Power down and up gets you back to upper case and graphics.

PRINT Speed - Up

In Transactor #2, Vol 2, a POKE was published that made PRINT to the screen much faster than normal. On recent machines this POKE can not only cause the machine to crash but may also result in internal damage! Avoid including this in your programs...especially those that you may want to RUN on other peoples machines. Software portability is very important, particularly business software. If your package crashes your clients machine, you may find yourself in a very embarassing situation.

Verbatim MD 577 Super Minidisk

In the past Commodore has frowned on the use of Verbatim diskettes for the 2040 floppy disk, particularly the MD

525-16. Verbatim recognized the problems with their disks and have improved the quality substantially. Result: The MD 577 Super Mini.

First, the thickness of the jacket PVC material has been increased from 7.5 to 8 mils giving the disks greater rigidity.

Secondly, the lamination pattern, which secures the inner lining to the jacket, was redesigned to eliminate potential "pillowing" problems. "Pillows" are minute raised areas on the lining surface which can interfere with the sideways movement of the disk.

Most importantly though, the new Verbatim MD 577s are provided with a factory installed "hard hole" or hub reinforcement ring, thus creating better centering ability and reducing the possibility of hub damage. Coincidentally, the performance of almost any diskette can be substantially improved by adding a hub ring prior to formatting.

Part of the problem was also the boxes they were packaged in, which put creases in the front two or three disks. These are no longer used.

We have tested the Verbatim 577s and found them to be of quite high quality. We've also decided to use them for distributing Commodore software which should appear on the market this fall.

Controlling Garbage Collections

Henry Troup,
Toronto, Ont.

We all know that the PET garbage collection can take an annoyingly long time. One highly frustrating time for a garbage collection to happen is while you are executing a GET loop input from the keyboard. There you are, typing away, and suddenly the cursor is still flashing at you, but no inputs are accepted.

To avoid this, we'd like to force an early garbage collection, at the start of the input, but only if it would have happened anyway.

First things first. A GET loop is very productive of garbage collections because it uses lots of memory. The typical form of this loop is:

```
10 GET A$: IF A$ = "" THEN 10
20 B$=B$+A$
```

What this does is create a set of partial strings. If the input is 'Mary had a little lamb', then the strings are:

```
M
Ma
Mar
Mary
and so on to
Mary had a little lam
Mary had a little lamb
```

That's a lot. Exactly how much ? We could count the number of characters and sum the numbers from 1 to n, but a rule of thumb is n squared over 2. (A more exact figure is (n squared + n)/2) For 22 characters, the memory used is 242 bytes. For 80 characters, it's around 3240 bytes.

So, what can we do about it. Well, we need some way of determining the free memory space. FRE(0) will do this - but it will cause a garbage collection, and we don't really want one yet. Let's define a function, FNFR(X):

```
1 DEF FNFR(X) = PEEK(48) + 256*PEEK(49) - (PEEK(46) +
256*PEEK(47))
```

That's simply the distance between the beginning of strings and the end of arrays. The argument is a dummy, just like FRE(X).

Our test then is:

```
5 IF FNFR(X) < (L*L)/2 THEN Q = FRE(0)
```

where L is the anticipated maximum string length.

One peculiarity of FNFR is that the statement:

PRINT FNFR(0)-FRE(0)   is almost never the same as:

PRINT FRE(0)-FNFR(0)   which is always 0.

True ASCII Output

Henry Troup,
Toronto, Ont.

We are all aware that the PET does not use true ASCII coding internallly. However, many of us have printers that do use real ASCII. In order to get upper and lower case operation, some code conversion is needed.

In this article, I shall present two ways of doing the conversion: one in BASIC, and one machine language. Both operate by a table lookup. This has the advantage that any other code conversion (to screen poke, Baudot or teletype code, for example, or ISO, or EIA, or what have you) can be had simply by changing the table. Or, a simple conversion to lower case can be had by ANDing each byte with 127.

I personally keep the conversion table in a disk file. It is appended at the end of this article.

First, the BASIC method. We dimension an integer array, M%(255), and use it as the table. Then we assign the string to be converted to S$.

```
1000 REM CONVERSION ROUTINE
1010 M$="" : IF S$= ""  THEN 1050
1020 FOR I = 1 TO LEN(S$)
1030 M$ = M$ + CHR$ (M%(ASC(MID$(S$,I))))
1040 NEXT I
1050 RETURN
```

This is slow, but tolerable if you're not doing too much conversion. It uses 519 bytes for storage of the table, and needs an available space of about five times the length of the string for working storage (it will work with less, but garbage collections will cause delays).

Now, the machine language method. This is faster and uses less storage. Here is the assembler listing. This program operates on the variable after the SYS. You must set up the table (anywhere you can get 256 bytes of free memory), and move the BASIC pointers. Then you can call the program.

```
                          ;convert2.src
                          ;convert petascii to true
                          ;ascii by lookup
           sl = $dd       ;a convenient place to put
                          ;the pointer (used in tape i/o)
           ts = $7f00     ;start of table
           va = $44
.skip
            * = 826
.skip
           lda sl
           pha
           lda sl+1
           pha
           jsr $cdf8      ;check comma
           jsr $cf6d      ;find variable
           lda $07        ;check type
           bne start
           jmp $cc9a      ;type mismatch error if numeric
.skip
 start     cpx #$00       ;check for null string
                          ;or undefined variable
           beq null
           ldy #$02
           lda (va),y     ;ptr lo
           sta sl+1
           dey
           lda (va),y     ;ptr hi
           sta sl
           dey
           lda (va),y     ;length
           tay
           beq null
           dey
 loop2     lda (sl),y
                          ;any character handling routine
                          ;can be substituted for the
                          ;next lines
           tax
           lda ts,x       ;do table lookup
           sta (sl),y     ;put back in string
           dey
           cpy #$ff       ;test for end
           bne loop2
 null      pla            ;restore zero page
           sta sl+1
           pla
           sta sl
           rts
.end
 ;to use this routine:
 ;sys 826,(string variable)
 ;the  converted  string  ]Lis  returned  into  the  original
space
 ;note:  if  the  variable  is  defined  in  text,  it  will  be
changed in text !
 ;string array variables work, except for the 0th element
 ;undefined variables are taken as nulls.
 ;undimmed arrays will be created
```

And as a basic loader: (which locates the table from the top of memory pointer)

```
10 DATA 165, 221,  72, 165, 222,  72
15 DATA  32, 248, 205,  32, 109, 207
20 DATA 165,   7, 208,,   3,  76, 154
25 DATA 204, 224,   0, 240,  31, 160
30 DATA   2, 177,  68, 133, 222, 136
35 DATA 177,  68, 133, 221, 136, 177
40 DATA  68, 168, 240,  14, 136, 177
45 DATA 221, 170, 189,  -1,  -2, 145
50 DATA 221, 136, 192, 255, 208, 243
60 DATA 104, 133, 222, 104

1000 FOR X = 826 TO 914:READ P
1010 IFP = -1 THEN P = PEEK(54):REM RELOCATE TABLE
1020 IFP = -2 THEN P = PEEK(53)
1030 POKE X,P :NEXTX
```

A Sample Initialization:

```
10 POKE53,PEEK(53-1):CLR:REM MOVE TOP OF MEMORY
20 OPEN4,4:GOSUB1000:REM GET PROGRAM
40 OPEN5,8,5,"CONVERT,S,R":REMM GET TABLE FROM DISK
50 FORX=0TO255:INPUT#5,M%:POKEPEEK(53)+X,M%:NEXTX:CLOSE5:REM
PUT TABLE IN
60  S$="THIS  IS  A  TEST":SYS826,S$:PRINT#4,S$:REM  ACTUAL
CONVERSION
```

This is much faster, and needs only the 256 bytes to store the table. The conversion table follows:

```
1000 data   0,   1,   2,   3,   4,   5,   6,   7,   8,   9
1010 data  10,  11,  12,  13,  14,  15
1020 data  16,  17,  18,  19,  20,  21,  22,  23,  24,  25
1030 data  26,  27,  28,  29,  30,  31,  32,  33,  34,  35
1040 data  36,  37,  38,  39,  40,  41,  42,  43,  44,  45
1050 data  46,  47,  48,  49,  50,  51,  52,  53,  54,  55
1060 data  56,  57,  58,  59,  60,  61,  62,  63,  64,  97
1070 data  98 ,  99, 100, 101, 102, 103, 104, 105, 106, 107
1080 data 108, 109, 110, 111, 112, 113, 114, 115, 116, 117
1090 data 118, 119, 120, 121, 122,  91,  92,  93,  94,  95
1100 data  96,  97,  98,  99, 100, 101, 102, 103, 104, 105
1110 data 106, 107, 108, 109, 110, 111, 112, 113, 114, 115
1120 data 116, 117, 118, 119, 120, 121, 122, 123, 124, 125
1130 data 126, 127, 128, 129, 130, 131, 132, 133, 134, 135
1140 data 136, 137, 138, 139, 140, 141, 142, 143, 144, 145
1150 data 146, 147, 148, 149, 150, 151, 152, 153, 154, 155
1160 data 156, 157, 158, 159, 160, 161, 162, 163, 164, 165
1170 data 166, 167, 168, 169, 170, 171, 172, 173, 174, 175
1180 data 176, 177, 178, 179, 180, 181, 182, 183, 184, 185
1190 data 186, 187, 188, 189, 190, 191, 192,  65   66,  67
1200 data  68,  69,  70,  71,  72,  73,  74,  75,  76   77
1210 data  78,  79,  80,  81,  82,  83,  84,  85,  86,  87
1220 data  88,  89,  90, 219, 220, 221, 222, 223, 224, 225
1230 data 226, 227, 228, 229, 230, 231, 232, 233, 234, 235
1240 data 236, 237, 238, 239, 240, 241, 242, 243, 244, 245
1250 data 246, 247, 248, 249, 250, 251, 252, 253, 254, 255
```

PET 2040 DISK BUFFER I/O ROUTINE

J.HOOGSTRAAT,
BOX 20. SITE 7. SS 1,
CALGARY. ALTA.

The major difficulty in programming direct access routines for the PET 2040 disk drives is the computation of the exact location of the recorded information on a disk sector, for the reason that the PET prints its data to the disk rather than transferring it byte for byte.

This results in variable length records on each disk write, unless the programmer takes special care converting each variable to a fixed length string variable before writing it to the disk. This is not too bad for string variables, but other variables could be ranging in length from one to more than ten characters after conversion to an equivalent string variable.

Suppose we want to program a direct access file consisting of records made up of an ITEM-NO, DESCRIPTION and COST.

The ITEM-NO ranges from 1 to 9999
The DESCRIPTION is 12 bytes long
The COST ranges from .00 to 9999999.00

We need 4 characters for the ITEM-NO, 12 for the DESCRIPTION and 10 for the COST. This would total up to 26 characters per record, but in order to be able to read it back we have to add at least one carriage return character after the COST string. After reading we can de-compose the information with MID$ calls. Or, if we wish to be able to update each field individually, a carriage return character must be added after each field, which ups our total record length to 29 characters

I personally found this method rather wasteful and cumbersome to program with all the STR$ calls and BLANK padding. No other software seemed to be available, except for Bill Macleans Block Get Routine published in the Commodore Transactor Vol 2, Dec 31, 1979. An excellent routine, but it can only read from the disk buffers with special care to be taken for the allocation of the input string variable.

So, what I needed was a routine with the following characteristics:

.. Be able to read the disk block buffers.

.. Be able to write the disk block buffers.

.. No need for blank padding of any variables or the need of adding carriage return characters.

.. Record and read numeric variables as 5 binary characters, as stored in PET's memory. This allows records of up to 51 numeric variables on a disk sector.

.. Be able to read single character string variables with an

ASC value of zero, in stead of getting a NULL string.

.. Exercise full control over the Block Buffer Pointers.

.. Perform like a basic WRITE or READ statement.

.. No need for special declarations or dummy manipulations of input variables.

.. Be able to output any kind of proper expressions.

.. Be totally relocatable.

Aided with Jim Butterfields excellent PET maps and the Macro-Tea assembler of Skyles Electric Works, I succesfully coded the needed routine.

I'll explain how to use it with some basic coding examples.

The basic format for the call to the PET 2040 disk buffer I/O routine is:

```
SYS XX, IO, CH, ( BP ,VA ,(LN))
-------------------------------
```

XX = Address were the routine is loaded.

IO = Input / Output key value.

CH = Disk direct access channel no.

BP = Buffer pointer value.

VA = Variable name.

LN = No of characters.

For single BP control the IO values are:

0 For normal reading.
1 For normal writing.
2 For special reading.
3 Same as 1.

For multiple BP control the IO values are:

4 For normal reading.
5 For normal writing.
6 For special reading.
7 Same as 5

BASIC NUMERIC VARIABLE EXAMPLES

```
10 DK = 1: CE = 15: CH = 2: XX = 634
20 OPEN CE,8,CE
30 OPEN CH,8,CH,"#"

40 T = 2: S = 5: BP = 13

50 REM WRITE 3 VARIABLES  TO  DISK
   -------------------------------

60 SYS XX, 1, CH, BP, A, B, C :REM OUTPUT

70 PRINT#CE, "U2:"CH;DK;T;S

880 REM READ 3 VARIABLES FROM DISK
   ------------------------------

90 PRINT#CE, "U2:"CH;DK;T;S

100 SYS XX, 0, CH, BP, X, Y, Z :REM INPUT
```

In this example we are writing the 3 numeric variables (A,B,C) to the disk buffer starting at character position 13. The result is then written to disk drive 1 at Track 2, Sector 5. The buffer pointer is automatically incremented by 5 for each variable and the variables are recorded in internal PET format. Note no padding or carriage returns needed. After the write, the variables are read back into X, Y and Z.

For numeric variables the parameter LN is implied and must not be coded.

If the PRINT#CE calls were omitted, no actual disk writing or reading would take place, but merely a transfer to and from the disk buffer allocated to channel CH, which maybe useful in passing parameters between overlays.

Statement 60 could be something like

```
60 SYS XX, 1, CH, BP,      1.,          A, A+B*C :REM OUTPUT
```

or

```
60 SYS XX, 1, CH, BP, SQR(A), SIN(A+B),     A/B :REM OUTPUT
```

or

```
60 SYS XX, 1, CH, BP,    1.+C,         -A, -55.5 :REM OUTPUT
```

The number of concatenated variables is only limited by the maximum length of a BASIC line. But at least one must be specified. We could also replace statement 60 by the following lines:

```
60 SYS XX, 1, CH, BP   , A  :REM OUTPUT
61 SYS XX, 1, CH, BP+ 5, B  :REM OUTPUT
62 SYS XX, 1, CH, BP+10, C  :REM OUTPUT
```

Which have the same effect as the original line 60.

Statement 100 could also be replaced by the following lines, which would read back the exact same information in the variables X, Y and Z.

```
   100 SYS XX, 0, CH, BP+ 5, Y, Z :REM INPUT
   101 SYS XX, 0, CH, BP    , X    :REM INPUT
```

If we want more control over the buffer pointer on the write, the value for IO must be 4 for reading and 5 for writing.

Statements 60 and 100 which were:

```
    60 SYS XX, 1, CH, BP, A, B, C :REM OUTPUT

   100 SYS XX, 0, CH, BP, X, Y,  Z :REM INPUT
```

can now be coded as:

```
    60 SYS XX, 5, CH, BP, A, BP+ 5, B, BP+10, C :REM OUTPUT

   100 SYS XX, 4, CH, BP, X, BP+ 5, Y, BP+10, Z :REM INPUT
```

The difference is that each variable now has a buffer pointer value preceeding it. The statements can now also be:

```
    60 SYS XX, 5, CH, BP+ 5, B, BP, A, BP+10, C :REM OUTPUT

   100 SYS XX, 4, CH, BP+10, Z, BP, X, BP+ 5, Y :REM INPUT
```

Since we now have full buffer pointer control.

BASIC STRING VARIABLES EXAMPLES

```
    10 DK = 1: CE = 15: CH = 2: XX = 634
    20 OPEN CE,8,CE
    30 OPEN CH,8,CH,"#"

    40 T = 2: S = 5: BP = 13

    50 REM WRITE 3 STRING VARIABLES  TO  DISK
       -------------------------------------
    60 SYS XX, 1, CH, BP, A$,5, B$,6, C$,10 :REM OUTPUT
    70 PRINT#CE, "U2:"CH;DK;T;S

    80 REM READ  3 STRING VARIABLES FROM DISK
       -------------------------------------
    90 PRINT#CE, "U2:"CH;DK;T;S

   100 SYS XX, 0, CH, BP, X$,5, Y$,6, Z$,10 :REM INPUT
```

In this example we are writing the 3 STRING variables (A$,B$,C$) to the disk buffer starting at character position 13. The result is then written to disk drive 1 at Track 2, Sector 5.

The difference between a numeric variable and a string variable is that the string variable is followed by LN, its length or number of characters. The specied length does not have to be the actual length of the string variable. In our example the first 5 characters of X$ are transferred, followed by the first 6 characters of Y$ and then the first 10 characters of Z$.

The buffer pointer is automatically incremented by 5,6 and 10. Note no padding or carriage returns needed. After the write, the variables are read back into the string$ X$, Y$ and Z$

Lets now examine what happens if we have the following statements:

```
55 Z$ = "HANS"+"MARGARET"
```

```
60 SYS XX, 1, CH, BP, Z$,LEN(Z$) :REM OUTPUT
```

The disk buffer (CH) will now contain starting at character position 13 the text "HANSMARGARET". The same results of the next statement:

```
60 SYS XX, 1, CH, BP, "HANS"+"MARGARET",12 :REM OUTPUT
```

And the statement:

```
100 SYS XX, 0, CH, BP, Z$,12 :REM INPUT
```

Will input and create a string variable with a length of 12 characters and containing the text "HANSMARGARET". However the statement:

```
100 SYS XX, 0, CH, BP, Z$,10 :REM INPUT
```

Will input and create a string variable with a length of 10 characters and containing the text "HANSMARGAR". Or the statements:

```
100 SYS XX, 0, CH, BP   , X$,6 :REM INPUT
101 SYS XX, 0, CH, BP+7, Z$,5 :REM INPUT
```

Will input and create two string variables X$ and Z$, containing "HANSMA" AND "GARET"

Note that no extra linefeeds or carriage return characters are written and that the record space needed for the original ITEM-NO, DESCRIPTION and COST example is now 5+12+5 or 22 characters instead of the 29 needed without this buffer I/O routine.

If the PRINT#CE calls were omitted no actual disk writing or reading would take place, but merely a transfer to and from the disk buffer allocated to channel CH, which again maybe useful in passing parameters between overlays, or to do some fancy string manipulations.

P.E.:

```
10 A$ = "XXXXXXXXXX"
11 B$ = "YYYYY"
12 SYS XX, 1, CH, 2, A$, LEN(A$) :REM OUTPUT
13 SYS XX, 1, CH, 5, B$, LEN(B$) :REM OUTPUT
14 SYSS XX, 0, CH, 2, A$, 10      :REM INPUT
```

First writes the string variables A$ and B$ overlaying the A$ information and then inputs and creates a string variable A$ containing "XXXYYYYYXX".

Statement 60 could be something like

```
60  SYS  XX,  1,  CH,  BP,  A$+"X",5,  A$+B$,6,  A$+"Z"+C$,10
:REM OUTPUT
```

The number of concatenated string variables is only limited by the maximum length of a BASIC line. But at least one must be specified. We could also replace statement 60 by the following lines:

```
60 SYS XX, 1, CH, BP    , A$,5   :REM OUTPUT
61 SYS XX, 1, CH, BP+ 5, B$,6   :REM OUTPUT
62 SYS XX, 1, CH, BP+11, C$,10  :REM OUTPUT
```

Which have the same effect as the original line 60.

Statement 100 could also be replaced by the following lines, which would read back the exact same information in the string variables X$, Y$ and Z$

```
100 SYS XX, 0, CH, BP+5, Y$,6, Z$,10 :REM INPUT
101 SYS XX, 0, CH, BP   , X$,5        :REM INPUT
```

If we want more control over the buffer pointer on the write, the value for IO must be 4 for reading and 5 for writing.

Statements 60 and 100 which were:

```
 60 SYS XX, 1, CH, BP, A$,5, B$,6, C$,10 :REM OUTPUT
100 SYS XX, 0, CH, BP, X$,5, Y$,6, Z$,10 :REM INPUT
```

can now be coded as:

```
60  SYS  XX,  5,  CH,  BP,A$,5,  BP+5,B$,6,  BP+11,C$,10  :REM
OUTPUT

100  SYS  XX,  4,  CH,  BP,X$,5,  BP+5,Y$,6,  BP+11,Z$,10  :REM
INPUT
```

The difference is that each string variable now has a buffer pointer value preceeding it and still its length following it. The statements can now also be:

```
60  SYS  XX,  5,  CH,  BP+5,B$,6,  BP+11,C$,10,  BP,A$,5  :REM
OUTPUT

100  SYS  XX,  4,  CH,  BP+11,Z$,10,  BP,A$,5,  BP+5,Y$,6  :REM
INPUT
```

Since we now have full buffer pointer control.

So far I only discussed write and reads of string variables of the same length on the writing and reading.

Now suppose we have the following statements:

```
55 A$ = "HANS"
60 SYS XX, 5, CH, 10,A$,10 , 20,A$+A$,10 :REM OUTPUT
```

This transfers to the buffer, starting at character location 10, the characters "hans*****hanshans**", where the "*" stands for an automatic padded carriage return character with an ASC value of 13. In other words the routine will always write the number of characters requested but if the output string expression is too short, the output will be padded with carriage return characters. This has a nice effect when we read the same data back with the following statement:

```
100 SYS XX, 4, CH, 10,A$,10 , 20,B$,10 :REM INPUT
```

This call will input and create the two string variables A$ and B$, but their contents will be "HANS" AND "HANSHANS", since the input quits on the first encountered carriage return characters for each variable and their length will be 4 and 8. However an otherwise null character string will always be returned as a character string of ASC value zero with a length of one.

Sometimes this technique is undesirable and we want to get back every character, no matter what their ASC values are. Now the special read I/O values 2 or 6 are to be used. The statement:

```
100 SYS XX, 6, CH, 10,A$,10 , 20,B$,10 :REM INPUT
```

Will now input and create an A$ and B$ variable containing "hans******" and "hanshans**".

Note, the length limit of a string variable is 255 bytes, allowing us to read or write entire disk buffer blocks at once.

By no means do we have to write separate statements for numeric or string variables, we can mix them up. The following statements are quite legal:

```
 51 IT  = 5469
 52 SS$ = "PET COMPUTER"
 53 CO  = 1365.25
 60 SYS XX, 1, CH, 2, IT, SS$,12, CO  :REM OUTPUT
100 SYS XX, 6, CH, 7,A$,12 ,2,A, 19,B :REM INPUT
```

Again the read call for I/O = 6 will properly return:

```
A$     = "PET COMPUTER", A = 5469, B = 1365.25
```

Still confused, please contact me !

```
               0010; ROUTINE TO TRANSFER FLOATING POINT VARIABLES AND STRING
               0020; VARIABLES BETWEEN PET'S MEMORY AND A D/A DISK BUFFER.
               0030; ------------------------------------------------------------
               0040;
               0050; WRITTEN BY J.HOOGSTRAAT
               0060;
               0070;            BOX-20,   SITE 7,   SS1
               0080;            CALGARY,  T2M-4N3,  ALTA
               0090;            PHONE     (403)239-0900
               0100;
               0110; ------------------------------------------------------------
               0120;
               0130; THIS ROUTINE IS TOTAL RELOCATABLE AND CAN BE LOADED ANYWHERE.
               0140;
               0150; FLOATING POINT VARIABLES ARE TRANSFERRED AS 5 BYTES ONLY.
               0160;
               0170; STRING VARIABLES ARE TRANSFERRED WITHOUT LINEFEEDS
               0180; OR CARRIAGE RETURNS.
               0190;
               0200; THIS ROUTINE IS IDEALLY SUITABLE FOR DIRECT DISK ACCESSING,
               0210; SINCE ALL BUFFER POINTERS CAN BE CALCULATED EXACTLY.
               0220;
               0230; ------------------------------------------------------------
               0240;
               0250;
               0260           .OS
               0270           .BA 634       ;FIRST CASSETTE BUFFER FOR NOW.
               0280;
               0290; LOCAL VARIABLES
               0300;
               0310STADR      .DI $1        ;SAVED ROUTINE START ADDRESS.
               0320SYSXX      .DI $11       ;BASIC ROUTINE START ADDRESS AS SYS XX.
               0330;
               0340IO         .DI $B1       ;SAVED IO.
               0350DCH        .DI $B2       ;SAVED DCH.
               0360LNG        .DI $B7       ;SAVED REQ. LENGTH.
               0370STP         .DI $B8       ;SAVED DATA TYPE.
               0380;
               0390; LOCAL VALUES
               0400;
               0410DCE        .DI $F        ;DISK COMMAND CHANNEL.
               0420CRT        .DI $D        ;CARRIAGE RETURN.
               0430FLN        .DI $5        ;FLT PNT WORD LENGTH.
               0440;
               0450; BASIC AREAS USED
               0460;
               0470DTP        .DI $07       ;DATA TYPE.
               0480SLN        .DI $16        ;STRING LENGTH.
               0490SAD        .DI $17       ;STRING ADDRESS.
               0500CAD        .DI $44       ;CURRENT VARIABLE ADDR.
               0510ACC        .DI $5E       ;ACCUMULATOR.
               0520NCH        .DI $77       ;NEXT INPUT FIELD CHAR.
               0530ASB        .DI $100      ;ASC BUFFER.
               0540;
027A-A511      0550START      LDA *SYSXX    ;START START ADDR
027C-8501      0560           STA *STADR    ;FOR SELF RELOCATION.
027E-A512      0570           LDA *SYSXX+1
0280-8502      0580           STA *STADR+1
               0590;
0282-20F8CD    0600           JSR CHKCOM    ;UPTO NEXT FIELD.
```

```
0285-209FCC 0610           JSR EVAEXP   ;EVALUATE EXPRESSION.
0288-20D2D6 0620           JSR FLTFIX   ;CONVERT TO INTEGER.
028B-84B1   0630           STY *IO      ;SAVE IO.
            0640;
028D-20F8CD 0650           JSR CHKCOM   ;UPTO NEXT FIELD.
0290-209FCC 0660           JSR EVAEXP   ;EVALUATE EXPRESSION.
0293-20D2D6 0670           JSR FLTFIX   ;CONVERT TO INTEGER.
0296-84B2   0680           STY *DCH     ;SAVE DCH.
            0690;
0298-20F8CD 0700AGAIN      JSR CHKCOM   ;UPTO NEXT FIELD.
029B-209FCC 0710           JSR EVAEXP   ;EVALUATE EXPRESSSION.
029E-20E9DC 0720           JSR BINASC   ;CVT BPT TO ASC.
            0730;
            0740; ISSUE PRINT#CE, "B-P:"CH;BP
            0750; --------------------------
            0760;
02A1-A20F   0770           LDX #DCE            ;OPEN CHANNEL 'CE'.
02A3-20C9FF 0780           JSR STODEV
            0790;
02A6-A0C4   0800           LDY #BPDCH-START    ;SET RELOCATION.
            0810;
02A8-A5B2   0820           LDA *DCH            ;STOW ASC OF DCH
02AA-0930   0830           ORA #$30            ;IN THE TEXT.
02AC-9101   0840           STA (STADDR),Y
            0850;
02AE-A0C1   0860           LDY #BPTXT-START    ;SET RELOCATION.
02B0-B101   0870OUTBP      LDA (STADR),Y       ;OUTPUT "B-P:"CH.
02B2-20D2FF 0880           JSR OUTCHR
02B5-C8     0890           INY
02B6-C0C6   0900           CPY #BPTXE-START    ;END OF TEXT ?
02B8-D0F6   0910           BNE OUTBP           ;NO, CONTINUE.
            0920;
02BA-A201   0930           LDX #1
02BC-BD0001 0940BPOUT      LDA ASB,X           ;OUTPUT ASC OF BP
02BF-F00A   0950           BEQ BPDON           ;END OF ASC.
02C1-20D2FF 0960           JSR OUTCHR
02C4-E8     0970           INX
02C5-D0F5   0980           BNE BPOUT           ;CONTINUE TILL END.
02C7-F002   0990           BEQ BPDON
            1000;
02C9-D0CD   1010AGAJJ      BNE AGAIN
            1020;
02CB-20CCFF 1030BPDON      JSR RESTIO
            1040;
            1050; ISSUE PRINT#CH FOR INPUT OR OUTPUT
            1060; ---------------------------------
            1070;
02CE-A6B2   1080           LDX *DCH
            1090;
02D0-A5B1   1100           LDA *IO      ;CHECK IO.
02D2-2901   1110           AND #1
02D4-F005   1120           BEQ OPINP    ;INPUT.
            1130;
02D6-20C9FF 1140OPOUT      JSR STODEV   ;OPEN OUTPUT CH.
02D9-D003   1150           BNE TRFER
            1160;
02DB-20C6FF 1170OPINP      JSR STIDEV   ;OPEN INPUT CH.
            1180;
02DE-20F8CD 1190TRFER      JSR CHKCOM   ;UPTO NEXT FIELD.
            1200;
```

```
02E1-A905    1210            LDA #FLN     ;DEFAULT LENGTH
02E3-85B7    1220            STA *LNG     ;TO FLT PNT LENGTH.
02E5-8516    1230            STA *SLN
             1240;
02E7-A5B1    1250            LDA *IO      ;CHECK IO.
02E9-2901    1260            AND #1
02EB-F053    1270            BEQ RINPT    ;READ INPUT.
             1280;
             1290; WRITE OUTPUT DATA
             1300; -----------------
             1310;
02ED-209FCC  1320WOUTP       JSR EVAEXP   ;EVALUATE EXPRESSION.
02F0-08      1330            PHP          ;SAVE STATUS
             1340;
02F1-A507    1350            LDA *DTP     ;CHARACTER STRING ?
02F3-F01D    1360             BEQ FLTDT    ;NO FLT PNT VARIABLE.
             1370;
             1380; OUTPUT STRING EXPRESSION
             1390;
02F5-207DD5  1400            JSR DSCSTR   ;DISCARD TEMP STRING
             1410;
02F8-28      1420            PLP          ;GET STATUS
02F9-100A    1430            BPL WOUTS    ;NOT A CONTANT STRING
             1440;
02FB-A002    1450WOUTC       LDY #2       ;SAVE STRING ADDRESS
02FD-B144    1460STRAD       LDA (CAD),Y
02FF-991600  1470            STA SLN,Y
0302-88      1480            DEY
0303-10F8    1490            BPL STRAD
             1500;
0305-20F8CD  1510WOUTS       JSR CHKCOM   ;UPTO NEXT FIELD.
0308-209FCC  1520            JSR EVAEXP   ;EVALUATE EXPRESSION.
030B-20D2D6  1530            JSR FLTFIX   ;CONVERT TO INTEGER.
030E-84B7    1540            STY *LNG     ;SAVE REQ. LENGTH.
0310-D011    1550            BNE WRITE    ;READY FOR OUTPUT.
             1560;
             1570; OUTPUT FLT PNT DATA IN ACCUMULATOR
             1580;
0312-28      1590FLTDT       PLP          ;CLEAR STACK
             1600;
0313-A563    1610            LDA *ACC+5   ;CORRECT SIGN ?
0315-3004    1620            BMI FLTCR    ;NO.
             1630;
0317-065F    1640            ASL *ACC+1   ;REMOVE SIGN BIT
0319-465F    1650            LSR *ACC+1   ;FROM ACCUMULATOR.
             1660;
031B-A95E    1670FLTCR       LDA #L,ACC   ;SET OUTPUT
031D-A000    1680            LDY #H,ACC   ;ADDRESS TO THE
031F-8517    1690            STA *SAD     ;ACCUMLATOR.
0321-8418    1700            STY *SAD+1
             1710;
             1720; OUTPUT CHARACTER LOOP
             1730;
0323-A000    1740WRITE       LDY #0       ;SET CHAR POINTER.
             1750;
0325-A90D    1760WRIT1       LDA #CRT     ;DEFAULT TO CR.
0327-C416    1770            CPY *SLN     ;MORE THAN ACTUAL LENGTH ?
0329-B002    1780            BCS WRIT2    ;YES, USE CR.
032B-B117    1790            LDA (SAD),Y  ;USE INPUT CHAR.
032D-20D2FF  1800WRIT2       JSR OUTCHR   ;OUTPUT THIS CHAR.
```

```
              1810;
0330-C8       1820            INY
0331-C4B7     1830            CPY *LNG     ;ALL DONE ?
0333-D0F0     1840            BNE WRIT1    ;NO.
0335-F061     1850            BEQ FIELD    ;YES.
              1860;
              1870; INBETWEEN JUMP AND CONSTANTS
              1880; ----------------------------
              1890;
0337-D090     1900AGAIJ       BNE AGAJJ
0339-F0A3     1910TRFEJ       BEQ TRFER
              1920;
033B-422D50   1930BPTXT       .BY 'B-P'
033E-5820     1940BPDCH       .BY 'X '
              1950BPTXE       .DI =
              1960;
              1970; READ INPUT DATA
              1980; ---------------
              1990;
0340-206DCF   2000RINPT       JSR GETVAR   ;GET VARIABLE ADDR.
              2010;
0343-8517     2020            STA *SAD     ;DEFAULT INPUT ADDRESS.
0345-8418     2030            STY *SAD+1   ;TO FLT PNT VARIABLE
              2040;
0347-A507     2050            LDA *DTP     ;SAVE AND CHECK DATA TYPE.
0349-85B8     2060            STA *STP
              2070;
              2080; INPUT FLT PNT VARIABLE
              2090;
034B-F020     2100            BEQ READI    ;FLT PNT INPUT VARIABLE.
              2110;
              2120; INPUT STRING VARIABLE
              2130;
034D-20F8CD   2140            JSR CHKCOM   ;UPTO NEXT FIELD.
0350-209FCC   2150            JSR EVAEXP   ;EVALUATE EXPRESSION.
0353-20D2D6   2160            JSR FLTFIX   ;CONVERT TO INTEGER.
              2170;
0356-98       2180            TYA
0357-A000     2190            LDY #0
0359-8516     2200            STA *SLN     ;SAVE REQ. LLENGTH.
035B-9117     2210            STA (SAD),Y  ;SAVE IN STRING INDEX.
              2220;
035D-20D0D3   2230            JSR GETSPC   ;GET SPACE FOR STRING.
              2240;
0360-98       2250            TYA
0361-A002     2260            LDY #2       ;SAVE ADDRESS OF SPACE
0363-9117     2270            STA (SAD),Y  ;IN STRING INDEX
0365-8545     2280            STA *CAD+1   ;AND CURRENT VARIABLE ADDRESS.
0367-8A       2290            TXA
0368-88       2300            DEY
0369-9117     2310            STA (SAD),Y
036B-8544     2320            STA *CAD
              2330;
036D-A000     2340READI       LDY #0       ;SET CHAR POINTER.
              2350;
036F-A5B1     2360            LDA *IO      ;CHECK IO.
0371-2902     2370            AND #2       ;SPECIAL STRING READ ?
0373-F002     2380            BEQ READ1    ;NO.
              2390;
0375-84B8     2400            STY *STP     ;CHANGE FROM 'FF' TO '00'.
```

```
              2410;
0377-20CFFF   2420READ1       JSR INPCHR     ;INPUT A CHAR.
              2430;
037A-C90D     2440            CMP #CRT       ;CARRIAGE RETURN ?
037C-D004     2450            BNE READ2      ;NO.
              2460;
037E-A6B8     2470            LDX *STP       ;YES. STRING ?
0380-D009     2480            BNE READ4      ;YES. TERMINATE STRING.
              2490;
0382-9144     2500READ2       STA (CAD),Y    ;STOW CHAR INTO INPUT.
              2510;
0384-C8       2520READ3       INY
0385-C416     2530            CPY *SLN       ;ALL DONE ?
0387-D0EE     2540            BNE READ1      ;NO.
0389-F00D     2550            BEQ FIELD      ;YES.
              2560;
038B-98       2570READ4       TYA
038C-F0F4     2580            BEQ READ2      ;INTERCEPT NULL STRINGS
              2590;
038E-A000     2600            LDY #0         ;SET RECORDED STRING LENGTH.
0390-84B8     2610            STY *STP       ;RESET DATA TYPE.
0392-9117     2620            STA (SAD),Y    ;TRUNCATE STRING IN INDEX.
0394-A8       2630            TAY
0395-18       2640            CLC
0396-90EC     2650            BCC READ3      ;CONTINUE READING.
              2660;
              2670; CHECK FOR MORE FIELDS
              2680; ---------------------
              2690;
0398-A000     2700FIELD       LDY #0         ;MORE FIELDS ARE PRESENT
039A-B177     2710            LDA (NCH),Y    ;IF THERE IS A COMMA IN
039C-C92C     2720            CMP #',        ;BASIC'S INPUT BUFFER.
039E-D008     2730            BNE ADONE      ;NO, WE QUIT.
03A0-A5B1     2740            LDA *IO        ;WHAT KIND
03A2-290C     2750            AND #12
03A4-F093     2760            BEQ TRFEJ      ;GO AGAIN, NO BP
03A6-D08F     2770            BNE AGAIJ      ;GO AGAIN, BP SET
              2780;
              2790; TERMINATE ROUTINE
              2800; -----------------
              2810;
03A8-20CCFF   2820ADONE       JSR RESTIO     ;RESTORE I/O DEVICE.
03AB-60       2830            RTS
              2840;
              2850; BASIC ROUTINES USED
              2860;
              2870EVAEXP      .DE $CC9F      ;EVALUATE EXPRESSION.
              2880CHKCOM      .DE $CDF8      ;CHECK FOR COMMA.
              2890GETVAR      .DE $CF6D      ;GET BASIC VARIABLE.
              2900GETSPC      .DE $D3D0      ;GET STRING SPACE.
              2910DSCSTR      .DE $D57D      ;DISCARD TEMP STRING.
              2920FLTFIX      .DE $D6D2      ;FLOAT TO INTEGER. CONVERSION
              2930BINASC      .DE $DCE9      ;CONVERT FLT TO ASC.
              2940RESTIO      .DE $FFCC      ;RESTORE DEFAULT I/O ADDRESSES.
              2950STIDEV      .DE $FFC6      ;SET INPUT DEVICE.
              2960STODEV      .DE $FFC9      ;SET OUTPUTT DEVICE.
              2970INPCHR      .DE $FFCF      ;INPUT CHARACTER.
              2980OUTCHR      .DE $FFD2      ;OUTPUT CHARACTER.
              2990            .EN
```

```
                LABELS
                ------

STADR    = 0001         SYSXX    = 0011
IO       = 00B1         DCH      = 00B2
LNG      = 00B7         STP      = 00B8
DCE      = 000F         CRT      = 000D
FLN      = 0005         DTP      = 0007
SLN      = 0016         SAD      = 0017
CAD      = 0044         ACC      = 005E
NCH      = 0077         ASB      = 0100
START    = 027A         AGAIN    = 0298
OUTBP    = 02B0         BPOUT    = 02BC
AGAJJ    = 02C9         BPDON    = 02CB
OPOUT    = 02D6         OPINP    = 02DB
TRFER    = 02DE         WOUTP    = 02ED
WOUTC    = 02FB         STRAD    = 02FD
WOUTS    = 0305         FLTDT    = 0312
FLTCR    = 031B         WRITE    = 0323
WRIT1    = 0325         WRIT2    = 032D
AGAIJ    = 0337         TRFEJ    = 0339
BPTXT    = 033B         BPDCH    = 033E
BPTXE    = 0340         RINPT    = 0340
READI    = 036D         READ1    = 0377
READ2    = 0382         READ3    = 0384
READ4    = 038B         FIELD    = 0398
ADONE    = 03A8         /EVAEXP = CC9F
/CHKCOM = CDF8          /GETVAR = CF6D
/GETSPC = D3D0          /DSCSTR = D57D
/FLTFIX = D6D2          /BINASC = DCE9
/RESTIO = FFCC          /STIDEV = FFC6
/STODEV = FFC9          /INPCHR = FFCF
/OUTCHR = FFD2
```

HEXADECIMAL DUMP

```
027A A5 11 85 01 A5 12 85 02
0282 20 F8 CD 20 9F CC 20 D2
028A D6 84 B1 20 F8 CD 20 9F
0292 CC 20 D2 D6 84 B2 20 F8
029A CD 20 9F CC 20 E9 DC A2
02A2 0F 20 C9 FF A0 C4 A5 B2
02AA 09 30 91 01 A0 C1 B1 01
02B2 20 D2 FF C8 C0 C6 D0 F6
02BA A2 01 BD 00 01 F0 0A 20
02C2 D2 FF E8 D0 F5 F0 02 D0
02CA CD 20 CC FF A6 B2 A5 B1
02D2 29 01 F0 05 20 C9 FF D0
02DA 03 20 C6 FF 20 F8 CD A9
02E2 05 85 B7 85 16 A5 B1 29
02EA 01 F0 53 20 9F CC 08 A5
02F2 07 F0 1D 20 7D D5 28 10
02FA 0A A0 02 B1 44 99 16 00
0302 88 10 F8 20 F8 CD 20 9F
030A CC 20 D2 D6 84 B7 D0 11
0312 28 A5 63 30 04 06 5F 46
031A 5F A9 5E A0 00 85 17 84
0322 18 A0 00 A9 0D C4 16 B0
032A 02 B1 17 20 D2 FF C8 C4
0332 B7 D0 F0 F0 61 D0 90 F0
033A A3 42 2D 50 33 20 20 6D
0342 CF 85 17 84 18 A5 07 85
034A B8 F0 20 20 F8 CD 20 9F
0352 CC 20 D2 D6 98 A0 00 85
035A 16 91 17 20 D0 D3 98 A0
0362 02 91 17 85 45 8A 88 91
036A 17 85 44 A0 00 A5 B1 29
0372 02 F0 02 84 B8 20 CF FF
037A C9 0D D0 04 A6 B8 D0 09
0382 91 44 C8 C4 16 D0 EE F0
038A 0D 98 F0 F4 A0 00 84 B8
0392 91 17 A8 18 90 EC A0 00
039A B1 77 C9 2C D0 08 A5 B1
03A2 29 0C F0 93 D0 8F 20 CC
03AA FF 60
```

```
60000 REM DATA STATEMENTS FOR D/A BUFFER ROUTINE
60001 REM
60002 REM TOTAL LENGTH 306 BYTES
60003 REM
60004 DATA 165,  17, 133,   1, 165,  18, 133,   2
60005 DATA  32, 248, 205,  32, 159, 204,  32, 210
60006 DATA 214, 132, 177,  32, 248, 205,  32, 159
60007 DATA 204,  32, 210, 214, 132, 178,  32, 248
60008 DATA 205,  32, 159, 204,  32, 233, 220, 162
60009 DATA  15,  32, 201, 255, 160, 196, 165, 178
60010 DATA   9,  48, 145,   1, 160, 193, 177,   1
60011 DATA  32, 210, 255, 200, 192, 198, 208, 246
60012 DATA 162,   1, 189,   0,   1, 240,  10,  32
60013 DATA 210, 255, 232, 208, 245, 240,   2, 208
60014 DATA 205,  32, 204, 255, 166, 178, 165, 177
60015 DATA  41,   1, 240,   5,  32, 201, 255, 208
60016 DATA   3,  32, 198, 255,  32, 248, 205, 169
60017 DATA   5, 133, 183, 133,  22, 165, 177,  41
60018 DATA   1, 240,  83,  32, 159, 204,   8, 165
60019 DATA   7, 240,  29,  32, 125, 213,  40,  16
60020 DATA  10, 160,   2, 177,  68, 153,  22,   0
60021 DATA 136,  16, 248,  32, 248, 205,  32, 159
60022 DATA 204,  32, 210, 214, 132, 183, 208,  17
60023 DATA  40, 165,  99,  48,   4,   6,  95,  70
60024 DATA  95, 169,  94, 160,   0, 133,  23, 132
60025 DATA  24, 160,   0, 169,  13, 196,  22, 176
60026 DATA   2, 177,  23,  32, 210, 255, 200, 196
60027 DATA 183, 208, 240, 240,  97, 208, 144, 240
60028 DATA 163,  66,  45,  80,  88,  32,  32, 109
60029 DATA 207, 133,  23, 132,  24, 165,   7, 133
60030 DATA 184, 240,  32,  32, 248, 205,  32, 159
60031 DATA 204,  32, 210, 214, 152, 160,   0, 133
60032 DATA  22, 145,  23,  32, 208, 211, 152, 160
60033 DATA   2, 145,  23, 133,  69, 138, 136, 145
60034 DATA  23, 133,  68, 160,   0, 165, 177,  41
60035 DATA   2, 240,   2, 132, 184,  32, 207, 255
60036 DATA 201,  13, 208,   4, 166, 184, 208,   9
60037 DATA 145,  68, 200, 196,  22, 208, 238, 240
60038 DATA  13, 152, 240, 244, 160,   0, 132, 184
60039 DATA 145,  23, 168,  24, 144, 236, 160,   0
60040 DATA 177, 119, 201,  44, 208,   8, 165, 177
60041 DATA  41,  12, 240, 147, 208, 143,  32, 204
60042 DATA 255,  96
60043 END
```

```
100 REM A RANDOM FILE DEMONSTRATION
110 REM WHICH NEEDS NO BLOCK-ALLOCATE
120  REM BY USING THE SPACE ALLLOCATED
130 REM OF ANY PREVIOUS CREATED FILE.
140 REM
150 REM THE RANDOM UPDATES CAN BE BITS
160 REM OF INFORMATION OF UPTO 254
170 REM BYTES OF STRING INFORMATION.
180 REM
190 REM FLOATING POINT VARIABLES ALWAYS
200 REM ARE ONLY 5 BYTES LONG. THE FIVE
210 REM BYTES PET USES.
220 REM
230 REM THIS DEMONSTRATION NEEDS THE
240 REM D/A BUFFER ROUTINE LOADED AT
250 REM XX=634.
260 REM
 270 REM TESTING DONE ON DISK DRIVE 1
280 REM
290 REM =======================
300 REM      J.HOOGSTRAAT
310 REM
320 REM BOX 20, SITE 7, SS 1
330 REM CALGARY, ALTA.   T2M-4N3
340 REM          PH(403) 239-0900
350 REM =======================
360 REM
370 REM
380 REM CREATE A SEQUENTIAL TEST FILE
390 REM ------------------------------
400 REM
410 F$="TESTING-TESTING"
420 XX=634:GOSUB1120
430 DK=1:CE=15:CS=2:CR=3:NN=200
440 DIMT(40),S(40)
450 A$="I"+CHR$(48+DK):OPENCE,8,CE,A$
460 A$="@"+CHR$(48+DK)+":"+F$+",U,W"
470 OPENCS,8,CS,A$
480 A$="...":FORI=1TO3:A$=A$+A$:NEXT
490 FORI=1TO27:PRINT#CS,A$:NEXT
500 CLOSECS
510 REM
520 REM FIND TRACK AND SECTOR EXTENTS
530 REM FOR CREATED TEST FILE
540 REM ------------------------------
550 REM
560 L=LEN(F$)
570 A$=CHR$(48+DK)+":"+F$+",U,R"
580 OPENCS,8,CS,A$
590 T=18:S=1:N=0
600 PRINT#CE,"U1:"CS;DK;T;S
610 SYSXX,0,CS,1,S$,1:S=ASC(S$)
620 FORI=2TO255STEP32
630 SYSXX,0,CS,I,A$,2,T$,1,S$,1,N$,L
640 IFASC(A$)>128ANDF$=N$THEN670
650 NEXT:IFS<255THEN600
660 PRINT"FILE "F$" NOT FOUND":END
670 N=N+1
```

```
680 T(N)=ASC(T$):S(N)=ASC(S$)
690 PRINT#CE,"U1:"CS;DK;T(N);S(N)
700 GET#CS,T$,T$,S$:IFT$<>""THEN670
710 CLOSECS
720 REM
730 REM OPEN RANDOM FILE WITH THE TEST
740 REM FILE EXTENTS. FILL IT ALL UP
750 REM ------------------------------
760 REM
770 PRINT"[cs]"
780 OPENCR,8,CR,"#"
790 FORI=1TON:A$=CHR$(I+48)
800 FORL=1TO5:A$=A$+A$+A$:NEXT
810 PRINT#CE,"U1:"CR;DK;T(I);S(I)
820 SYSXX,1,CR,2,I,-1,A$,NN
830 SYSXX,0,CR,2,S,U,A$,NN
840 PRINT"[dn]BLOCK";S:PRINTA$;
850 PRINT#CE,"U2:"CR;DK;T(I);S(I)
860 NEXT
870 REM
880 REM UPDATE SOME TEXT IN A BLOCK
890 REM --------------------------
900 REM
910 REM
920 INPUT"[dn]BLOCK,POS,TEXT";B,P,B$
930 PRINT"[cs]"
940 FORI=1TON
950 PRINT#CE,"U1:"CR;DK;T(I);S(I)
960 IFI<>BTHEN990
970 SYSXX,1,CR,7,P
980 SYSXX,1,CR,11+P,B$,LEN(B$)
990 SYSXX,0,CR,2,S,U,A$,NN
1000 PRINT"[dn]BLOCK"S;
1010 PRINT" LAST UPDATE AT POS";U
1020 PRINTA$;
1030 PRINT#CE,"U2:"CR;DK;T(I);S(I)
1040 NEXT
1050 GOTO920
1060 REM
1070 REM LOAD UP THE D/A BUFFER ROUTINE
1080 REM AT LOCATION XX. THIS ROUTINE
1090 REM A TOTAL RELOCATABLE.
1100 REM ----------------------------
1110 REM
1120 FORI=1TO306:READA:POKEXX-1+I,A:NEXT
1130 RETURN
1140 REM
1150 REM INSERT DATA STATEMENTS
1160 REM FOR D/A BUFFER ROUTINE HERE
1170 REM TOTAL LENGTH 306 BYTES
1180 REM
```

Filestatus

Henry Troup

There's been quite a lot written about disk files, and tape files, but very little about the PET's logical files. Here are some suggestions and a routine which may have some utility.

When you OPEN a file, you specify a logical file number, a device number, and (optionally) a secondary address, and filename. Then the PET does what is necessary. This information is saved, the number of files open is incremented and checked, and action is taken to open the file.

The file data is stored in three tables - logical files, devices, and secondary addresses. The tables start at $0251 ($0242 old ROM), $025B ($024C), and $0265 ($0256) respectively. The count of number of files is at $00AE ($0262). The filename is not saved - it's sent to the device.

The secondary address is OR'd with $60, and then stored. If no SA is specified, a value of $FF will be found in the table.

When a file is closed, the file last opened is swapped into its place. So if you open files 1, 3, and 5; and then close 1, the file table contains entries for 5 and 3 (plus a dummy copy of 5).

Now, we can write a routine to check on file status. Here it is:

```
10 REM FIND FILE STATUS
15 INPUT"LOGICAL FILE NUMBER ";LF
20 NF = PEEK(174):IF NF = 0 THEN PRINT "NO FILES
OPEN":END
30 PF = 0:FOR X=1 TO NF:IF PEEK(592+X) = LF THEN PF = X
40 NEXTX:IF PF = 0 THEN PRINT "FILE" LF "NOT OPEN":END
50 PRINT "LOGICAL FILE";LF "OPEN"
52 PRINT "ON DEVICE";PEEK(602+PF)
55 P = PEEK(612+PF) AND 159 :IF P = 159 THEN P = 0
60 PRINT "WITH SECONDARY ADDRESS";P
```

To use this, just open the files, and GOTO10. If you RUN the program, you'll abort all files.

You could use a version of this routine if you're doing dynamic LOADs - files are not affected by the LOAD, and you can find them.

More On Screen Print

John [illegible],
Wawa, Ont.

I found Jim Butterfield's machine language Screen Print Routine (Transactor #5) very useful in a program I am developing. But in order to stretch the forty columns on the screen to eighty columns on the printer I have added an [illegible].

The change is quite easy.

Method #1 using Supermon1.0

1. load the screen print routine code,

2. use command '.T 0359 03B3 035E' to open up 5 bytes in the code at $0359,

3. use command '.M 0359 035E' and change
   '.: 0359 A9 11 AE 4C E8 A9 11 AE' to
   '.: 0359 A9 01 20 D2 FF A9 11 AE',

4. use command '.M 03B0 03B7' and change 'A6' at $03B0 to 'A1',

5. use command '.S "dn:name",dv,033A,03B9'.

Method #2 using the Basic Loader for the code

1. load the screen print routine basic loader,

2. change 947 in line 100 to 952,

3. add ',1,32,210,255,169' to the end of the DATA statement at line 230,

4. change 166 at the end of line 330 to 161,

5. save the modified program.

This modification sends a control character (CHR$(1) as per the above modification) to the printer after every carriage return.

To use the screen print routine simply use 'SYS826' in your code. To change or ensure the mode of the routine just use 'POKE858,1 or 129' before the SYS826 command. For '[illegible]' mode, use '1': for 'unenhanced' mode, use '129'.

commodore

comments and bulletins
concerning your
COMMODORE PET™

The Transactor

VOL 2
BULLETIN 11

PET™ is a registered Trademark of Commodore Inc.

Bits and Pieces

WordPro and the NEC Spinwriter

Those using WordPro 3 or 4 are probably just realizing the potential of the PET as a dedicated wordprocessing system. With a Spinwriter for letter quality hard copy, this potential is substantialy increased. However, the Spinwriter requires a little preliminary set-up before it will operate correctly with WordPro. The front panel switches of the NEC are covered in the WordPro manuals but some extra switches inside the printer are not.

Inside the Spinwriter are four large circuit boards near the back of the unit. (A smaller fifth board is also there but not important here) The two boards closest to the back of the housing contain these extra switches. A word of caution: these boards support some CMOS chips... excessive static discharge to pins on CMOS chips will result in ireparable damage. You may want to have qualified personel make these changes.

On the very back board lies one of these switches. The switch, labelled 'SW1', is actually a DIP switch with 8 small slide switches on it. The second most back board contains the other three DIP switches labelled 'SW1', 'SW2', and 'SW3'. Early versions of these boards require you to pull them out of their sockets to gain access to the switches. This also means removal of a bracket and four cable connectors, two of which are tucked away at the right of the unit. Newer versions have the DIP switches placed near the top edges of the boards which will have you finished these mods in a flash. NEC assures me that both versions operate identically, only the board artwork was changed.

Now for the switch positions. Each set of 8 slide switches for the four DIPs will be labelled from left to right 1 for on and 0 for off:
(X = Do NOT Change)

back board :	SW1 =	0 0 1 0 1 X 1 1
2nd from back board :	SW1 =	0 0 0 0 0 0 0 0
	SW2 =	1 0 1 0 0 0 0 0
	SW3 =	1 0 1 0 0 0 0 0

Soft Disk Device Number

```
OPEN 1, 8, 15
PRINT#1,"M-W" CHR$(50) CHR$(0) CHR$(2) CHR$(9+32) CHR$(9+64)
```

The above command sequence will change a Commodore disk unit from device #8 to device #9. This works on the 2040 (DOS 1.0), the 4040 (DOS 2.0) or the 8050 (DOS 2.5). Once executed, another logical file must be OPENed to the command channel else a ?DEVICE NOT PRESENT ERROR will occur on the next PRINT#1. Alternately, since device #8 is no longer on the bus, CLOSE 1 and reOPEN using 9 instead of 8. The disk can actually be changed to any device number by substituting the 9 in the last two CHR$'s for any number between 8 and 15. Reset (PRINT#1,"U:" or "UJ") or power up will restore to device #8.

This works best when you need two disks on line but don't want to cut the jumpers of the main logic board inside the disk. Remember though, if two disks are powered up on the bus as device #8, the above sequence will change the both to device #9.

Commodore Education Advisory Board

Commodore has now received enough educational programs to produce and distribute 4 CEAB Diskettes, with a fifth one in the works. On behalf of Commodore, the Board and the recipients, I would like to thank all who have contributed. Through you we have successfully established a software share program for learning institutions across Canada and beyond. Let's keep it going!

TPUG Minutes

Richvale Telecommunications have available cassette recordings of the Toronto PET Users Group meetings. Richvale also has CEAB programs on tape for those operating without disk. For more information contact:

Richvale Telecommunications
10610 Bayview Ave. Unit 18
Richmond Hill, Ontario
L4C 3N8
416 884 4165

Supermon Notes

To get 'long' disassemblies on your printer, find the line-count with:

```
.H xxxx,yyyy A9 16 85 B5
```

where xxxx to yyyy is the memory range of Supermon. Change the '16' value to some higher number (maximum FF) to disassemble lots of lines at a time.

If you'd like the output split into pages on your 202X printer, that's all you need do. PET printers will page after every 60 lines of output and continue printing for the specified number of lines. But if you want a 'continuous' printout without paging, you should also do a hunt for:

```
.H xxxx,yyyy 86 B9 A9 93
```

and change 93 to 13. Remember to restore the 16 and 93 values if you plan to return to "screen" monitor.

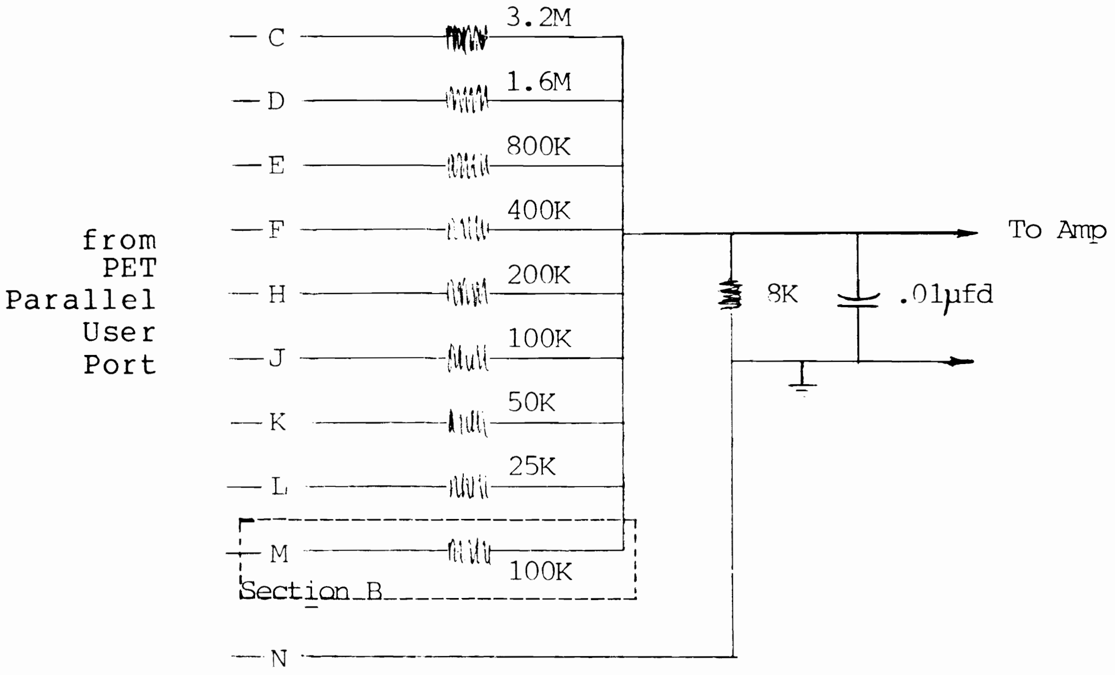
PET Sound

These next two items go hand-in-hand. The first was originaly printed in Volume 1 Transactor but, due recent inquiries, felt it worth reprinting in Volume 2. The second item is an inexpensive amplifier submitted by Tom Guzik of the Selkirk Electronics Club in Thunder Bay.

Poor Man's D/A Converter

Cheap; good for generating Chamberlin style music. Precision resistors are preferred, but most anything will generate a recognizable sound.

Section B of the diagram supports CB2 sound effects - so that this interface covers most sound requirements.

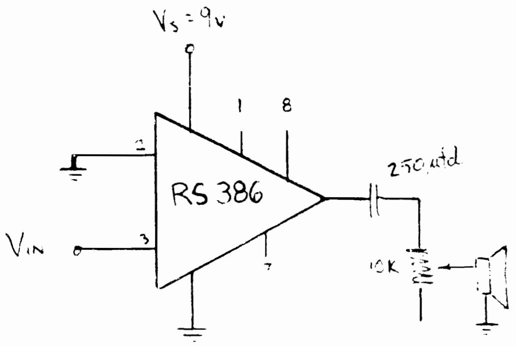


The capacitor provides some reduction of the sampling frequency (when generating music) ...tone controls on the amplifier will also help, if available.

The output of this D/A converter can be fed directly into an input of your stereo for excellent results.

500 Milliwatt Amplifier

This simple 500mw amp works on 9 volts available from pin 1 or 4 of the J11 connector inside the PET. All you need is a $2.00 I.C., a 50 cent capacitor, a spare potentiometer and a speaker.



Commercial Salvaging of Information from 2040 Diskettes

Diskette salvaging should be seldom needed by the average PET user. If backup copies are made, and due care is exercised, there is little chance of losing information. Even so, there are occurences where vital information is lost and urgently needs to be retrieved, if possible.

Of course there are cases where diskettes are too badly damaged to recover. Such cases would include exposure to a strong magnetic field, physically corrupted disks (torn, folded, coffee stained, etc.), or even re-formatted diskettes. However, some forms of diskette damage can be overcome. For example, a disk that can be initialized but has an unreadable directory stands an excellent chance of being totally reclaimed. Even diskettes that can't be initialized can often be recovered.

There are now, in Toronto, 3 diskette repair stations prepared to offer this service on a commercial basis to the PET community.

Syntax Logic Design
32 Ecclesfield Drive
Agincourt, Ontario
M1W 3J6
416 498 1093
416 447 1750

Technical Data Services
19 Wagon Trailway
Willowdale, Ontario
M2J 4V4
416 497 0595

Bret Butler
17 Astoria Ave.
Toronto, Ontario
M6N 2V5
416 763 6758

Fees

Standard charges have been set up for all 3 stations. Anyone wishing diskette repair should send the diskette to one of the above addresses, amply protected for transit, and include a cheque or m/o for $25.00. Any pertinent information about the diskette would also be helpful (directory listings, WordPro files?).

Diskettes that cannot be repaired will be returned with a written report and a refund of $15.00.

Information that is recovered will be transfered to a new diskette and returned with the original and a written report.

The above applies to sequential type data only (i.e. PRG files, SEQ files or USR files). Direct access information will require custom work at an extra $25.00 or more.

It is suggested that customers call before sending any material.

Text Editor

Jim Butterfield
Toronto

A poor man's word processor? Not exactly, although this program can be used in that way. It's more accurately a general purpose text editor, and can create, revise and print most sequential data files.

The program is line-oriented: it gets a line, then outputs it. However, there are features that allow you to deal with smaller elements - words or characters - or larger elements such as paragraph or entire text.

Because it keeps only a line at a time, it will run on very small PETs and you won't be bothered by garbage collection delays. It's written entirely in BASIC: that makes it portable and easy to chang (say to cassette tape operation). It won't run too fast, but that's part of the plan: you'll be able to stop and correct information as you go.

For cassette tape operation, just change the OPEN statements in lines 120 and 160. If you still have original ROMs, you'll need to change SW=ST on line 410 to read IF ST <> 0 GOTO 470.

Operation

You can enter information from the keyboard and/or a file; you can write the output to either a file or the printer. Just answer the startup questions.

If you have an input file, you'll be prompted at the start and at other parts of the program run with a half-shaded character. This asks you to supply an input-mode. Your choices are:

I - Don't input; accept an insertion from the keyboard;
T - Input the entire text from the file;
S - Search; input until you receive a selected character string;
P - Input a paragraph;
L - Input a line;
W - Input a word;
C - Input a character.

Press any of the above characters, or press SPACE to continue in the same input-mode as before.

Insert mode, I, remains in force until you enter a null line, that is, a line with no characters (not even a space). At that time, you'll either prompt for a new input-mode, or quit if there's no input remaining.

If you don't have an input file or if your input file is finished, you'll go directly into Insert mode without prompting. Entering a null line will stop the program.

All input-modes except Insert can be changed while input is taking place. For example, if you're in Text mode, touch the L key for Line mode, and you'll stop at the end of the current line; or W will stop you after the next word.

When input pauses, you may press RETURN and input will resume, printing the next line, word, or whatever. Alternately, you may delete or insert text, pressing RETURN when you are finished.

If you have deleted or inserted text, you will be prompted for a new input-mode. Select it, or press SPACE to continue as before.

When you're in Paragraph mode, a deletion or insertion will signal the program that you probably want to add a paragraph to the text. In this case, pressing RETURN won't take you back to the prompt for inputmode. Like Insert mode, you can keep going until you enter a null line.

A word about null lines and blank lines. A null line, which has nothing on it, is never written. If you want a blank line to be written, you must put at least one space there. A blank line - containing one or more spaces - is used by the program to detect the end of a paragraph.

To review: a null line is not written; it's a good way of deleting a line entirely from a file. A blank line, with one or more spaces, will be written and mark the end of a paragraph.

For your convenience, the Delete key has an automatic repeat feature built in. Cursor control keys other than Delete are ignored.

As mentioned before, you can switch modes during input just by tapping a key. Input timing is rather brief, however, during Character mode or word mode. In this case it's easier to force the input-mode prompt with a "dummy" insertion: tap SPACE, then DELETE. You will have changed nothing, but the input-mode prompt will appear when you press RETURN.

NOTE : 'CL' IN SQUARE BRACKETS MEANS CURSOR-LEFT

```
100 PRINT"TEXT EDITOR     JIM BUTTERFIELD"
110 INPUT"INPUT FILE NAME  N[CLCLCL]";N$
120 IF N$<>"N"THEN M=2:OPEN1,8,3,N$
130 INPUT"OUTPUT FILE TYPE (DISK OR PRINTER)  P[CLCLCL]";T$
140 IF ASC(T$)<>68 GOTO 200
150 INPUT"OUTPUT FILE NAME";F$
160 OPEN 2,8,4,"0:"+F$+",S,W":GOTO 210
200 OPEN 2,4:U$=CHR$(17)
210 B$=CHR$(32)+CHR$(20)+CHR$(20)
220 P$=CHR$(175)+CHR$(157)
230 R$=CHR$(13)
240 S$=CHR$(32)
250 J$=CHR$(168)+CHR$(157)
260 POKE 59468,14
270 D$=R$:GOSUB820
280 IF N$="N"GOTO 480
300 REM TEST KEYBOARD FOR MODE
310 GET M$:GOSUB860
400 REM GET INPUT STUFF
410 GET#1,D$:SW=ST
420 IF D$=R$ OR (D$=K$ AND S=1) THEN GOSUB500
430 L$=L$+D$:IF D$<>S$ THEN S=1
440 PRINT D$;:IF M=6 THEN GOSUB500
450 IF D$=G$ THEN IF LEN(L$)>=H THEN IF RIGHT$(L$,H)=H$ THEN K=1:GOSUB500
460 IF SW=0 GOTO 300
470 CLOSE 1
480 M=0:GOSUB500
490 CLOSE2:END
500 F=0:S=0:REM  PAUSE FOR CHANGE/RETURN KEY EXITS
510 L=LEN(L$)
520 IF M=2 GOTO700
530 IF (M=3 OR  M=7) AND K=0 GOTO700
540 PRINT P$;
550 R=R+1:P=PEEK(151):GETC$:IFC$<>""THEN R=0:C=ASC(C$):GOTO580
560 IF P=255 OR C<>20 THEN C=0:GOTO550
570 IF R<20 GOTO550
580 IF C=20 AND L>0 THEN F=1:L$=LEFT$(L$,L-1):PRINTB$;
590 IF C=13 GOTO700
600 IF C=34 THEN PRINT CHR$(34);CHR$(20);
610 IF (C AND 127)>31 THEN F=1:PRINT C$;:L$=L$+C$
620 GOTO510
700 REM  RETURN - TEST FOR EXIT
710 IF D$<>R$ AND M>1 GOTO810
720 IF L=0 GOTO750
730 IF K=0 AND M=3 THEN FOR J=1 TO L:IF MID$(L$,J,1)=S$ THEN NEXT J:K=1
740 PRINT" ":PRINT#2,U$;L$;R$;:L$="":IF M<3 OR K=1 GOTO510
750 D$=""
800 REM CHECK FORMAT KEYS
810 IF F=0 GOTO920
820 IF M=0 GOTO920
830 PRINT J$;
840 GET M$:IF M$=""GOTO840
850 IF M$="I" THEN M=1:K$="":G$="":GOTO510
860 IFM$="S" THEN M=7:K$="":GOSUB930
870 IF M$="C" THEN M=6:K$="":G$="AA"
880 IF M$="W" THEN M=5:K$=S$:G$="AA"
890 IF M$="L" THEN M=4:K$="":G$="AA"
900 IF M$="P" THEN M=3:K$="":G$="AA"
910 IF M$="T" THEN M=2:K$="":G$="AA"
920 K=0:RETURN
930 PRINT:INPUT"SEARCH FOR";H$
940 H=LEN(H$):G$=RIGHT$(H$,1):RETURN
```

CARD PRINTING UTILITY

D. Hook
58 Steele St.
Barrie, Ontario
L4M 2E9

There are some occasions when you may wish to program a card game. By addition of a subroutine that gives good graphics (for the card symbols), most would be improved. Since this would represent too much work, the finished version fails to take advantage of the Pet's forte.

This program attempts to remove the drudgery from the task. By understanding its mechanics, I hope that you can add the feature. See me at the next TPUG meeting if you can't be bothered typing it all in.

I have included a variables cross-reference (thanks to Jim Butterfield) and a separate chart to make the graphics more easily entered.

Consult the listing as we work through the flow:

Line 10-80:
Data statements used in the initialization.

Subroutine 40000:
First seeds the random number generator. Creates the D%(array for D% decks of cards. The card values are:

0-12	A2345...K	clubs
13-25	A2345...K	diamonds
26-38	A2345...K	hearts
39-51	A2345...K	spades

These values are very important identifiers to recover the suit and value of the card in question.

Since D% is no longer needed, it is redefined to the total number of cards in the game (minus 1).

The I$(array is simply the index value to be printed in the corner of each card. These are read from Data Line 10.

The S$(array is two-dimensional. The first has 0,1,2 suit symbols read from blanks, S1$ and S2$. The second dimension refers to the suits: 0-3 in the same order as above.

The S%(array is for spot cards 1 through 10, and for rows 1 to 7 of each card to be printed. Line 20 provides the data. Note the lack of "0" entries, as the comma is sufficient.

The entries in the array indicate the NUMBER OF SUIT SYMBOLS that belong in each row. Since we are not concerned with the actual suit at this time, all the spot cards of a given value will be the same.

For the face cards, the F$(I,J) array is defined:

```
I = 1,2,3   for J,Q,K
J = 1-7     for rows 1-7
```

The data in Lines 40-80 give the strings for the card pictures. To facilitate entry of these graphics, the table below is provided:

GRAPHICS DATA LINES 40 - 80

```
Jack
 1.  "  la RO b  Rv )  b  b  "
 2.  "  b  RO <  Rv '  :  b  Rv b  b  "
 3.  "  b  !  RO )  Rv )  RO la Rv b  b  "
 4.  "  b  &  )  V  RO )  Rv &  b  "
 5.  "  b  b  la RO )  Rv )  RO !  Rv b  "
 6.  "  b  b  '  P  b  RO ;  Rv b  "
 7.  "  b  b  RO )  b  la "

Queen
 8.  "  )  P  b  b  b  "
 9.  "  b  RO )  Rv B  *  b  b  b  "
10.  "  b  RO b  b  b  b  Rv ;  b  "
11.  "  b  &  V  V  V  &  b  "
12.  "  b  <  RO b  b  b  b  Rv b  "
13.  "  b  b  b  4  ]  )  b  "
14.  "  b  b  b  L  )  "

King
15.  "  la RO b  b  Rv )  b  "
16.  "  b  b  '  b  &  B  b  "
17.  "  b  RO )  b  b  b  <  Rv b  "
18.  "  b  &  ?  ?  ?  &  b  "
19.  "  b  RO ;  b  b  b  Rv )  b  "
20.  "  b  ]  &  b  %  b  b  "
21.  "  b  RO )  b  b  la "
```

Code:

```
"  = quote
b  = blank
Rv = reverse on
RO = reverse off
la = shifted left arrow
```

All other keys are their "shifted" equivalents (i.e. graphics).

We now return to the main-line program.

Three subroutines are offered in the menu:

The DISPLAY CARDS is simply for use in de-bugging, and need not be part of any program that you may use.

The SHUFFLE is an integral part of any game program, and is both compact and fast.

The SUBROUTINE FOR GAMES allows the many options that are essential to the utility of this program.

DISPLAY CARDS--SUBROUTINE 42000

Virtually all of this is duplicated in Subroutine 43000, where the main discussion will take place.

The purpose is to print (on the sceen) the pictures for all the cards used in the game. Line 42020 defines the starting line, L%=7, sets up to print, A%=5, cards across the screen, and will start printing at tab, TB=0.

Variable L is the loop counter to print from card C%=0 to C%=D%, the last card of the last deck. Since no shuffling takes place, you may see the deck(s) flash by. Starting at the A of clubs, the K of spades will be the last, regardless of the number of decks selected.

SHUFFLE--SUBROUTINE 41000

Some sort routines require an extra array to store the intermediate values. Others require a pointer array to flag the cards already taken.

No such precautions need be taken here. The array is sorted in place. A card already chosen will not be shuffled again, so the process takes only N-1 passes for N cards. If you haven't seen this before, follow the logic below:

The loop variable is I, for the D% cards. Variable J% provides a random number from 0 to D% on the first pass. Thus all cards are available to be selected.

Assume we have one deck, so D% is 51. Assume the random number, or J%, is 14. Let's say that card number 14 is the A of clubs. In our deck, the card value is 0 (see above).

Define K%=D%(14), which means K%=0 this time.

Now comes the exchange, where we take the last value, D%(51), and put it into D%(14). (We haven't lost D%(14), since it is stored in K%).

Put K% into the last position, or D%(51), and the first pass is complete.

If you have observed that the loop counter, variable I, appears throughout, you may see what happens next.

As NEXT I is reached for the next pass, the upper boundary shrinks by one. The (former) last entry cannot be chosen for J%, nor be part of the subsequent exchange.

Each pass gets another card, and the deck(s) get shuffled. A pretty tidy routine!

SUBROUTINE FOR GAMES--SUBROUTINE 43000

Initially you are asked to respond to a series of questions which will establish the various variables to be used in your game. These prompts are only to provide a cue for your usage, so lines 43000 to 43040 can be dropped. Be advised that you will have to provide for these to be defined in your program.

P% is the total number of cards to be printed. Make it a large number if you plan to play your game for a while.

L% is the screen line number where the card is to be printed. The cards are 9 rows high, so watch where you start. You may define this differently before each subroutine call.

A% is the number of cards across the screen. The tab values are reset based on this value. Since the cards are 7 spaces wide, only 5 cards may fit across the screen. This too may be changed before calling.

TB% is the tab position for the first card on the line. Note that if 5 cards are printed, you must start at TB%=0. Whenever A% is checked, the tab position advances by 8 positions (Line 43150). Change this line if you want wider spacing between the cards.

M% is the variable to detect when to reshuffle the whole deck(s). Line 43040 sets this to the whole deck(s), so keep this if it suits you. Otherwise redefine it to a convenient number, based on the number of decks in play.

Note that this routine does not give an automatic first shuffle when the "game" begins. Do this yourself with a call to SBR 41000.

On to the meat of the routine:

Line 43100:
Initializes the card counter, C%, to select the next card from the deck. This is an index to the actual card array, D%(. The next check is to see whether it is time to shuffle--if it is, then the shuffle is done.

Please observe that I have included the "L" loop as part of the subroutine. In your game this would undoubtedly be part of the main code. It has been done this way to allow printing as part of this program.

Line 43130:
You will recall that our deck consists of coded (0-51) values to represent the cards. Here we extract the suit into variable S%=0,1,2,3 and the card value into variable V%=1-13 (A23...K).

Since you will want to employ these values in your game, you have them on return to your main routine.

Line 43140:
Checks to see if it is time to print back in the "first" location again. Depends on the afore-mentioned values for A%, TB%, L%. Recall that variable "L" simply is the counter for the total number of cards printed.

Line 43150:
Tabs ahead 8 spaces horizontally and moves the cursor up. Only used where several cards are to appear on the same screen line. Change as described above, if you wish.

Line 43160:
The top line of every card has its "index" name (upper left corner) and is filled out with blanks. We then enter the loop to print the next seven rows down.

Line 43170:
If the card is a face card, branch around to Line 43500.

Line 43250:
This part gets tricky...use the value of the card, V% and the row number,J as an index into the S%(array. That array will give you the number of suit symbols (0,1,2) to be printed on a given line for the card.

Then combine that with the suit variable, S% to determine which symbols to put on that line. The array, S$(gets the right ones to print.

For spot cards, this is repeated for each of the seven rows.

Lines 43500,43510:
For face cards, we need to print the proper suit symbol near the upper left and the lower right. If J=1 or J=7 then this is done at the start of each of these lines.
Then we look at array F$(to get the card value, V%-10 and the row number, J.

Line 43520:
If J<>1 or J<>7 then we just do the second half of the above.

We loop 7 times then print the bottom line of the card. The lower right corner also contains our "index" name.

Line 43800:

Since we are printing "L" cards, we cannot forget this loop. This, like the FOR in Line 43100 should be in your main program.

```
CROSS REFERENCE - PROGRAM CARD UTILITY

A%     42020 42030 43020 43140
C%     42000 42010 43100 43130
D%     40000 40010 40020 41000 41010 42000 43040
D%(    40020 41000 41010 42010 43130
E9     40000
F$(    40080 42500 42510 42520 43500 43510 43520
I      40000 40020 40030 40050 40060 40070 40080 41000 41010
I$(    40030 42050 42750 43160 43750
J      40000 40020 40070 40080 42050 42070 42500 42510 42520 42750 43160
       43250 43500 43510 43520 43750
J%     40000 41000 41010
K%     40000 41000 41010
L      42000 42030 42800 43100 43140 43800
L%     42020 42030 43010 43140
M%     43040 43100
P%     43000 43100
S$(    40050 40060 42070 43250
S%     42010 42070 42500 42510 43130 43250 43500 43510
S%(    40070 42070 43250
S1$    40040 40050
S2$    40040 40060
T%     42030 42040 42050 42070 42500 42510 42520 42750 43140 43150 43160
       43250 43500 43510 43520 43750
TB%    42020 42030 43030 43140
TI     40000
V%     42010 42050 42060 42070 42500 42510 42520 42750 43130 43160 43170
       43250 43500 43510 43520 43750
Z        130   140   150 15010 43000 43010 43020 43030 43040
Z$       120   130   160 15000 15010
10 DATA A,2,3,4,5,6,7,8,9,10,J,Q,K
20 DATA,,,1,,,,1,,,,,,1,1,,,1,,,1,2,,,,,,,2,2,,,1,,,2
30 DATA2,,,2,,,2,2,,1,,2,,2,2,,2,,2,,2,2,,2,1,2,,2,2,1,2,,2,1,2
40 DATA" ◥▆ ▞◤   "," ▪▘▟ └┘ ▟  "," ▌▆◤▞◤▀◥▟  "
50 DATA" ▒◤×▆◤▟▒ ","  ◥▆◤▞◤█ ▟ "
60 DATA"  ┌┐ ▙ ▟ ","  ▆◤ ◥","  ◤┐   "," ▆◤▟│ │   "," ▀    ▟  "," ▒•▒•▒ "
70 DATA" ▀▄    ▟ ","  │▐◤ ","  ▄▁▆◤","◥▆   ▞◤ ","  │▒│ "," ▆◤    ▘▟ "
80 DATA" ▒▘▘▘▒ "," ▙    ▞◤ "," │▒│  "," ▆◤   ◥"
90 GOSUB40000
100 PRINT"■"TAB(10)"■CARD UTILITY":PRINT"■1. DISPLAY CARDS":PRINT"■2. SHUFFLE
110 PRINT"■3. SUBROUTINE FOR GAMES":PRINT"■4. QUIT":PRINT"■■■SELECTION  ?";
120 GETZ$:IFZ$=""THEN120
130 Z=VAL(Z$):PRINTZ:IFZ<1ORZ>4THEN100
140 IFZ=4THENEND
150 ONZGOSUB42000,41000,43000:PRINT"■■DONE--HIT A KEY
160 GETZ$:IFZ$=""THEN160
170 GOTO100
14998 END
15000 INPUT"   ▒███";Z$:IFZ$="▒"THEN15000         :REM INPUT SBR.
15010 Z=VAL(Z$):RETURN
40000 I=RND(-TI*1E9):J=0:D%=0:J%=0:K%=0          :REM INITIALIZATION
40010 INPUT"NUMBER OF DECKS    1███";D%
40020 DIMD%(D%*52):FORI=1TOD%:FORJ=0TO51:D%(52*(I-1)+J)=J:NEXTJ,I:D%=D%*52-1
40030 DIMI$(13):FORI=1TO13:READI$(I):NEXTI
40040 S1$="   ♣   ♦   ♥   ♠   ":S2$=" ♣   ♣ ♦   ♦ ♥   ♥ ♠   ♠ "
40050 DIMS$(2,3):FORI=0TO3:S$(0,I)="       ":S$(1,I)=MID$(S1$,I*4+1,7)
40060 S$(2,I)=MID$(S2$,I*6+1,7):NEXTI
40070 DIMS%(10,7):FORI=1TO10:FORJ=1TO7:READS%(I,J):NEXTJ,I
40080 DIMF$(3,7):FORI=1TO3:FORJ=1TO7:READF$(I,J):NEXTJ,I
40090 RETURN
40999 REM SHUFFLE
41000 FORI=0TOD%:J%=(D%+1-I)*RND(1):K%=D%(J%)
41010 D%(J%)=D%(D%-I):D%(D%-I)=K%:NEXTI:RETURN
41999 REM DISPLAY ALL CARDS
42000 PRINT"■":C%=0:FORL=0TOD%:C%=C%+1:
42010 S%=D%(C%-1)/13:V%=D%(C%-1)-13*S%+1
42020 L%=7:A%=5:TB%=0
42030 IFL/A%=INT(L/A%)THENT%=TB%:PRINTLEFT$("■■■■■■■■■■■■■■■■",L%):GOTO42050
42040 T%=T%+8:PRINT"IIIIIIIII]";
42050 PRINTTAB(T%)"■"LEFT$(I$(V%)+"       ",7):FORJ=1TO7
42060 IFV%>10THEN42500
42070 PRINTTAB(T%)"■"S$(S%(V%,J),S%):GOTO42750
42500 IFJ=1THENPRINTTAB(T%)"■ "MID$("♣♦♥♠",S%+1,1)F$(V%-10,J):GOTO42750
42510 IFJ=7THENPRINTTAB(T%)"■"F$(V%-10,J)"■"MID$("♣♦♥♠",S%+1,1)" ":GOTO42750
42520 PRINTTAB(T%)"■"F$(V%-10,J)
42750 NEXTJ:PRINTTAB(T%)"■"RIGHT$("       "+I$(V%),7)
42800 NEXTL:RETURN
42999 REM GAME-TYPE SUBROUTINE
43000 PRINT"■HOW MANY CARDS TO PRINT";:GOSUB15000:P%=Z
43010 PRINT"START ON LINE (1-16)";:GOSUB15000:L%=Z
43020 PRINT"HOW MANY ACROSS (1-5)";:GOSUB15000:A%=Z
43030 PRINT"START AT TAB (0-32)";:GOSUB15000:TB%=Z
43040 M%=D%+1:PRINT"SHUFFLE AFTER (1-"M%")";:GOSUB15000:M%=Z
43100 PRINT"■":C%=0:FORL=0TOP%-1:C%=C%+1:IFC%=M%+1THENC%=1:GOSUB41000
43130 S%=D%(C%-1)/13:V%=D%(C%-1)-13*S%+1
43140 IFL/A%=INT(L/A%)THENT%=TB%:PRINTLEFT$("■■■■■■■■■■■■■■■■",L%):GOTO43160
43150 T%=T%+8:PRINT"IIIIIIIII]";
43160 PRINTTAB(T%)"■"LEFT$(I$(V%)+"       ",7):FORJ=1TO7
43170 IFV%>10THEN43500
43250 PRINTTAB(T%)"■"S$(S%(V%,J),S%):GOTO43750
43500 IFJ=1THENPRINTTAB(T%)"■ "MID$("♣♦♥♠",S%+1,1)F$(V%-10,J):GOTO43750
43510 IFJ=7THENPRINTTAB(T%)"■"F$(V%-10,J)"■"MID$("♣♦♥♠",S%+1,1)" ":GOTO43750
43520 PRINTTAB(T%)"■"F$(V%-10,J)
43750 NEXTJ:PRINTTAB(T%)"■"RIGHT$("       "+I$(V%),7)
43800 NEXTL:RETURN
```

Simple 8010 Modem Programs

Jim Butterfield
Toronto

The programs that come with the Commodore 8010 modem may not quite fit your needs. For one thing, a NULL character from the line will cause the program to stop, since the input arrives as a null string and the ASC function won't work. If you communicate with computers that send parity - an extra bit intended to safeguard transmission - you'll get some funny looking things on your PET screen.

Speed is of the essence in this kind of Basic program: waste a few moments and you may lose an incoming character. As a result, the programs are no-frills. Watch carefully for timing if you try dressing them up with your own features.

PET TO ASCII

We need to translate PET's internal code to ASCII, and vice versa; and we need to do it fast. Result: an array for quick translation each way. F(x) translates incoming characters from the line; T(x) translates to the line.

Most non-printing characters are dropped; I've preserved only the carriage return, CHR$(13), and the Delete, CHR$(20) to PET and CHR$(8) to the line. If your favorite computer needs other special characters, you may put them in the table: for example, if the computer recognizes Data Link Escape (DLE, sometimes called Control-P), you could code it as shifted-zero on the PET keyboard with: 250 T(176)=16.

The POKE 1020,0 on line 280 is needed for the new 4.0 systems to ensure that IEEE timeout works properly.

PET to PET

Much simpler, of course, since no translation is needed. Delete the POKE 59468, 14 (or change it to POKE 59468, 12) if you want to stay in graphics. This way, you can draw pictures on the other PET's screen.

All of the cursor controls and graphics work, of course. You can even clear the opposite screen remotely, if you wish.

For communications to an ASCII system:

```
100 REM     8010 INTERFACE      JIM BUTTERFIELD
110 REM     FOR ASCII LINES
120 REM     SET SWITCH TO HD
200 DIM F(255), T(255)
210 FOR J=32 TO 64 : T(J)=J : NEXT J : T(13)=13 : T(20)=8
220 FOR J=65 TO 90 : K=J+32 : T(J)=K : NEXT J
230 FOR J=91 TO 95 : T(J)=J : NEXT J
240 FOR J=193 TO 218 : K=J-128 : T(J)=K : NEXT J
250 REM    ADD EXTRA FUNCTIONS HERE
260 FOR J=0 TO 255 : K=T(J)  : IF K THEN F(K)=J : F(K+128)=J
270 NEXT J
280 POKE 1020, 0 : POKE 59468,14
290 OPEN 5,5 : PRINT "ASCII I/O READY"
300 GET A$ : IF A$ <> "" THEN PRINT#5, CHR$(T(ASC(A$)));
310 GET#5, A$ : IF ST=0 AND A$ <> "" THEN PRINT CHR$(F(ASC(A$)));
320 GOTO 300
```

For Communications to Another PET:

```
100 REM     8010 INTERFACE     JIM BUTTERFIELD
110 REM     FOR PET INTERCOMMUNICATION
120 REM     SET SWITCH TO HD
280 POKE 1020, 0 : POKE 59468,14     If text mode desired
290 OPEN 5,5 : PRINT "PET I/O READY"
300 GET A$ : IF A$ <> "" THEN PRINT#5, A$;
310 GET#5, A$ : IF ST=0 THEN PRINT A$;
320 GOTO 300
```

Editor's Note

We're looking into the possibility of downloading PET programs using a simple BASIC driver. Attempts thus far have failed, mostly due to the fault of the driver. The task may require a little machine language, but we'll keep you posted.

An Incomplete PET/CBM Bibliography

Hands On Basic With A PET — Herbert D. Peckham
McGraw-Hill Ryerson 330 Progress Ave., Toronto Ont., M1P 2Z5

Some Common Basic Programs PET/CBM Edition — ed. L. Poole
Osborne/McGraw-Hill 630 Bancroft Way, Berkeley, Ca. 94710

Practical Basic Programs — ed. L. Poole
Osborne/McGraw-Hill 630 Bancroft Way, Berkeley, Ca. 94710

6502 Assembly Language Programming — Lance A. Leventhal
Osborne/McGraw-Hill 630 Bancroft Way, Berkeley, Ca. 94710

Pet and the IEEE 488 Bus (GPIB) — Eugene Fisher / C.W. Jensen
Osborne/McGraw-Hill 630 Bancroft Way, Berkeley, Ca. 94710

PET/CBM Personal Computer Guide * — C.S. Donahue & J.K. Enger
Osborne/McGraw-Hill 630 Bancroft Way, Berkeley, Ca. 94710

Care and Feeding of The Commodore PET
Elcomp Publishing 3873-L Schaefer Ave. Chino Ca. 91710

8K Microsoft BASIC Reference Manual
Elcomp Publishing 3873-L Schaefer Ave. Chino Ca. 91710

The Pet Subroutine Library * — Nick Hampshire
Computabits Ltd. P.O. Box 13, Yeovil, Somerset, UK

The Pet Revealed 2nd ed * — Nick Hampshire
Computabits Ltd. P.O. Box 13, Yeovil, Somerset, UK

6502 Software Design — Leo J. Scanlon
Howard W. Sams 4300 West 62nd St., Indianapolis, Indiana 46206

Programming & Interfacing the 6502, With Experiments — Dr. De Jong
Howard W. Sams 4300 West 62nd St., Indianapolis, Indiana 46206

Programming the 6502 — Rodney Zaks
Sybex 2344 6th St. Berkeley, Ca.

6502 Games — Rodney Zaks
Sybex 2344 6th St. Berkeley, Ca.

6502 Applications Book — Rodney Zaks
Sybex 2344 6th St. Berkeley, Ca.

Microprocessor Systems Engineering — R.C.Kemp/T.A.Smay/C.J.Triska
Matrix Publishers 30 N.W. 23rd Place Portland, Oregon 97210

Practical Microcomputer Programming: The 6502 — W. J. Weller
Northern Technology Books Evanston, Il.

Programming a Microcomputer: 6502 — Caxton C. Foster
Addison-Wesley Publishing 36 Prince Andrew Pl. Don Mills Ont.

PIMS
Scelbi Publishing Co. P.O. Box 133 PP Stn Milford , Ct. 06460

Pet Machine Language Guide — Arnie Lee
Abacus Software P.O. Box 7211 Grand Rapids Mich. 49510

Understanding Your PET/CBM Vol. 1
TIS Workbooks: Getting Started With Your PET
PET String and Array Handling
PET Graphics
PET Cassette I/O
Miscellaneous PET Features
PET Control and Logic
TIS P.O. Box 921 Los Alamos, Ca.

PERIODICALS

The Transactor
Commodore Bus. Mach. 3370 Pharmacy Ave. Agincourt, Ont. Canada

CPUCN Newsletter (UK)
Commodore Systems Ltd. Slough Trading Estate 818 Leigh Rd.
Slough Berkshire, England SL1 6BB

Commodore U.S. Newsletter
Commodore Systems Ltd. 3330 Scott Blvd. Santa Clara, Ca. 95051

Compute, The Journal of Progressive Computing
P.O.Box 5406 Greensboro, NC 27403

Cursor
The Code Works Box 550 Goleta, Ca 93017

The PAPER
P.O. Box 524 East Setauket, New York 11733

PET Prime c/o Guy Leger
Metro Separate School Board 146 Laird Dr. Toronto, Ont. M4G 3V8

Kilobaud MICROCOMPUTING 1001001 Inc.
Peterborough , NH 27403

Micro: The 6502 Journal
P.O. Box 6502 Chelmsford, Ma 01824

Creative Computing
P.O. Box 789 Morristown, NJ 07960

Practical Computing
IPC Business Press Oakfield House, Perrymount Rd.,
Hayward Heath, Sussex RH16 3DH, UK

Printout
Greenacre, North ST. Sussex RH16 3DH, UK

PET Benelux Exchange (Dutch text)
Burgemeester Van Such Telenstraat 46 7413 XP
Deventer, Holland

NOTES

The above books and periodicals cover a wide range of information and topics. The PET masochist reader will want (or already has them all) to include them in his or her library. The novice will find several elementary books. * Available from Commodore dealers.

The periodicals are directly related to the PET (or have significant monthly columns). The British magazines are usually available in large Metropolitan areas.

commodore

comments and bulletins
concerning your
COMMODORE PET™

The Transactor

VOL 2
BULLETIN 12

PET™ is a registered Trademark of Commodore Inc.

Transactor Article Contest Winners

In Transactor #8, we promised awards for the best articles published in Volume 2. We also promised free subscriptions to The Transactor Volume 3 for any article published. Here are the winners:

Best Article goes to J. Hoogstraat of Calgary, Alberta, for his BASIC Labelling Routine published this issue and also for his 2040 Disk I/O Routine in bulletin #10. Mr. Hoogstraat gets a free Visicalc package.

Runner up award goes to F. Van Duinen of Toronto, Ontario, for ?LOAD ERROR, D.R.I.P and Program Plus. Mr. Van Duinen receives a Commodore calculator, model # SR9190.

Free Volume 3 subscriptions are going to:

J. Hoogstraat
*Kevin Erler
*James Yost
Michael Casey
*B. Brown
Dave Hook
Henry Troup
*Sheldon H. Dean
Brad Templeton

F. Van Duinen
John A. Cooke
Chuan Chee
G. Hathaway
*L.D. Gardner
Paul Barnes
Gord Campbell
Don White

*Jim Russo
Rick Ellis
Jim Hindson
W.T. Garbutt
Tom Wojdylo
*S. Donald
*John Macdonald
Dave Berezowski
*Robert Oei

* Please call or send in your address.

This contest will be held again for The Transactor Volume 3, prizes may differ.

If you're asking "What about Jim Butterfield?", don't worry, he's been well taken care of.

As a Commodore dealer, Bill MacLean of BMB CompuScience was not eligible for a prize, but we'll figure something out.

I'd like to thank all who contributed to Volume 2 and special thanks to Jim and Bill for some really excellent stuff! Special thanks also to Terry Garbutt for his truly genuine help and support. Hoping to hear from all of you in Volume 3, I remain,

Karl J. Hildon
Editor, The Transactor

The Transactor is produced on the CBM 8032 using WordPro IV and the NEC Spinwriter

Exclusive OR on Your PET

In boolean algebra there are three main operators: AND, OR and NOT. All three of these are included in PET BASIC. However, one sometimes very useful boolean function was not included in BASIC. This is the EXclusive OR function. EXOR is a function of AND, OR and NOT:

```
  (a)EXOR(b) = ((a) AND (NOT(b)) ) OR ((b) AND (NOT(a)) )
```

Of course the above would result in ?SYNTAX ERRORS if coded literally. The following will accomplish a% EXclusive OR'd with b% in BASIC.

```
         ex% = ((a%)and(not(b%)))or((b%)and(not(a%)))
```

An Extra Note on Logical Operators

Try RUNning this short program: (enter exactly as shown)

```
     10 xt=5 : xf=6 : rem just random values
     20 print xtandxf
```

Now replace line 20 with each of the following line 20's and RUN each again.

```
     20 print xtorxf
     20 print xforxt
```

Each of the three will result in ?SYNTAX ERROR IN LINE 20. But why? When you hit return on a line of BASIC, the PET procedes to "tokenize" the line by parsing the characters from left to right.

Line:	Would be tokenized as:
20 print xtandxf	print x tan dxf
20 print xtorxf	print x to rxf
20 print xforxt	print x for xt

A general rule: When preceding logical operators with floating point variables, insert a space or enclose the variable in brackets. Integer type variables will not be succeptable to this problem because the "%" sign will act as a delimiter. Brackets are still necessary for hierarchy of operations.

This gotcha surfaces in one other command in BASIC 4.0:

```
     header"diskname",d0,ifx
```

The breakdown of this would be format a diskettes on drive 0 with "diskname" as the title and "fx" as the disk id. But on hitting return, a ?SYNTAX ERROR is printed because ifx is tokenized as ifx .

The BASIC Difference Is:

	BASIC 1.0	BASIC 2.0	BASIC 4.0
PEEK(50003) =	0	1	160
Disable STOP POKE	537,136	144, 49	144, 88
Enable STOP POKE	537,133	144, 46	144, 85

New BASIC 4.0 machines reportedly crash on some old programs. The culprit is most likely a disable STOP key POKE. Also check for POKE59458,62, the screen speed-up POKE. As mentioned before, this can also crash machines. See article this issue on BASIC 2.0 - BASIC 4.0 Conversions for more info.

Screen Loading

All you need is a "screen-set-up" routine to "draw" your screen out, and this program will store it on disk:

```
100 REM SCREEN SAVER
110 OPEN 8, 8, 8, "0:SCREEN NAME,P,W"
120 PRINT#8, CHR$(0)CHR$(128);
130 EN=33767 : IF PEEK(50003)=160 THEN EN=34767
140 FOR J=32768 TO EN
150 PRINT#8, CHR$ ( PEEK (I) ) ;
160 NEXT
170 CLOSE 8
180 END
```

Line 130 sets end screen (EN) to 33767 for 40 columns, 34767 for 80 columns.

SAVE this program and do a NEW. Now enter:

```
 10 ON X GOTO 120
100 PRINT "[clrscrn]";
110 X=1 : LOAD "0:SCREEN NAME",8
120 END
```

RUN this and the old screen should pop back on the screen as fast a loading 1k from disk. The cursor will remain in the home position since nothing is actually printed. No pointers or variables are changed since it was a "dynamic load". But the loader program would RUN from the beginning, hence the ON X GOTO statement. This could be expanded to accommodate more screen loads simply by adding more GOTO data to line 10 and setting X appropriately prior to the load. The SCREEN SAVER program could also be modified to store only a portion of the screen. But don't forget to change the load address in line 120, else the files will always load back to screen starting at HOME.

PET BASIC LABEL SUPPORT INTERFACE

J. Hoogstraat
Calgary, Alta

This amazing routine resides in the second cassete buffer and allows the use of labels in basic and has no effect on the speed of basic.

A label starts with a # character and is retricted in length to the basic line length.

EXAMPLE NO LABELS

```
100 FOR I = 1 TO 3
110 ON I GOSUB 500, 550, 600
120 NEXT
130 GOTO 800
140 :
500 PRINT "SUBROUTINE";I :RETURN
510 :
550 PRINT "SUBROUTINE";I :RETURN
560 :
600 PRINT "SUBROUTINE";I :RETURN
610 :
800 PRINT "END OF TEST":END
```

EXAMPLE WITH LABELS

```
10  SYS826
20  :
100 FOR I = 1 TO 3
110 ON I GOSUB #SUB1, #SUB2, #SUB3
120 NEXT
130 GOTO #ALLDONE
140 :
500 #SUB1:PRINT "SUBROUTINE";I :RETURN
510 :
550 #SUB2
555 PRINT"SUBROUTINE";I :RETURN
560 :
600 #SUB3:PRINT "SUBROUTINE";I :RETURN
610 :
800 #ALLDONE:PRINT "END OF TEST":END
```

The #labels can be mixed up with basic statement numbers.

```
110 ON I GOSUB #SUB1, 550, #SUB3
```

Since the routine resides in the second cassette buffer and modifies the basic GET character routine, it prohibits the use of any other routines in the second cassette buffer or the use of the DOS support program. However it can be made part of the DOS support program.

I do have available a modified DOS support program which includes the following:

1. Regular DOS support.

2. The BASIC label support interface.

3. An excellent repeat key function.

4. A basic disk append command. no messing around with tapes

Just send me $20.00 and a floppy and I will return a copy of the above including all the assembly source on your floppy, or for $27.00 I'll send you a floppy with the same.

By the way, the # label prefix is my choice and can be altered to any other special character.

Have a lot of Basic fun !!!

Editor's Note:

Mr. Hoogstraat's routine works on BASIC 2.0 only. To convert to BASIC 4.0, some JSRs would need changing. Also, the program could no longer reside in the second cassette buffer. This space is used by some new BASIC 4.0 commands.

```
800 FOR J=826 TO 1008 : READ X :POKE J, X : NEXT
826 DATA 169,  71, 133, 113, 169,   3
832 DATA 133, 114, 169,  76, 133, 112
838 DATA  96, 230, 119, 208,   2, 230
844 DATA 120, 164,  55, 200, 208,   3
850 DATA  76, 118,   0, 160,   0, 177
856 DATA 119, 201,  35, 208, 245, 186
862 DATA 189,   1,   1, 201,  62, 240
868 DATA  24, 201, 172, 240,  20, 201
874 DATA 143, 240,  16, 201, 105, 208
880 DATA 107,  32, 112,   0, 201,  44
886 DATA 208, 249, 104, 104,  76,  95
892 DATA 200, 200, 166,  40, 165,  41
898 DATA 208,   8, 160,   0, 177,  92
904 DATA 170, 200, 177,  92, 134,  92
910 DATA 133,  93, 133,  91, 177,  92
916 DATA 208,   3,  76, 235, 199,  24
922 DATA 165,  92, 105,   4, 133,  90
928 DATA 144,   2, 230,  91, 136, 177
934 DATA  90,  32, 226,   3, 133,  89
940 DATA 177, 119, 200,  32, 226,   3
946 DATA 197,  89, 208, 206, 201,   0
952 DATA 208, 235, 104, 104, 186, 189
958 DATA 255,   0, 201, 143, 208,  21
964 DATA 165, 120,  72, 165, 119,  72
970 DATA 165,  55,  72, 165,  54,  72
976 DATA 169, 141,  72, 169, 198,  72
982 DATA 169, 195,  72,  32, 205, 199
988 DATA  32,   0, 200,  76, 118,   0
994 DATA 201,  32, 240,   8, 201,  58
996 DATA 240,   4, 201,  44, 208,   2
998 DATA 169,   0,  96
```

```
.M 033A 03F0
.
.:  033A A9 47 85 71 A9 03 85 72
.:  0342 A9 4C 85 70 60 E6 77 D0
.:  034A 02 E6 78 A4 37 C8 D0 03
.:  0352 4C 76 00 A0 00 B1 77 C9
.:  035A 23 D0 F5 BA BD 01 01 C9
.:  0362 3E F0 18 C9 AC F0 14 C9
.:  036A 8F F0 10 C9 69 D0 6B 20
.:  0372 70 00 C9 2C D0 F9 68 68
.:  037A 4C 5F C8 C8 A6 28 A5 29
.:  0382 D0 08 A0 00 B1 5C AA C8
.:  038A B1 5C 86 5C 85 5D 85 5B
.:  0392 B1 5C D0 03 4C EB C7 18
.:  039A A5 5C 69 04 85 5A 90 02
.:  03A2 E6 5B 88 B1 5A 20 E2 03
.:  03AA 85 59 B1 77 C8 20 E2 03
.:  03B2 C5 59 D0 CE C9 00 D0 EB
.:  03BA 68 68 BA BD FF 00 C9 8F
.:  03C2 D0 15 A5 78 48 A5 77 48
.:  03CA A5 37 48 A5 36 48 A9 8D
.:  03D2 48 A9 C6 48 A9 C3 48 20
.:  03DA CD C7 20 00 C8 4C 76 00
.:  03E2 C9 20 F0 08 C9 3A F0 04
.:  03EA C9 2C D0 02 A9 00 60 00
```

```
            0010           .OS
            0020           .BA $33A
            0030;
            0040; ---------------------------------
            0050; - BASIC LABEL SUPPORT INTERFACE -
            0060; ---------------------------------
            0070;
            0080; SYS826 ACTIVATES THE BASIC LABEL
            0090; SUPPORT INTERFACE AND ALLOWS THE
            0100; USE OF LABELS IN BASIC FOR 'GOTO'
            0110; 'THEN' AND 'GOSUB' STATEMENTS.
            0120;
            0130; A LABEL IS PREFIXED WITH A
            0140; # CHARACTER AND TERMINATES
            0150; WITH A BLANK, COMMA OR COLON.
            0160;
            0170; BY J.HOOGSTRAAT
            0180;
            0190; BOX 20, SITE 7, SS 1
            0200; CALGARY, T2M-4N3
            0210; ALBERTA. 403-239-0900
            0220;
            0230; ---------------------------------
            0240;
            0250; HOOK UP THE BASIC LABEL INTERFACE
            0260;
033A-A947   0270HOOKUP     LDA #L,LABELS
033C-8571   0280           STA *GETCHR+1
033E-A903   0290           LDA #H,LABELS
0340-8572   0300           STA *GETCHR+2
0342-A94C   0310           LDA #$4C
0344-8570   0320           STA *GETCHR
0346-60     0330           RTS
            0340;
            0350; BASIC LABELS SUPPORT INTERFACE
            0360;
0347-E677   0370LABELS     INC *CHAD    ;DO MISSING PART
0349-D002   0380           BNE =+3      ;OF GETCHR.
034B-E678   0390           INC *CHAD+1
            0400;
034D-A437   0410           LDY *CLIN+1  ;IMMEDIAT MODE ?
034F-C8     0420           INY
0350-D003   0430           BNE LABEL1   ;NOT IMMEDIAT.
            0440;
0352-4C7600 0450NLABEL     JMP GOTCHR   ;NORMAL CONTINUE.
            0460;
0355-A000   0470LABEL1     LDY #0       ;# PREFIX ?
0357-B177   0480           LDA (CHAD),Y
0359-C923   0490           CMP #'#
035B-D0F5   0500           BNE NLABEL   ;NO PREFIX, EXIT.
            0510;
            0520; DECIDE ON WHAT ACTION TO TAKE
            0530;
035D-BA     0540CHKLAB     TSX
035E-BD0101 0550           LDA $101,X   ;GET STACK VALUE.
            0560;
```

```
0361-C93E    0570            CMP #S.THEN  ;BASIC THEN ?
0363-F018    0580            BEQ FLABEL   ;YES, FIND LABEL.
             0590;
0365-C9AC    0600            CMP #S.GOTO  ;BASIC GOTO ?
0367-F014    0610            BEQ FLABEL   ;YES, FIND LABEL.
             0620;
0369-C98F    0630            CMP #S.GSUB  ;BASIC GOSUB ?
036B-F010    0640            BEQ FLABEL   ;YES, FIND LABEL.
             0650;
036D-C969    0660            CMP #S.ONDO  ;BASIC ON.DO ?
036F-D06B    0670            BNE SKPLAB   ;NO, IT'S A LABEL.
             0680;
             0690; ON.DO ACTION
             0700;
0371-207000  0710SCOMMA      JSR GETCHR   ;FOR ON.DO
0374-C92C    0720            CMP #',      ;STATEMENT GET PAST
0376-D0F9    0730            BNE SCOMMA   ;THE COMMA.
0378-68      0740            PLA
0379-68      0750            PLA
037A-4C5FC8  0760            JMP ON.RET   ;RETURN TO ON.DO
STUFF.
             0770;
             0780; GOTO, THEN OR GOSUB ACTION
             0790;
037D-C8      0800FLABEL      INY
037E-A628    0810            LDX *BSTR    ;COPY START ADDRESS
0380-A529    0820            LDA *BSTR+1  ;OF BASIC.
0382-D008    0830            BNE CKSTAT   ;GO CHECK FIRST STAT.
             0840;
0384-A000    0850NXSTAT      LDY #0       ;SET ADDRESS OF NEXT
0386-B15C    0860            LDA (CLAD),Y         ;BASIC
STATEMENT.
0388-AA      0870            TAX
0389-C8      0880            INY
038A-B15C    0890            LDA (CLAD),Y
             0900;
038C-865C    0910CKSTAT      STX *CLAD    ;SETUP CURRENT
038E-855D    0920            STA *CLAD+1  ;BASIC LINE ADDRESS.
0390-855B    0930            STA *TMP2+1
             0940;
0392-B15C    0950            LDA (CLAD),Y         ;END OF BASIC
?
0394-D003    0960            BNE CKSTAT1  ;NO, CONTINUE.
             0970;
0396-4CEBC7  0980            JMP UNDEFD   ;UNDEF'D STATEMENT.
             0990;
0399-18      1000CKSTAT1     CLC  ;GET PAST NEXT BASIC
039A-A55C    1010            LDA *CLAD    ;LINE ADDRESS AND
BASIC
039C-6904    1020            ADC #4       ;STATEMENT NUMBER.
             1030;
039E-855A    1040            STA *TMP2    ;SAVE THE ADDRESS.
03A0-9002    1050            BCC =+3
03A2-E65B    1060            INC *TMP2+1
             1070;
03A4-88      1080            DEY
```

```
            1090;
            1100; SEARCH BASIC FOR MATCHING LABEL
            1110;
03A5-B15A   1120MATCH      LDA (TMP2),Y         ;CHECK IF THE
03A7-20E203 1130           JSR CORRECT  ;LABEL MATCHES THE
03AA-8559   1140           STA *TMP1    ;SPECIFIED LABEL.
03AC-B177   1150           LDA (CHAD),Y
03AE-C8     1160           INY
03AF-20E203 1170           JSR CORRECT
03B2-C559   1180           CMP *TMP1
03B4-D0CE   1190           BNE NXSTAT   ;NO MATCH FOUND.
            1200;
03B6-C900   1210           CMP #0       ;END OF LABEL ?
03B8-D0EB   1220           BNE MATCH    ;NO, CONTINUE
MATCHING.
            1230;
03BA-68     1240           PLA
03BB-68     1250           PLA
            1260;
03BC-BA     1270           TSX
03BD-BDFF00 1280           LDA $FF,X    ;MATCHING LABEL
FOUND.
03C0-C98F   1290           CMP #S.GSUB  ;GOSUB ACTION ?
03C2-D015   1300           BNE NOSUB    ;NO, THEN OR GOTO.
            1310;
            1320; STACK CORRECTION FOR GOSUB
            1330;
03C4-A578   1340           LDA *CHAD+1
03C6-48     1350           PHA
03C7-A577   1360           LDA *CHAD
03C9-48     1370           PHA
03CA-A537   1380           LDA *CLIN+1
03CC-48     1390           PHA
03CD-A536   1400           LDA *CLIN
03CF-48     1410           PHA
03D0-A98D   1420           LDA #$8D
03D2-48     1430           PHA
03D3-A9C6   1440           LDA #H,SUBRET
03D5-48     1450           PHA
03D6-A9C3   1460           LDA #L,SUBRET
03D8-48     1470           PHA
            1480;
03D9-20CDC7 1490NOSUB      JSR SETLAD   ;SET LINE ADD.
            1500;
03DC-2000C8 1510SKPLAB     JSR SKPSTT   ;SKIP STATEMENT.
            1520;
03DF-4C7600 1530NOPREFIX   JMP GOTCHR   ;BACK TO BASIC.
            1540;
            1550; LABEL CHARACTER CORRECTIONS
            1560;
03E2-C920   1570CORRECT    CMP #'
03E4-F008   1580           BEQ CORRECT1
03E6-C93A   1590           CMP #':
03E8-F004   1600           BEQ CORRECT1
03EA-C92C   1610           CMP #',
03EC-D002   1620           BNE CORRECT2
```

```
03EE-A900      1630CORRECT1    LDA #0
03F0-60        1640CORRECT2    RTS
               1641;
               1642; SYSTEM ADDRESS EQUATIONS
               1650;
               1660CLIN        .DI $36         ;BASIC CURR LINE NO
               1670BSTR        .DI $28         ;BASIC START ADD
               1680CHAD        .DI $77         ;BASIC CURR CHAR ADD
               1690CLAD        .DI $5C         ;BASIC CURR LINE ADD
               1700;
               1710GETCHR      .DI $70         ;GET NEXT CHAR ROUT
               1720GOTCHR      .DI $76         ;GET CURR CHAR ROUT
               1730;
               1740S.THEN      .DI $3E         ;STACK KEY 'THEN'
               1750S.GOTO      .DI $AC         ;STACK KEY 'GOTO'
               1760S.GSUB      .DI $8F         ;STACK KEY 'GOSUB'
               1770S.ONDO      .DI $69         ;STACK KEY 'ON.DO'
               1780;
               1790UNDEFD      .DI $C7EB       ;UNDEF'D STAT ERR
               1800SETLAD      .DI $C7CD       ;SET NEW LINE ADD
               1810SKPSTT      .DI $C800       ;SKIP REST OF STAT
               1820;
               1830ON.RET      .DI $C85F       ;ON.DO RETURN ADD
               1840SUBRET      .DI $C6C3       ;GOSUB RETURN ADD
               1850;
               1860TMP1        .DI $59         ;WORK SPACE
               1870TMP2        .DI $5A         ;WORK SPACE
               1880            .EN

HOOKUP    = 033A    LABELS    = 0347
NLABEL    = 0352    LABEL1    = 0355
CHKLAB    = 035D    SCOMMA    = 0371
FLABEL    = 037D    NXSTAT    = 0384
CKSTAT    = 038C    CKSTAT1   = 0399
MATCH     = 03A5    NOSUB     = 03D9
SKPLAB    = 03DC    NOPREFIX  = 03DF
CORRECT   = 03E2    CORRECT1  = 03EE
CORRECT2  = 03F0    CLIN      = 0036
BSTR      = 0028    CHAD      = 0077
CLAD      = 005C    GETCHR    = 0070
GOTCHR    = 0076    S.THEN    = 003E
S.GOTO    = 00AC    S.GSUB    = 008F
S.ONDO    = 0069    UNDEFD    = C7EB
SETLAD    = C7CD    SKPSTT    = C800
ON.RET    = C85F    SUBRET    = C6C3
TMP1      = 0059    TMP2      = 005A
```

BASIC 4.0, DOS 2.0 and The Relative Record System

Commodore is now distributing computers and disks with new operating systems. These are, of course, BASIC 4.0 and DOS 2.0. But many users that have BASIC 2.0 and DOS 1.0 are asking themselves, "Should I upgrade ?".

The new operating systems offer many advantages over the old, but there are cases where upgrading may hurt more than help. This would refer to those who 1) have a working system performing without mishap, and 2) don't do any programming of their own. More specifically, this would be businesses that have aquired equipment and a custom program(s) to perform special tasks. There are suttle differences in the new systems that may cause discrepencies once upgraded. However, this does not rule out the possibility of upgrading. Higher capacity may be necessary to maintain your systems efficiency. This would mean a "forced" upgrade to the 8050 disk, which contains the new DOS, and program modification may be required.

Serious programmers, on the other hand, should consider upgrading as seriously as their programs. Some new features are:

BASIC 4.0

1. Garbage collection time has been reduced to negligible.

2. Shifted RUN/STOP loads and runs first disk file.

3. Disk error channel read automatically into DS and DS$, same as TI and TI$ read the clock. These new variables are one reason programs may require mods. See article this issue on converting.

4. PRINT# command omits line feed after carriage return on files OPENed with a logical file number less than 128; 128 or greater still sends CRLF.

5. Disk commands now included in the BASIC. Although BASIC 2.0 could handle the disk, PRINT#ing to the command channel was somewhat clumsy.

```
BASIC 2.0                  BASIC 4.0

LOAD"prog",8               DLOAD"prog"
SAVE"1:prog",8             DSAVE"prog",d1           ;defaults to d0
VERIFY"1:prog",8           VERIFY"1:prog",8         ;no change
OPEN 2,8,6,"1:file,s,w"    DOPEN#2,"file",u8,d1,w   ;defaults unit 8,
                                                     omit w for read
                                                     no change for USR files
CLOSE 2                    DCLOSE#2,d1 ON u8        ;omit "#2" and "d1" and
                                                     close all files ON u8
LOAD"$1",8:LIST            DIRECTORY d1 or CATALOG d1
PRINT#15,"N1:title,xx"     HEADER"title",d1,ixx
   ''     "S1:prog"        SCRATCH"prog",d1
   ''     "V1"             COLLECT d1
   ''     "D1=0"           BACKUP d0 TO d1
   ''     "R1:file=1:prog" RENAME "prog",d1 TO "file",d1
   ''     "C1:prog=0:prog" COPY "prog",d0 TO "prog",d1
```

Direct access disk commands do not change in BASIC 4.0 (i.e. format is still PRINT#15,"ul", b-a, b-p, etc.) but do change in DOS 2.0. (see DOS 2.0 below). Also note that the INITIALIZE command does not get keyword priveledges in BASIC 4.0. BASIC 4.0 was designed to work best with DOS 2.0 which does automatic initializes. BASIC 4.0 also has other commands that work only with DOS 2.0:

```
APPEND#2,"file",dl
CONCAT "more data",d0 TO "existing data",dl
RECORD#2, 3000, 5
```

The APPEND# command OPENs an existing file for writing. DOS 2 positions to the end of that file such that data can be "appended".

The CONCAT command concatenates one file "TO" another existing file (SEQ type files only). Concatenating was possible with the DOS 1.0 'C'opy command, but an extra sequence of scratch and rename commands would be necessary to accomplish the above:

```
DOS2   CONCAT "more data",d0 TO "existing data",dl
DOS1   PRINT#15,"Cl:temporary=l:existing data,0:more data"
       PRINT#15,"Sl:existing data"
       PRINT#15,"Rl:existing data=l:temporary"
       PRINT#15,"S0:temporary"
```

Thanks to DOS 2.0, a single BASIC 4.0 command does it all! But remember, DOS 2.0 does the work; BASIC 4 only sends the command string to the disk command channel.

RECORD# works the DOS 2 Relative Record System. This feature of the new DOS makes it virtually indispensable!

Although the above three commands belong to BASIC 4.0, they can be simulated with BASIC 2.0, however, DOS 2.0 must be in the disk for them to work. (See article on DOS 2.0 commands from BASIC 4.0)

DOS 2.0

1. Automatic initializing.

2. "@" SAVE with replace fixed.

3. Formatting and Duplicating approximately 5 times faster.

4. Directory track and 6 other tracks have l less sector for 144 directory entries max and 664 blocks free max. It was felt that the recording density for DOS 1.0 diskette middle tracks was too high for reliability. DOS 1.0 diskettes will require converting to work on DOS 2.0 (see COPY command below). Although both diskette types can be read on either DOS, writing DOS 2 diskettes with DOS l is fatal. DOS 2 doesn't allow writing to DOS l disks.

5. RENAME command fixed.

6. COPY command now allows default characters. (e.g. COPY "fi*",d0 to "*",dl would copy all files starting with "fi" on d0 to the same name on dl. Also COPY d0 TO dl copies all files over... good for converting DOS 1.0 diskettes to DOS 2.0 diskettes)

7. "B-W" direct access commands removed; use "U2" instead. All others remain the same.

8. Sector byte zero now accessible from B-P command.

9. Error channel cleared on receiving correct command syntax. DOS 1 left the error light on until completion of a successful command (excluding LOAD"$0",8).

The Relative Record File System

Built in to the new DOS 2.0 is a filing system known as The Relative Record System. It's called Relative Record because each record is relative to another.

When a relative file (type REL on directory) is created, each record will have the same byte length. The length of the records are chosen by the user and can be any length between 1 and 254. No bytes are wasted which means, in most cases, records will span sector boundaries.

Essentially, a REL file is like an SEQ file with entry points. These entry points are stored in "side sectors" which take up space on the disk, but are transparent to the user. Each side sector can handle up to 30K with a maximum of 6 side sectors. This limits REL files to 180K, but since 2040 diskettes are 170K, a REL file could use up the whole disk. The 180K limit also applies to the 8050.

The speed of the system is incredible; maximum 3 block reads to access any record, regardless of file size.

A maximum of three REL files can be open on the disk simultaneously provided no other files are open.

The command set associated with REL files is:

```
DOPEN#
RECORD#
INPUT#
GET#
DCLOSE#
```

REL files can be COPYd, SCRATCHed, RENAMEd, etc., just like any other file. Treat them no differently than any other file, but with the same amount of respect. REL files must be DOPENd and DCLOSEd properly, using ST and DS/DS$ for file status interrogation.

Example Set-Up

First you must decide how many bytes maximum your information will need. This will be the number of bytes maximum per field plus one byte for a carriage return at the end of each field. You could save on bytes by not using carriage returns but then you must know how to split up the record into fields using MID$ upon retrieval. Once again, no more than 80 characters without a carriage return.

Once you've chosen a length or Record Size, put it in a variable, say RS. Choose a logical file number, a filename and a drive and:

```
DOPEN#6, "FILENAME",D0,L(RS)
```

You can write or read a REL file once opened. When DOPENing for the first time, the record size (RS) must be specified. After that the length need not be given. If it is, it must be the same as before else a disk error will occur and the disk will abort the open attempt.

On creating the file, the disk procedes to build records in disk RAM. These will be empty until you fill them with data. An empty record starts with CHR$(255) followed by RS-1 CHR$(0)'s. (see note 1 below)

You are now ready to store data. The DOPEN automatically positions to record number 1. After a PRINT#, the DOS will position to record 2. This means that placing multiple strings into a single record must be done using one PRINT# statement, else the strings will go into successive record numbers. Assuming R$=CHR$(13)...

```
DO        100 PRINT#6,"HELLO"R$;A$;R$;B$;R$;X%;R$;

DON'T!    100 PRINT#6,"HELLO"R$;
          110 PRINT#6,A$;R$;
          120 PRINT#6,B$;R$;
          130 PRINT#6,X%;R$;
```

This would put "HELLO" in record #1, A$ in record 2, B$ in record 3 and X% in record #4.

This could be a drawback, especially if your variables are in an array and you wish to use a loop to output all to the same record #. This brings us to the RECORD# command.

```
RECORD#LF,(RR),(PN)
```

RECORD# tells the file (LF) to position to record number RR at byte position PN within the record. The variable PN can be from 1 to 254. Variables in the RECORD# command must be enclosed in brackets. Output using a loop might look like:

```
100 PN=1
110 FOR J=1 TO NF        ;NF=number of fields
120 RECORD#6,(RR),(PN)
130 PRINT#6, FL$(J);R$;
140 PN=PN+LEN(FL$(J))+1 ;+1 for carriage rtn
150 NEXT
```

The ";R$;" in line 130 could be left off since this would be handled by BASIC.

Another method would be to concatenate the fields into one string and output:

```
100 FL$=""
110 FOR J=1 TO NF
110 FL$ = FL$+FL$(J)+R$
120 NEXT
130 PRINT#6,FL$
```

Remember... strings in memory can be length 255 max. Max REL record length is 254. If you print a string to a REL record that is longer than the record length, an OVERFLOW IN RECORD error will occur in the error channel. BUT, the first RS characters of the string will make it into the record; the rest will be lost. Should this happen, there probably won't be a carriage return at the end of the record. That doesn't matter. You will still be able to retrieve this data. As a matter of fact, carriage returns are not necessary at the end of a record, even if the data doesn't fill the record! "But why?", you ask....

REL Record Retrieval

As mentioned earlier, an empty record starts with CHR$(255) followed by RS-1 CHR$(0)'s. This is done by the DOS.

Let's say our record size is 50. If we take the characters H, E, L, L, and O, and send them into REL REC #1 starting at position 1 without a carriage return, (i.e. PRINT#6,"HELLO";) the DOS would do as it's told and put "HELLO" into REL REC #1 with no carriage return. Not too surprising, eh. However, once that's done, the DOS procedes to "pad" the remainder of the record with CHR$(0)'s; in this case 45 of 'em. The DOS is now positioned at REL REC #2.

Now let's say we position back to REL REC #1 with a RECORD#6,1 command.

The INPUT# command stops on carriage return or EOI. ST is set to 64 on EOI, otherwise ST = 0. (see note 2 for details)

If we now execute an INPUT#, the DOS sends the H, E, L, L, and O. But when the DOS sees the CHR$(0) it also sends EOI which is just as good as a carriage return. ST is set to 64 and the DOS positions automatically to the next record; REL REC #2.

The DOS would also send EOI if the character being sent was from the last position in the record. In this case the record is not full, but this means that the character in the last position doesn't have to be a CHR$(13). You can save 1 byte per record this way. For 2500 records that's almost 10 full blocks!

Back to our example, INPUT# terminated when the DOS saw CHR$(0) and sent EOI. This has further ramifications. Suppose you were to execute something like:

```
100 RECORD#6, 1, 1
110 PRINT#6,"HELLO";     ;or "HELLO";R$;
120 RECORD#6, 1, 20
130 PRINT#6,"JIM";
```

there would be CHR$(0)'s left in between "HELLO" and "JIM". "JIM" would be lost forever to INPUT#, unless you position back to it using RECORD# before INPUT#ing. Otherwise, only GET# could get it back. The DOS does not send EOI with CHR$(0) when using GET#.

Therefore, if you're anticipating blanks between data, or blank fields representing no data, it's best to construct the record in RAM first using spaces as field padding. Remember though, leading spaces will PRINT# to the disk, but INPUT# (as with INPUT) ignores them. Leading spaces include spaces at the beginning of a record and spaces immediately following a carriage return within a record.

Printover

Recall that the PRINT# command sends the characters into the record and then pads to the end of the record with CHR$(0)'s. This can be hazardous, especially if valid data exists beyond the data being sent into the record. This data would be wiped out with zeros. One more reason why you should construct the record in RAM first. You could get around this by putting the new data into the disk buffer with a "Memory-Write" routine, but that's fairly advanced and we won't cover that here.

End Of File Detection

The following routine could be used to read the entire contents of a REL file:

```
10 DOPEN#8,"FILE NAME"
20 INPUT#8,A$
30 PRINT A$
40 IF DS=50 THEN DCLOSE#8 : END
50 GOTO 20
```

On DOPENing, the file positions to record 1 and automatically positions to successive records after INPUT#ing each records' valid data. This would continue until reaching a record that hasn't yet been formatted. DS/DS$ would read 50, RECORD NOT

PRESENT. But the last record used isn't necessarily the last record formatted. (see note 1.) Storing the number of the last record used would take care of that. Give it a SEQ file of it's own and update it every time it changes using "@" write with replace.

Empty files start with CHR$(255). This gets done by the DOS initially, but if a record DELETE is done, this "empty" flag should be replaced (i.e. PRINT#lf,CHR$(255)). This available file space can then be detected for future use.

One Minor Gotcha

When a REL file is DOPENed for the first time, only one sector is allocated for data. If the file is aborted (i.e. no DCLOSE, DIRECTORY display, reset, etc.) before the DOS allocates a second data sector, the side sector information doesn't get written to the disk. That second data sector allocation forces the side sector onto the disk, but DCLOSing properly will always prevent this.

To be absolutely sure, although probably unecessary, the following routine could be used:

```
50000 DOPEN#lf,"FILE NAME",D0,L(RS)
50010 RECORD#lf,(INT(254/RS)+1)
50020 PRINT#lf,CHR$(255);
50030 DCLOSE#lf
50040 RETURN
```

The fix actually defeats its own purpose as the file is properly DCLOSEd in line 50030!

This would only have to be done once and your file is ready for I/O. Once againg, the record size (RS) need only be given in the very first DOPEN.

NOTE 1

When a REL file is created, the DOS goes looking for some RAM to use inside the disk unit; a 256 byte buffer. The first two bytes are used to store the track and sector numbers of the next sector in the REL file just like SEQ files. The remaining 254 bytes are for record space, hence the 254 byte maximum record size.

At this point the DOS fills the record space with CHR$(0)'s and puts a CHR$(255) "marker" in the first byte of each record. This byte would be a multiple of the record size. If the record size were 50, there would be CHR$(255) at bytes 2, 52, 102, 152, 202, and 252 (offset by 2 due to track & sector bytes at 0 and 1).

If REL REC #1 were currently being written to or read from, you could procede to read or write REL RECs 2, 3, 4, and 5 without any mechanical disk activity. Requesting record #6 (i.e. RECORD#lf,6,1) would return an error #50, RECORD NOT PRESENT because disk space for a 6th record hasn't yet been formatted. But 5 records don't fill the buffer completely; there are still 4 bytes left (252-255). These belong to record #6. The next PRINT# would start putting characters into these 4 bytes, at which point the DOS would find another available scetor, stick it's co-ordinates into bytes 0 and 1, and write the buffer contents onto the diskette. Now the buffer is re-formatted with the first 46 bytes of the record space belonging to record #6. A DCLOSE would write the rest of the data to disk. Requesting record #3000 would force the DOS to format all records inbetween before allowing access to the record.

NOTE 2

1. INPUT# continues to input characters from the disk until it sees a carriage return (, comma or a colon but we'll ignore these here). The next line of your program should be a check of ST. If there is more data, ST will be 0; if not, ST will be 64. (see ST table, center page)

2. INPUT# also terminates on receiving EOI (End Or Identify). EOI has a line of it's own on the IEEE bus. INPUT# checks this line. If it turns on, then no matter what character INPUT# has just received, inputting stops and ST is set to 64.

That all sounds like a lot but it really isn't. The Relative Record System is really quite easy to work. Being new, it'll take some getting used to. Once you're storing data in REL RECS, you'll hate to think how you did it any other way!

Paul Higginbottom,
Commodore U.K. Software Department

The best way I found to convert programs, was to divide all of the programs into four catagories. These are as follows:

1. Programs written entirely in BASIC, with no PEEK, POKE, USR, WAIT or SYS statements.

2. Programs written entirely in BASIC, with PEEK, POKE, USR, WAIT and/or SYS statements.

3. Programs written partly in BASIC and in machine code, with PEEK, POKE, USR, WAIT or SYS statements.

4. Programs written entirely in machine code.

First, I would like to discuss the utilities I use when converting programs. I use BASIC AID for the BASIC conversion. This has FIND, CHANGE (something the TOOLKIT lacks), NUMBER (renumber), KILL (to exit), DELETE, and BREAK (drops you into the monitor). This is a BUTTERFIELD abbreviation of our own BASIC AID (MP096, now on sale for 10 pounds! and has 16 commands - I think), but for BASIC 4.0. Also I use SUPERMON4.REL (by BUTTERFIELD/WOZNIAK/SEILER/QUITEAFEWOTHERS) which is an add-on to the monitor commands for 4.0, allowing you to hunt for code or text, disassemble, assemble, list memory in ASCII as well as hex, step through programs with trace or step, etc. I use a disk unit for conversion, but I should think a tape user could do the same sort of thing, only slower. The memory maps mentioned below have been published and are avaialable in any one of a number of current publications.

Now I will go through each catagory, one at a time.

1. This catagory shouldn't need any conversion.

2. Let's take the POKE statements first. Apart from those used to alter the screen RAM (which stay the same), usually the corresponding locations from machine to machine can be found by looking at Jim Butterfield's memory maps, which are public domain documents. The only other problems that seem to arise, are when a location has been POKEd with a certain value to make the PET function in a different way. A good example of this is the well known one that will disable the RUN/STOP key. If you understand why it works, then conversion to BASIC 4.0 is easy. All that is necessary, is to add three to the current contents of 144. On a 2.0 PET, POKE144,49 will disable the stop key. This is three more than its normal contents (46). Therefore POKE144,PEEK(144)+3 would work on either machine. Just to save you the bother, it is in fact POKE144,88 (to disable), and POKE144,85 (to enable), on BASIC 4.0 machines.

If the program is entirely BASIC, then the USR and SYS commands will not be used (unless routines from the ROMs are being used). If ROM routines are being used, again memory maps are necessary.

The WAIT command is generally only used for keyboard activity: WAIT152,1 (wait for shift key), and WAIT158,1 (wait until bit 0 of the number of keypresses in the buffer is a 1; i.e wait until an odd number of keypresses > 0). The two just mentioned would be the same on 2.0 and 4.0.

The USR command would only be used if machine code was also used, but that is not covered in this catagory.

3. All hints made in catagory 2 should be observed for this catagory as well. The USR command uses bytes 1 and 2 as an indirect address to a machine code routine. The parameter in the USR command is 'floated' and put into the first accumulator. The address POKEd into the bytes 1 and 2 will obviously not need to be changed, but the actual machine code routines, will more than likely need to be changed. The routines most commonly used by USR routines are FLPINT (floating point to integer conversion for accumulator #1, and of course INTFLP (the other way round!). The corresponding locations can again be found in the Butterfield memory maps. Use FIND/POKE1/ to find the USR command set-up statements, and work out the hex address. Use SUPERMON to disassemble the USR code, and make any changes on the screen (JMP's into ROM usually). You should also know where your program starts in memory. To find this out off of a disk unit on a BASIC 4.0 machine, the following program will do:

```
10 INPUT"FILENAME";F$:INPUT"DRIVE";DR
20 DOPEN#1,(F$),D(DR):IF DS THEN PRINTDS$:GOTO60
30 GET#1,A$,B$:N$=CHR$(0)
40 AD=ASC(A$+N$)+ASC(B$+N$)*256
50 PRINT"PROGRAM STARTS AT"AD
60 DCLOSE#1
```

You may want to add a little hex converter into the program.

To resave programs that do not start at $0401/1025, you would need to drop into the monitor (SYS4 for example). Then you would need to see where your program ends by typing in .M 002A 002A <RETURN>. The contents of 002A,002B are the end of your program (LOW, HIGH). Let us say for example that .: 002A 40 1B 40 1B 40 1B 00 00 appears. To save your program onto drive 0 on disk, you would need to type:-

```
.S "0:FILENAME",08,033A,1B41
                      !      !
         Start address       1 More than necessary,
      ($033A for example)    because the monitor
                             doesn't save the last byte!
```

4. Programs written entirely in machine code usually fall into three catagories.

(i) Those that use ROM entry points, and system variables all over the place.
(ii) Those that only use system variables (keyboard usually).
(iii) Those that manage everything by themselves.

As before, I will handle each case separately.

(i) Tiresome, because usually the whole program will have to be disassembled onto paper, and the listing gone through with a pen, whilst clutching memory maps!

(ii) Shouldn't be too much trouble, since most system variables are the same.

NOTE: $97 (151) = Keyboard Matrix coordinate on graphics keyboards,
= Unshifted ASCII on business keyboards.

(iii) Will almost certainly work. Only keyboard type may cause problems.

Editor's Note:

SUPERMON4.REL and AID4 are available from all Canadian Commodore dealers as part of the Commodore Assembler Developement Pak.

Most programs will probably fall into category 1 and won't need too much conversion at all. If a program run turns suddenly quite, check for the obvious first (i.e. STOP key disable and don't forget that nasty screen POKE).

Also remember that BASIC 4.0 has reserved two more variables besides TI, TI$ and ST. These are DS and DS$; the Disk Status. Any of these on the left of an "=" sign will cause ?SYNTAX ERROR, however, they are allowed on the right. If your date or something appears as "00, ok, 00, 00" or if a variable starts acting weird then you've probably missed one.

Programs using PRINT# should also take note. The PRINT# command no longer outputs a LINE FEED after the carriage return unless the logical file # is 128 or greater. This won't need too much attention since most programmers inhibit line feeds in their PRINT# statements by following with CHR$(13); . However, if for some reason the program depends on that line feed, simply change the file numbers to 128 or greater.

One last point to bear in mind (although chances of this one surfacing are slim to nil) is the fact that strings stored in RAM now require two more bytes of overhead. This gets you the faster garbage collection. However, if your 2.0 system packs PET's RAM to capacity with a lot of good strings (i.e. large string arrays with considerable length strings) then on 4.0 these two extra bytes per string can add up and possibly cause ?OUT OF MEMORY ERROR. Once again, highly doubtlful.

Although converting programs can be a pain, the advantages of BASIC 4.0 make it all worth it.

DOS 2.0 Commands from BASIC 2.0

I really shouldn't be telling you this because Commodore does not reccommend this combination of equipment. However, there are still owners of the original 8k PETs that have upgraded to BASIC 2.0 to work disk, but can't upgrade to BASIC 4.0 because there simply aren't enough sockets on the board. BASIC 4.0 requires one ROM installed in the $B000 socket which does not exist on original machine boards.

If you have a PET/CBM that came with BASIC 2.0 (three empty sockets), I strongly reccommend that you upgrade to BASIC 4. If you bought the machine after July 1st, 1980, then the upgrade is free, so why not! The advantages of BASIC 4.0 are listed in another article in this issue.

For those of you who don't upgrade your BASIC but do upgrade your DOS, you'll have to use the PRINT#15," command to access some of the new DOS 2.0 features. Of course all of the old DOS 1.0 commands remain the same except for "B-W"; use "U2" instead.

APPEND#

This BASIC 4 command OPENs a SEQ file for appending:

```
BASIC4:   APPEND#6, "FILENAME"        ;defaults to D0,U8
BASIC2:   OPEN 6,8,4,"0:FILENAME,A"   ;,A for append
```

CONCAT

This one's quite simply a variation of the DOS1.0 Copy command. However, if sent to DOS1.0, a dos syntax error would be placed in the error channel.

```
BASIC4:   CONCAT "FILE 2",D1 TO "FILE 1",D0
BASIC2:   PRINT#15,"C0:FILE 1=0:FILE 1,1:FILE 2"
```

RECORD#

Two commands are affected here. First you need to DOPEN a relative file, specifying the length of each relative record; 50 in the following example:

```
BASIC4:   DOPEN#6,"REL FILE NAME",L50
BASIC2:   OPEN 6,8,SA, "0:REL FILE NAME,L"+CHR$(50)
```

(See BASIC 4.0 and DOS 2.0 for more on The Relative Record system, this issue).

The RECORD# command uses the logical file number, but the BASIC 2.0 <u>artificial</u> RECORD# command uses the secondary address (SA) that you chose in the OPEN command. In BASIC 4.0 the DOPEN command choses an SA for you.

```
BASIC4:   RECORD#6, (RR), 2        ;RR is rel rec #
BASIC2:   HI = INT(RR/256) : LO = RR-HI*256
          PRINT#15,"P"CHR$(SA+96)CHR$(LO)CHR$(HI)CHR$(2)
```

The "P" stands for Position. The command tells the DOS to position to relative record number RR. The "2" tells the DOS to position to the second character of the record before reading or writing. 96 is added to SA because that's how RECORD# does it.

This program demonstrates how to use the artificial relative record commands. BASIC 4.0 users should be able to replace them with the high level syntax.

```
1000 OPEN1,8,15:REM OPEN I/O CHAN
1100 INPUT"[CS]FILENAME ";F$
1110 CLOSE2:OPEN2,8,2,F$:REM OPEN IT
1120 GOSUB9000:REM ANY ERROR ?
1130 IFEN=0THEN1200:REM NO - GO ON
1140 IFEN<>62THENGOSUB9100:END
1150 INPUT"RECORD SIZE ";RS
1160 F$=F$+",L"+CHR$(RS):GOTO1110
1200 INPUT"READ,WRITE,END  ";A$
1220 A$=MID$(A$,1,1)
1230 IFA$="R"THEN2000
1240 IFA$="W"THEN3000
1250 IFA$="E"THEN4000
1260 PRINT"[CU]";:GOTO1200
2000 :
2005 :
2010 :REM ** READ A RECORD **
2020 :
2030 INPUT"RELATIVE RECORD NUMBER ";RR
2040 INPUT"RECORD POSITION ";PN
2050 GOSUB9200:REM POSITION DISK
2060 GOSUB9000:REM CHECK THE DISK
2070 IFEN<>0THENGOSUB9100:GOTO1200
2080 INPUT#2,A$:PRINTA$:GOTO1200
3000 :
3005 :
3010 :REM ** WRITE A RECORD **
3020 :
3030 INPUT"RELATIVE RECORD NUMBER ";RR
3040 PN=1:INPUT"DATA";A$
3050 GOSUB9200:REM POSITION DISK
3060 GOSUB9000:REM CHECK THE DISK
3070 IFEN<>0THENGOSUB9100
3080 PRINT#2,A$:GOTO1200
4000 CLOSE2:CLOSE1:END
9000 :
9001 :
9002 :REM ** READ DISK MESSAGE **
9003 :
9005 INPUT#1,EN$,EM$,ET$,ES$
9010 EN=VAL(EN$):RETURN
9100 :
9101 :
9102 :REM ** PRINT DISK MESSAGE **
9103 :
9105 PRINTEN$","EM$","ET$","ES$:RETURN
9200 :
9201 :
9202 :REM ** DOES RECORD#2,(RR),(PN)
9203 :
9205 RH=INT(RR/256):RL=RR-RH*256
9210 C$="P"+CHR$(2+96)+CHR$(RL)+CHR$(RH)
9220 C$=C$+CHR$(PN)
9230 PRINT#1,C$:RETURN
```

NMI is the Non Maskable Interrupt. An interrupt is a way of telling the processor that its attention is needed for something else - right now! The regular PET interrupts are generated every 1/60th second, and are used to process the clock, keyboard, stop key and so on. These interrupts can be 'shut off' by setting the interrupt mask. There is, however, another interrupt, NMI. NMI cannot be masked - that means that it is always active.

On the old PET, the NMI line is held high (off) by the hardware. If you have an old PET, there's nothing you can do. The 6502 NMI vector is at $FFFA-$FFFB. This vector is in ROM. It points to a routine in ROM at $FCFE. This routine does a jump indirect through location $94-95 in zero page. On power-up, these locations are set to point at $C389, the BASIC warm start.

So, what can we do with NMI ? Well, it can get us out of a few sticky situations with the disk. The NMI line is available on the expansion port. The port is two connectors of 50 pins each. NMI is on the front connector, on the inside. Count forwards from the break between the two connectors. NMI is the second pin. RESET is the fourth pin. If you have a RESET button which uses an alligator clip to connect to the RESET line, just move it to this pin. Otherwise, get a mini or micro size clip and connect it to NMI. Now get another lead to ground (any of the outer pins on the connector), and connect a switch between the two. Are we ready ?

Now, when you push the RESET button, you ground the NMI line, and the 6502 jumps to the BASIC warm-start. Try it - nothing spectacular, the machine just prints READY and the cursor. OK, now let's do something silly. Try WAIT32768,1,1: Normally, that's a crash. Push NMI - READY. Neat, isn't it.

At this point, we can see that NMI can recover from some crashes - but for others (processor crashes, not infinite loops) we'll still need RESET.

But now comes the interesting stuff. We can change the NMI vector at $94,95 to anything we want. If we point it at $FD17, we can use NMI to jump to the monitor at any time. Useful for machine language programs - and all you need is an RTI instruction to get back to where you were. (You could use it to try and examine BASIC while it runs, too.)

But, that's pretty tame. OK, how about having two BASIC programs available alternately. Here's how it can be done. Set up the first BASIC program in the usual place. Set its end-of-memory pointer to 1K short of half of your memory. That is, in a 32K machine, set eom to $3C00, in 8k, to $0C00. Then copy all of zero-page to the 256 bytes just after the eom pointer of this program, and the stack to the next 256. Now, set the start of BASIC to after this stuff. For 32k, that's

$3E00. Set the eom pointer to 512 bytes short of the real end-of-memory. That would be at $7E00. Now save all of 0-page into this space, and follow it with the stack.

Now, we can write a routine (in the cassette buffer) to swap the two copies of 0-page and the stack around. You'll also have to juggle the top of the stack somewhat. When you push NMI, the PC and the stack pointer go on the stack. You'll need to push the X,Y, and accumulator, too. Then do the swap, and restore X, Y, A. Then an RTI should get things rolling. Point the NMI vector (and the copies of the NMI vector) to this routine. Once all of this is debugged, we can start one of the programs running. Then push NMI, and we swap to the other program. Push the button again, and back to the other program.

I haven't done this, so I can't promise that I didn't miss something out. If anyone does implement it (and finds a use for it!), I'd like to hear.

You can also use NMI to handle some outside device. Good luck!

Editor's Note:

Henry's concept is sound. It would require some careful thought, although not much programming to accomplish. An article on this would be a likely candidate for Best Apllication award of Volume 3.

Fun With WAIT Statements

Henry Troup, Diemaster Tool

Most of us find that the WAIT statement is of limited use. Until recently, the only use I had ever found was:

```
WAIT 59411, 8, 8
```

to wait for the cassette recorder play switch. But I did find some amusing and useful applications for WAIT.

First, a quick review.

The statement WAIT I, J, K causes the value of location I to be exclusive-OR'ed with K, and AND'ed with J. If the result is 0, the process repeats until a non-zero result is obtained. Most often, only tangible results are obtained when values of J and K are powers of 2 (1, 2, 4, 8, 16, etc.) since WAIT is a bit testing function. However testing for combinations of bits can also be useful. Be very careful though... during WAIT, the STOP is not tested. If a WAIT command is in entered, be certain a non-zero will occur or else!

Obviously, most memory locations will be of very little interest with respect to WAIT. The only locations which are of interest, in fact, are those which are affected by external events. There are two sets of these: the keyboard/ cassette/ user port/ IEEE locations in E-page, and a few in zero page. It's the zero page locations I want to talk about.

GET Loops

The classic get loop is:

```
1000 GET A$: IF A$ = "" GOTO 1000
```

which loops until a non-null input is received. The same effect can be obtained by WAITing for the keyboard buffer pointer:

```
1000 WAIT 158, 127: GET A$
```

This waits until the keyboard buffer count (decimal 158 for new ROM, 525 for old) is non-zero. It's a little harder to understand, but shorter and probably slightly faster. For experimentation, try replacing the GET command with INPUT and the 127 with 2, 4 and 8.

WAITing for a key

Very often, a GET loop is used on a "Push Any Key To Continue" basis. One interesting alternative is to use:

```
WAIT 152, 1
```

This waits for the shift key to be pushed (old ROM, 516). The advantage is that nothing is put in the keyboard buffer, so that you need not clear the buffer.

Or, if you want to have fun, try experimenting with WAITing for location 151 - key held down (515, old ROM). WAIT 151, 127, 255 will wait for any key. Specific keys are harder to WAIT for, since WAIT will only wait on one bit at a time. Remember that we're talking about un-decoded keyboard values here.

WAITing for the Clock

The real time clock occupies locations 141-143 in zero page. WAITing for one particular bit in the clock to change state will give an interesting delay effect. For example, WAIT 142, 1, 1 will wait for the rightmost bit of the second byte. This bit changes state every 256 jiffies, or 4 and a fraction seconds. WAIT 143, 1, 1 will wait till the start of the next jiffy.

While some of these are not particularly useful, playing with the WAIT statement is quite a bit of fun. If anyone finds any more useful or interesting locations, I'll be WAITing to hear from you.

8032 Control Characters

This table is a summary of the 8032 screen control functions. The ESC/RVS characters will display as lower/upper case or upper case/graphics, depending on which mode you're in. POKE59468,X (where X=12 for graphics, 14 for lower case) still changes modes without changing the gap between the lines. Notice that complimentary functions differ by 128 using CHR$(. See the Commodore BASIC 4.0 manual for details on functions.

Control Function	CHR$(value)	ESC/RVS char.
BELL	7	g
GRAPHICS	142	shift n
TEXT	14	n
SCROLL DOWN	153	shift y
SCROLL UP	25	y
SET BOTTOM	143	shift o
SET TOP	15	o
INSERT LINE	149	shift u
DELETE LINE	21	u
ERASE BEGIN	150	shift v
ERASE END	22	v
SET/CLR TAB	137	shift i
TAB	9	i

The above describes the special 80 column screen control functions. The functions can be activated two ways; by using CHR$(and the appropriate value or, preferably, by placing the appropriate character in reverse field within quotes. This is done by entering quote mode, hitting 'ESC', then 'RVS' and the character. For example, to do a Scroll Down enter quote mode and type 'ESC', 'RVS', shift & 'Y' and RETURN. 'ESC' takes you out of quote mode. If you wish to continue with more characters following the Scroll Down you'll have to do an OFF/RVS, another quote and DELete the quote. This is comparable to the cursor control characters but not quite so automatic.

Although you could use the CHR$(values, the ESC/RVS method saves bytes and will eventually become much more legible. After all, when was the last time you used a CHR$(17) to do a cursor right. (or is it a cursor up?... or is 17 delete?... no, I think it's a cursor down... I'd better check... hmm)

There is still another way to activate these functions without using PRINT. This is directly from the keyboard. But you say "There is no key on the keyboard assigned to do a scroll down or set top...". By pressing certain key combinations simultaneously, the keyboard value that is passed to the operating system will be the CHR$ value that activates the function. This information was published by Roy Busdiecker in Compute #7, but Roy found many combinations that do the same functions. I've listed only the easiest ones to remember.

Control Function	Key Combination
TEXT	BOTHShifts / "
GRAPHICS	
SCROLL DOWN	LeftShift / TAB / I
SCROLL UP	
SET BOTTOM	Shift / Z / A / L
SET TOP	Z / A / L
INSERT LINE	Shift / RVS / A / L
DELETE LINE	RVS / A / L
ERASE BEGIN	Shift / TAB / leftarrow / DEL
ERASE END	/ TAB / leftarrow / DEL
SET/CLR TAB	Shift / TAB
TAB	TAB

The two empty spaces beside TEXT and SCROLL UP are empty because they haven't been found yet. If anyone does, please let me know.

The window can also be POKEd to size. The pokes are:

Screen TOP:	224,T	where T=0 to 24
BOTTOM:	225,B	where B=T to 24
LEFT:	226,L	where L=0 to 79
RIGHT:	213,R	where R=L to 79

I'm not sure what weird or interesting effects you can get by making TOP less than BOTTOM or LEFT greater than RIGHT. This is handled by the 6845 Screen Controller chip. The 6845 does all kinds of neat things which we'll cover in a future Vol 3 Transactor.

A halt-scroll key has been added to the 8032. LIST a fairly long program and touch the ":" key. To restart scrolling, hit the left arrow key which is also the slow-scroll key.

ESCape quite simply escapes you from quote mode or insert mode (where cursor keys get displayed as reverse characters).

SYS 54386 is the command to Call the monitor rather than break to the monitor which can be done with SYS4.

POKE 144,88 disables the STOP and the clock. POKE 144,85 enables.

To clear the window hit or PRINT 2 HOMEs consecutively. If a "window reset disable" were desired, it would be easy enough to insert a pre-interrupt routine to zeroize the home count ($E8) so that the 8032 would never see 2 HOMEs in a row. The code would be LDA #0, STA $E8, JMP   (the IRQ vector). Enter it fast with these steps:

```
1.  Enter m.l.m. with SYS4
2.  Type: m 027a 027a
3.        .:  027a  a9 00 85 e8 4c 55 e4 00
4.  Now take the cursor up and change the
    IRQ vector to 027a  <RETURN>
5.  Exit the mlm with x <RETURN>
6.  Set a window with the key combination (above)
7.  Just try and clear it!
```

Best use for this would be for bulletproof INPUT. The program would set the window to one screen line with rightwindow - leftwindow = max input length. Then OPEN 1,0 (input file from the keyboard) and use INPUT#1,A$. This way, no question mark is printed and hitting RETURN with no data input doesn't break out of the program. The window could not be cleared by the user either thanks to the pre-interrupt. Wella!...failsafe keyboard input!

commodore

Commodore Canada's
Tech/News Periodical

The Transactor

	Subscription Fees		
	Canada	U.S.A.	All other foreign
The Best of The Transactor Volume 1	$10.00	$10.00	$12.00
The Transactor Volume 2	$15.00	$17.00	$19.00
The Transactor Volume 3	$10.00	$11.00	$13.00

The Best of The Transactor Volume 1 is Volume 1 back issues together in one bind.

All 12 of The Transactor Volume 2 back issues will be available for a limited time only. After supplies run out, The Best of The Transactor Vol. 2 will replace the back issues, cost unchanged.

A subscription to The Transactor Volume 3 will cover 6 issues, back issues included.

NOTE: Pre-payment required. Invoices cannot be issued for subscription fees.

NAME ____

ADDRESS ____

____

____

____

Please check here for:

Vol. 1 ___ Vol. 2 ___ Vol. 3 ___

Total amount of cheque/m.o. $______________

Please return subscription forms to:

Commodore Business Machines
3370 Pharmacy Ave.
AGINCOURT, Ontario
M1W 2K4
Atn: The Transactor

Office use only:

Added ____________
B.I.'s sent ____
Send # ______ on

PET MEMORY LOCATIONS

0000-0002	0-2	USR Jump instruction
0003	3	Current I/O Device for prompt-suppress
0005	5	Cursor position for Input & Print
0008-0009	8-9	Integer address from Basic (for SYS, GOTO, etc.)
000A-0059	10-89	Basic input buffer; # of array subscripts
005A	90	Search character (usually ':' or end-of-line)
005B	91	Scan-between-quotes flag
005C	92	Basic input buffer pointer; number of subscripts
005D	93	First-character of array-name; default DIM flag
005E	94	Type: FF=string; 00=numeric
005F	95	Type: 80=integer; 00=floating point
0060	96	'DATA' scan flag; LIST quote flag; memory flag
0061	97	Subscript flag; FNx flag
0062	98	0=input, 64=get, 152=read (flag)
0063	99	flag for trigonometric signs/comparison evaluation flag
0064	100	input flag (suppress output if negative)
0065	101	variable pseudo-stack pointer
0066	102	fixed-point pseudo-stack pointer
0067	103	dummy value (0)
0068-0070	104-112	variable ~~x~~ pseudo-stack
0071-0072	113-114	pointer for number transfer
0073-0074	115-116	number pointer
0075-0078	117-120	product staging area for multiplication
007A-007B	122-123	start of basic pointer
007C-007D	124-125	end of basic/start of varibles pointer
007E-007F	126-127	end of variables/startof arrays
0080-0081	128-129	start of available space pointer
0082-0083	130-131	bottom of strings (moving down) pointer
0084-0085	132-133	top of strings (moving down) pointer
0086-0087	134-135	limit of Basic memory pointer
0088-0089	136-137	current program line number
008A-008B	138-139	previous line number
008C-008D	140-141	previous line address (for CONT)
008E-008F	142-143	line number of DATA line
0090-0091	144-145	memory address of DATA line
0092-0093	146-147	input vector (DATA ete.)
0094-0095	148-149	current variable name
0096-0097	150-151	current variable address
0098-0099	152-153	variable pointer for current FOR/NEXT
009A	154	Y save register; new operator save
009C	156	comparison symbol accumulator: < 1 =2 > 4
009D-00A1	157-161	number work area for SQR, etc.
00A2	162	pseudo-stack yardstick (3 or 7)
00A3-00A5	163-165	jump vector for functions
00A6-00AA	166-170	numeric store area
00AB-00AF	171-175	numeric store area
00B0-00B5	176-181	primary accumulator E,M,M,M,M,S
00B6	182	Taylor series constant counter
00B7	183	accumulator high-order propogation word
00B8-00BD	184-189	secondary accumulator
00BE	190	sign comparison, primary/secondary
00BF	191	low-order rounding byte for primary acc

00C0-00C1	192-193	Cassette buffer length/Taylor constant pointer
00C2-00D9	194-217	Subrtn: Get Basic Char; C9,CA=pointer
00DA-00DE	218-222	RND storage and work area
00E0-00E1	224-225	Pointer to screen cursor line
00F2	226	Poisition of cursor on line
00E3-00E4	227-228	Utility pointer; tape buffer,scrolling
00E5-00E6	229-230	End of current program/tape end address
00E7-00E8	231-232	Tape timing constants
00E9	233	Tape buffer character
00EA	234	Direct/programmed cursor; 00=direct
00EB	235	Tape read/verify flag
00EC	236	Tape write flag
00ED	237	
00EE	238	Number of ~~x~~ characters in file name
00EF	239	Logical file number
00F0	240	File command (from OPEN)
00F1	241	Device number
00F2	242	Maximum line length (40 or 80)
00F3-00F4	243-244	Tape buffr address (start of buffer)
00F5	245	Line where cursor lives
00F6	246	Last key pushed (ASCII); buffer checksum
00F7-00F8	247-248	Tape start address/tape pointer
00F9-00FA	249-250	File name pointer
00FB	251	Number of "insert" keys pushed
00FC	252	Serial bit shift word
00FD	253	# blocks remaining to write
00FE	254	Serial word buffer
0100-010A	256-266	Binary to ASCII conversion area
010B-01FF	267-511	Stack area
0200-0202	512-514	TI and TI$ clock - jiffies
0203	515	Which key depressed: 255 = no key
0204	516	Shift key: 1 if depressed
0205-0206	517-518	Clock (unused?)
0207	519	Cassette 1 status switch
0208	520	Cassette 2 status switch
0209	521	Keyswitch PIA: STOP & RVS flags, etc.
020A	522	
020B	523	Load=0, Verify=1
020C	524	Status
020D	525	# characters in keyboard buffer
020E	526	Reverse flag
020F-0218	527-536	Keyboard buffer
0219-021A	537-538	Hardware interrupt vector
021B-021C	539-540	Break interrupt vector
021D	541	
021E	542	End-of-line-for -input pointer
0220-0221	544-545	Cursor log (row, column)
0222	546	PBD image for tape I/O
0223	547	Key image
0224	548	0=flashing cursor; else no cursor shows
0225	549	Cursor timing countdown
0226	550	Character under cursor
0227	551	Cursor blink flag
0228	552	Tape write
0229-0241	553-577	Line address high & screen line wrap table

0242-024B	578-587	Logical numbers of open files
024C-0255	588-597	Device numbers of open files
0256-025F	598-607	Command/Secondary address of open files
0260	608	Input from screen/input from keyboard
0261	609	X-save flag
0262	610	How many open files
0263	611	Input device, normally 0
0264	612	Output CMD device, normally 3
0265	613	Tape parity
0266	614	
0268	616	Pointer in filename transfer
026A	618	
026C	620	Serial bit count
026F	623	
0270	624	Tape write countdown
0271	625	Tape buffer #1 count
0272	626	Tape buffer #2 count
0273	627	Leader counter
0274	628	Flag for tape error
0275	629	0 if 1st ½-byte cntr not written
0276	630	2nd ½-byte cntr/tape error count
0277	631	
0278	632	Cassette read flag
0279	633	Checksum working word
027A-0339	634-825	Tape #1 buffer
033A-03F9	826-1017	Tape #2 buffer
0400-7FFF	1024-32767	Available RAM including expansion
8000-8FFF	32768-36863	Video RAM
9000-BFFF	36864-49151	Available ROM expansion area
C000-E077	49152-57463	Microsoft Basic
E078-E7F8	57464-59384	Keyboard/Screen/Interrupt monitor
E810	59408	PIA1 - Keyboard A register; (Direction with CRA2=1)
E811	59409	PIA1 - Keyboard A control
E812	59410	PIA1 - Keyboard B register; (Direction with CRB2=1
E813	59411	PIA1 - Keyboard B control
E820	59424	PIA2 - IEEE A register; (Direction with CRA2=1)
E821	59425	PIA2 - IEEE A control
E822	59426	PIA2 - IEEE B register; (Direction with CRB2=1)
E823	59427	PIA2 - IEEE B control
E840	59456	VIA I/O register B
E841	59457	VIA I/O register A with handshake
E842-E843	59458-59459	VIA Data Direction regs, A and B
E844-E845	59460-59461	VIA Timer 1
E846-E847	59462-59463	VIA Timer 1 latch
E848-E849	59464-59465	VIA Timer 2
E84A	59466	VIA shift register
E84B	59467	ACS: T1.T1.T2.SR.SR.SR.PB.PA
E84C	59468	PCR: B2.B2.B2.B1.A2.A2.A2.A1
E84D-E84E	59469-59470	IFR, IER: T1.T2.CB1.BC2.SR.CA1.CA2
E84F	59471	I/O Register A without handshake
F000-FFFF	61440-65535	Reset/tape/diagnostic monitor

E810	DIAGNOS. SENSE	IEEE EOI in	CASSETTE SENSE #2	#1	KEYBOARD ROW SELECT PA				59408
E811	TAPE#1 INPUT FLAG	...	SCREEN BLANK (OLD ROM) EOI OUT CA2			DDRA ACCESS	CASSETTE #1 READ CONTROL CA1		59409
E812	KEYBOARD ROW INPUT								59410
E813	RETRACE I FLAG	...	CASSETTE #1 MOTOR OUTPUT CB2			DDRB ACCESS	RETRACE INTERR. CONTROL CB1		59411

E820	IEEE INPUT								59424
E821	$\overline{ATN}$ I FLAG	...	IEEE $\overline{NDAC}$ out CB2			DDRA ACCESS	IEEE $\overline{ATN}$ in CONTROL CA1		59425
E822	IEEE OUTPUT								59426
E823	SRQ I FLAG	...	IEEE $\overline{DAV}$ out CB2			DDRB ACCESS	IEEE $\overline{SRQ}$ in CONTROL CB1		59427

E840	$\overline{DAV}$ in	$\overline{NRFD}$ in	RETRACE IN	CASS #2 MOTOR	CASSETTE OUTPUT	$\overline{ATN}$ out	NFRD out	$\overline{NDAC}$ In PB	59456
E841	PARALLEL USER PORT I/O w/H.SHAKE								59457
E842	DIRECTION REGISTER B (FOR E840)								59458
E843	DIRECTION REGISTER A (FOR E84F) (P.U.P.)								59459
E844	TIMER 1 L								59460
E845	H								59461
E846	TIMER 1 LATCH L								59462
E847	H								59463
E848	TIMER 2 L								59464
E849	H								59465
E84A	SHIFT REGISTER								59466
E84B	T1 CONTROL PB7 OUT	ONE-SHOT FREE-RUN	T2 CONTR PB6 SENSE	SHIFT REG. CONTROL			PB,PA LATCH CONTROL		59467
E84C	CB2 (P.U.P. PIN M) CONTROL IN/OUT			CB1 in CASSETTE#2 POLARITY	CA2 (graphics, lower case) Control IN/OUT			CA1 in POLARITY	59468
E84D	IRQ STATUS	T1 INT	T2 INT	CB1 CASS#2 INT	CB2 INT	SR INT	CA1 (P.U.D.B.) INT	CA2 INT	59469
E84E	ENABLE CLEAR/SET	T1 INT ENAB	T2 INT ENAB	CB1 INT ENAB	CB2 INT ENAB	SR INT ENAB	CA1 INT ENAB	CA2 INT ENAB	59470
E84F	PARALLEL USER PORT I/O (PA) PA								59471

A few routines from PET BASIC (Original ROM) Compiled by Jim Butterfield, Toronto

C2AC-C2D9	peeks at the stack for an active FOR loop
C2DA-C31C	'opens up' a space in Basic for insertion of a new linex.
C31D-C329	tests for stack-too-deep and aborts if found.
C32A-C356	check available memory space
C357-C388	sends a canned error message from C190 area, then drops into:
C389-C391	Signals 'ready'
C394-C3A9	gets a line of input, analyzes it, executes it
C3AC-C42E	handles a new line of Basic from keyboard: deletes old line, etc.
C430-C460	corrects the chaining between Basic lines after insert/delete
C462-C476	receives a line from the keyboard into the Basic buffer
C479-C48C	gets each character from keyboard
C48D-C521	looks up the keywords in an input lines and changes to "tokens"
C522-C550	searches for the loeation of a Basic line from number in 8,9
C551-C599	implements NEW command - clears everything (011/019 ROM change)
C59A-C5A7	sets the Basic pointer to start-of-program
C5A8-C647	performs LIST command
C649- C68F	executes a FOR statement
C692-C6B4	continues to build FOR vectors
C6B5-C6EF	reads and executes the next Basic statement, finds next line, etc.
C6F2-C70A	executes the Basic Command as a subroutine
C70D-C71B	performs RESTORE
C71C-C742	handles STOP, END, and BREAK procedures.
C745-C75E	performs CONT
C75F-C76D	set pause after carriage return (never called)
C770-C772	performs CLR
C775-C77D	performs RUN
C780-C79A	performs GOSUB
C79D-C7C9	performs GOTO
C7CA-C7FD	performs RETURN
C7FE-C81E	scans for start of next Basic Line
C820-C840	performs IF
C843-C862	performs ON
C863-C89A	gets a fixed-point number from Basic and stores in 8,9
C89D-C91B	performs LET
C91C-C97E	check numeric digit/move string pointer
C97F-C982	performs PRINT#
C985-C996	performs CMD
C999-CA24	performs PRINT
CA27-CA41	prints string from address in Y, A
CA44-CA76	prints a character
CA77-CA9E	handles bad input data
CA9F-CAC5	performs GET
CAC6-CADF	performs INPUT#
CAE0-CB14	performs INPUT
CB17-CB21	prompts and receives the input
CB24-CC11	performs READ
CC12-CC35	canned messages: EXTRA IGNORED; REDO FROM START
CC36-CC8F	performs NEXT
CC92-CCB5	checks Basic format, data type, flags TYPE MISMATCH
CCB8-CD38	inputs and evaluates any expression (numeric or string)
CD3a-CD9C	pushes a partially-evaluated argument to the stack
CD9D-CDB9	evalues a numeric, variable, or pi, or identifies other symbol
CDBC-CDC0	value of pi in floating binary

CDC1-CDE7	checks for special characters (+,-,",.) at start of expression
CDE8-CDF6	performs NOT function
CDF7-CE04	checks for various functions
CE05	evaluates expression within parentheses ()
CE0B	checks for right parenthesis)
CE0E	checks for left parenthesis (
CE11-CE1B	checks for comma
CE1C-CE20	prints SYNTAX ERROR and exits
CE21-CE27	sets up function for future evaluation
CE28-CE39	set up a variable name search
CE3B-CE96	checks for special variables TI, TI$, and ST
CE97-CED5	identifies and sets up function references
CED6-CF05	perform the OR and AND functions
CF06-CF6D	performs comparisons
CF6E-CF7A	sets up DIM execution
CF7B-D00E	searches for a Basic variable
D00F-D078	creates a new Basic variable
D079-D087	logs Basic variable location
D088-D098	is array pointer subroutine
D099-D09C	is 32768 in floating binary
D09D-D0B8	is floating point-to-fixed conversion for signed values
D0B9-D263	locates and/or creates arrays
D264-D277	performs FRE function
D278-D284	converts fixed point-to-floating
D285-D28A	performs POS function
D28B-D294	checks direct/indirect command, gives 'ILLEGAL DIRECT'
D295-D348	executes DEF statements and evaluation FNx
D349-D36A	performs STR$ function
D36B-D3D1	scans and sets up string elements
D3D2-D403	builds string vectors
D404-D5C3	does 'garbage collection' - discards unwanted strings
D5C4-D5D7	performs CHR$ function
D5D8-D653	performs LEFT$, RIGHT$, MID$ functions
D654-D662	performs LEN, gets string length
D663-D672	performs ASC function
D673-D684	gets a single-byte value from Basic
D685-D6C3	evaluates VAL function
D6C4-D6CF	gets two arguments (16-bit and 8-bit) from Basic
D6D0-D6E5	checks argument is in range 0-65535
D6E6-D701	performs PEEK and POKE
D702-D71D	executes WAIT statement
D71E-D890	performs addition and subtraction
D891-D8BE	contains floating-point constants
D8BF-D8FC	performs LOG function
D8FD-D95D	performs multiplication
D95F-D988	loads secondary accumulator from memory ($B8 to $BD)
D989-D9B3	test and adjust primary/secondary accumulators
D9B4-D9E0	routines to multiply or divide by 10
D9E1-DA73	performs division
DA74-DA98	loads primary accumulator from memory ($b0-$B5)
DA99-DACD	transfers primary accumulator to memory
DACE-DADD	transfers secondary accumulator to primary
DADE-DAEC	transfers primary accumulator to secondary
DAED-DAFC	rounds the primary accumulator
DAFD-DB29	extracts primary sign; performs SGN function
DB2A-DB2C	performs ABS
DB2D-DB6C	compares primary accumulator to memory

```
DB6D-DB9D Convert floating point to fixed, unsigned
DB9E-DBC4 perform INT function
DBC5-DC4F convert ASCII string to floating point
DC50-DC84 get new ASCII digit
DC94-DCAE print Basic Line number
DCAF-DDE2 convert floating point to ASCII string (at 0100 up)
DDE3-DE23 conversion constants - decimal or clock
DE24-DE2D evaluation SQR function
DE2E-DE66 evaluation of power function
DE67-DE71 negate (monadic -)
DEA0-DEF2 perform EXP function
DEF3-DF3C perform function series evaluation
DF45-DF9D perform RND calculation
DF9E      evaluate COS function
DFA5-DFED evaluate SIN function
DFEE-E019 evaluate TAN function
E048-E077 evaluate ATN function
E0B5-E0CC Basic scan program, transferred to 00C2-00D9
E0D2-E173 completion of power-on-reset; memory test, etc.
E19B-E1BB partial test for TI and TI$
E1BC-E1E0 input/read/get director
E1E1-E27C initialize I/O registers, clear screen, reset subroutine
E27D-E3C3 receive input from keyboard/screen
E3C4-E3E9 set up new screen line
E3EA-E52F output character to screen
E530-E5DA check for and perform screen scrolling
E5DB-E66A start new screen line
E66B-E67D interrupt entry
E67E-E684 interrupt return
E685-E73E hardware interrupt routine: cursor flash, tape motor, keyboar
E73F-E7AB convert keyboard matrix to ASCII
E7AC-E7B9 write-on-screen subroutine
E7D2-E7EE print canned monitor message
F0B6-F1CB IEEE-488 channel open, test, close
F1CC-F22F get input charact r from keyboard, screen  cassette, IEEE
F230-F27C output character to screen, cassette, IEEE
F27D-F2A3 restore normal I/O, clear IEEE channels
F2A4-F2AA abort (not close!) all files
F2AB-F2B7 locate logical file table entry
F2B8-F2C7 transfer file table entries to Device, Command
F2C8-F329 perform file CLOSE
F32A-F33E test stop key
F33F-F345 test if direct/indirect command for  suppressing file advice
F346-F3FE perform file LOAD
F3FF-F421 print "SEARCHING .. "
F422-F432 print "LOADING .. " or "VERIFYING"
F433-F461 get parameters for LOAD and SAVE
F462-F494 perform IEEE sequences for LOAD, SAVE, and OPEN
F495-F4BA search for specific tape header
F4BB-F4D3 perform VERIFY
F4D4-F529 get parameters  for OPEN and CLOSE
F52A-F5AD perform OPEN
F5AE-F5E2 search for any tape header
F5E3-F5EC clear tape buffer
F5ED-F64C write tape header
F64D-F666 get start & end addresses from tape header
```

```
F667-F67C Set buffer start address
F67D-F694 set tape buffer start and end pointers
F695-F69D perform SYS command
F69E-F71B perform SAVE
F71C-F735 find unused secondary address
F736-F78A update clock
F78B-F7DB set input device
F7DC-F82C set output device
F82D-F83A bump tape buffer counter
F83B-F85D wait for cassette PLAY switch
F85E-F870 test cassette switch line
F871-F87E wait for cassette RECORD and PLAY switches
F87F-F8B8 read tape initiation routine
F8B9-F8D1 write tape initiation routine
F8D2-F912 complete tape read or write
F913-F91D wait for I/O completion
F91E-F92D test stop key and abort if necessary
F92E-F95E subroutine to set tape read timing
F95F-FB0B interrupt routine for tape read
FB0C-FBE4 save memory pointer
FBE5-FBEB set ST error flag
FBEC-FBFF subroutine to count 8 serial bits per byte
FC00-FC1B subroutine to write a bit to tape
FC1C-FCFA interrupt 1 for tape write - entry at FC21
FCFB-FD15 terminate I/O and restore normal vectors
FD16-FD37 subroutine to set interrupt vector
FD38-FD47 power-on reset entry; test for diagnostic
FD48-FD7B diagnostic routine
FD7C-FD8F checksum routine
FD90-FD9A pointer advance subroutine
FD9B-FFB1 diagnostic routines
          JUMP TABLE:
FFC0      OPEN
FFC3      CLOSE
FFC6      set input device
FFC9      set output device
FFCC      restore normal I/O devices
FFCF      input character (from screen)
FFD2      output character
FFD5      LOAD
FFD8      SAVE
FFDB      VERIFY
FFDE      SYS
FFE1      test stop key
FFE4      get character from keyboard buffer
FFE7      abort all I/O channels
FFEA      update clock

FFED-FFFA turn off cassette motors
FFFA-FFFB NMI vector (mangled)
FFFC-FFFD reset vector
FFFE-FFFF interrupt vector
```

Memory locations for ROM upgrade on PET computers
Jim Butterfield, Toronto

0000-0002	0-2	USR Jump instruction
0003	3	Search character
0004	4	Scan-between-quotes flag
0005	5	Basic input buffer pointer; # subscripts
0006	6	Default DIM flag
0007	7	Type: FF=string, 00=numeric
0008	8	Type: 80=integer, 00=floating point
0009	9	DATA scan flag; LIST quote flag; memory flag
000A	10	Subscript flag; FNx flag
000B	11	0=input; 64=get; 152=read
000C	12	ATN sign flag; comparison evaluation flag
000D	13	input flag; suppress output if negative
000E	14	current I/O device for prompt-suppress
0011-0012	17-18	Basic integer address (for SYS, GOTO etc)
0013	19	Temporary string descriptor stack pointer
0014-0015	20-21	Last temporary string vector
0016-001E	22-30	Stack of descriptors for temporary strings
001F-0020	31-32	Pointer for number transfer
0021-0022	33-34	Misc. number pointer
0023-0027	35-39	Product staging area for multiplication
0028-0029	40-41	Pointer: Start-of-Basic memory
002A-002B	42-43	Pointer: End-of-Basic, Start-of-Variables
002C-002D	44-45	Pointer: End-of-Variables, Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: Bottom-of-Strings (moving down)
0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit of Basic Memory
0036-0037	54-55	Current Basic line number
0038-0039	56-57	Previous Basic line number
003A-003B	58-59	Pointer to Basic statement (for CONT)
003C-003D	60-61	Line number, current DATA line
003E-003F	62-63	Pointer to current DATA item
0040-0041	64-65	Input vector
0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/NEXT
0048	72	Y save register; new-operator save
004A.	74	Comparison symbol accumulator
004B-004C	75-76	Misc numeric work area
004D-0050	77-80	Work area; garbage yardstick
0051-0053	81-83	Jump vector for functions
0054-0058	84-88	Misc numeric storage area
0059-005D	89-93	Misc numeric storage area
005E-0063	94-99	Accumulator#1: E,M,M,M,M,S
0064	100	Series evaluation constant pointer
0065	101	Accumulator hi-order propogation word
0066-006B	102-107	Accumulator#2
006C	108	Sign comparison, primary vs. secondary
006D	109	low-order rounding byte for Acc#1
006E-006F	110-111	Cassette buffer length/Series pointer

Memory map, upgrade ROM, contd.

0070-0087	112-135	Subrtn: Get Basic Char; 77,78=pointer
0088-008C	136-140	RND storage and work area
008D-008F	141-143	Jiffy clock for TI and TI$
0090-0091	144-145	Hardware interrupt vector
0092-0093	146-147	Break interrupt vector
0094-0095	148-149	NMI interrupt vector
0096	150	Status word ST
0097	151	Which key depressed: 255=no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Correction clock
009B	155	Keyswitch PIA: STOP and RVS flags
009C	156	Timing constant buffer
009D	157	Load=0, Verify=1
009E	158	# characters in keyboard buffer
009F	159	Screen reverse flag
00A0	160	IEEE-488 output flag: FF=character waiting
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row, column)
00A5	165	IEEE-488 output character buffer
00A6	166	Key image
00A7	167	0=flashing cursor, else no cursor
00A8	168	Countdown for cursor timing
00A9	169	Character under cursor
00AA	170	Cursor blink flag
00AB	171	EOT bit received
00AC	172	Input from screen/input from keyboard
00AD	173	X save flag
00AE	174	How many open files
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally 3
00B1	177	Tape character parity
00B2	178	Byte received flag
00B4	180	Tape buffer character
00B5	181	Pointer in filename transfer
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Countdown for tape write
00BB	187	Tape buffer#1 count
00BC	188	Tape buffer#2 count
00BD	189	Write leader count; Read pass1/pass2
00BE	190	Write new byte; Read error flag
00BF	191	Write start bit; Read bit seq error
00C0	192	Pass 1 error log pointer
00C1	193	Pass 2 error correction pointer
00C2	194	0=Scan; 1-15=Count; $40=Load; $80=End
00C3	195	Checksum for read; Leader length for write
00C4-00C5	196-197	Pointer to screen line
00C6	198	Position of cursor on above line

Memory map, upgrade ROM, contd.

00C7-00C8	199-200	Utility pointer: tape buffer, scrolling
00C9-00CA	201-202	Tape end address/end of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	00=direct cursor, else programmed cursor
00CE	206	Timer 1 enabled for tape read; 00=disabled
00CF	207	EOT signal received from tape
00D0	208	Read character error
00D1	209	# characters in file name
00D2	210	Current logical file number
00D3	211	Current secondary addrs, or R/W command
00D4	212	Current device number
00D5	213	Line length (40 or 80) for screen
00D6-00D7	214-215	Start of tape buffer, address
00D8	216	Line where cursor lives
00D9	217	Last key input; buffer checksum; bit buffer
00DA-00DB	218-219	File name pointer
00DC	220	Number of keyboard INSERTs outstanding
00DD	221	Write shift word/Receive input character
00DE	222	#blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	Screen line table: hi order address & line wrap
00F9	249	Cassette#1 status switch
00FA	250	Cassette#2 status switch
00FB-00FC	251-252	Tape start address
0100-010A	256-266	Binary to ASCII conversion area
0100-013E	256-318	Tape read error log for correction
0100-01FF	256-511	Processor stack area
0200-0250	512-592	Basic input buffer
0251-025A	593-602	Logical file number table
025B-0264	603-612	Device number table
0265-026E	613-622	Secondary address, or R/W cmd, table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape#1 buffer
033A-03F9	826-1017	Tape#2 buffer
03FA-03FB	1018-1019	Vector for Machine Language Monitor
0400-7FFF	1024-32767	Available RAM including expansion
8000-8FFF	32768-36863	Video RAM
9000-BFFF	36864-49151	Available ROM expansion area
C000-E0F8	49152-57592	Microsoft Basic interpreter
E0F9-E7FF	57593-59391	Keyboard, Screen, Interrupt programs
E810-E813	59408-59411	PIA1 - Keyboard I/O
E820-E823	59424-59427	PIA2 - IEEE488 I/O
E840-E84F	59456-59471	VIA - I/O and Timers
F000-FFFF	61440-65535	Reset, tape, diagnostic monitor

E810	DIAGNOS. SENSE	IEEE EOI in	CASSETTE SENSE #2	#1	KEYBOARD ROW SELECT			PA	59408
E811	TAPE#1 INPUT FLAG	...	SCREEN BLANK (OLD ROM) EOI OUT		CA2	DDRA ACCESS	CASSETTE #1 READ CONTROL	CA1	59409
E812	KEYBOARD ROW INPUT								59410
E813	RETRACE I FLAG	...	CASSETTE #1 MOTOR OUTPUT		CB2	DDRB ACCESS	RETRACE INTERR. CONTROL	CB1	59411

E820	IEEE-INPUT								59424
E821	$\overline{ATN}$ I FLAG	...	IEEE $\overline{NDAC}$ out		CB2	DDRA ACCESS	IEEE $\overline{ATN}$ in CONTROL	CA1	59425
E822	IEEE-OUTPUT								59426
E823	$\overline{SRQ}$ I FLAG	...	IEEE $\overline{DAV}$ out		CB2	DDRB ACCESS	IEEE $\overline{SRQ}$ in CONTROL	CB1	59427

E840	$\overline{DAV}$ in	$\overline{NRFD}$ in	RETRACE IN	CASS #2 MOTOR	CASSETTE OUTPUT	$\overline{ATN}$ out	NFRD out	$\overline{NDAC}$ in PB	59456
E841	PARALLEL USER PORT I/O W/H.SHAKE								59457
E842	DIRECTION REGISTER B (FOR E840)								59458
E843	DIRECTION REGISTER A (FOR E84F) (P.U.P.)								59459
E844	TIMER 1							L	59460
E845	WRITE							H	59461
E846	TIMER 1							L	59462
E847	LATCH							H	59463
E848	TIMER 2							L	59464
E849								H	59465
E84A	SHIFT REGISTER								59466
E84B	T1 CONTROL PB7 OUT	ONE-SHOT FREE-RUN	T2 CONTR. PB6 SENSE	SHIFT REG. CONTROL			PB,PA LATCH CONTROL		59467
E84C	CB2 (P.U.P. PWM) CONTROL IN/OUT			CB1 in CASSETTE#2 POLARITY	CA2 (Graphics, Lower Case) Control IN/OUT			CA1 in POLARITY	59468
E84D	IRQ STATUS	T1 INT	T2 INT	CB1 CASS#2 INT	CB2 INT	SR INT	CA1 (P.U.P.B) INT	CA2 INT	59469
E84E	ENABLE CLEAR/SET	T1 INT ENAB	T2 INT ENAB	CB1 INT ENAB	CB2 INT ENAB	SR INT ENAB	CA1 INT ENAB	CA2 INT ENAB	59470
E84F	PARALLEL USER PORT I/O (PA)							PA	59471

Routines in Upgrade ROM   Jim Butterfield, Toronto

C000-C045	Action addresses for primary keywords
C046-C073	Action addresses for functions
C074-C091	Hierarchy and action addresses for operators
C092-C192	Table of Basic keywords
C193-C2A9	Basic messages, mostly error messages.
C2AA-C2D7	Search stack for FOR or GOSUB activity
C2D8-C31A	Open up space in memory
C31B-C327	Test: stack too deep?
C328-C354	Check available memory
C355	Send canned error message, then:
C389-C3AA	Print READY.
C3AB-C441	Handle new Basic line from keyboard
C442-C46E	Rebuild chaining of Basic lines in memory.
C46F-C494	Receive line from keyboard
C495-C52B	Change keywords to Basic tokens
C52C-C55A	Search Basic for a given Basic line number
C55B	Perform NEW, then:
C577-C5A6	Perform CLR
C5A7-C5B4	Reset Basic execution to start-of-program
C5B5-C657	Perform LIST
C658-C6FF	Perform FOR
C700-C72F	Execute Basic statement
C730-C73E	Perform RESTORE
C73F-C76A	Perform STOP and END
C76B-C784	Perform CONT
C785-C78F	Perform RUN
C790-C7AC	Perform GOSUB
C7AD-C7D9	Perform GOTO
C7DA	Perform RETURN, and perhaps:
C7F3-C80D	Perform DATA, i.e., skip rest of statement
C80E	Scan for next Basic statement
C811-C82F	Scan for next Basic line
C830	Perform IF, and perhaps:
C843-C852	Perform REM, i.e., skip rest of line
C853-C872	Perform ON
C873-C8AC	Get fixed-point number from Basic
C8AD-C927	Perform LET
C928-C936	Add ASCII digit to accumulator #1
C937-C98A	Continue to perform LET
C98B-C990	Perform PRINT#
C991-C9A4	Perform CMD
C9A5-CA1B	Perform PRINT
CA1C-CA38	Print string from memory
CA39-CA4E	Print single format character (space, cursor-right, ?)
CA4F-CA7C	Handle bad input data
CA7D-CAA6	Perform GET
CAA7-CAC0	Perform INPUT#
CAC1-CAF9	Perform INPUT
CAFA-CB06	Prompt and receive input
CB07-CBFB	Perform READ; common routines used by INPUT and GET
CBFC-CC1F	Messages: EXTRA IGNORED, REDO FROM START
CC20-CC78	Perform NEXT
CC79-CC9E	Checks data type, prints TYPE MISMATCH
CC9F	Inputs & evaluates any expression (numeric or string)
CDEC	Evaluate expression within parentheses ()
CDF2	Check right parenthesis)
CDF5	Check left parenthesis (
CDF8-CE02	Check for comma
CE03-CE07	Print SYNTAX ERROR and exit
CE08-CE0E	Set up function for future evaluation
CE0F-CE88	Search for variable name
CE89-CEC7	Identify and set up function references
CEC8	Perform OR
CECB-CEF7	Perform AND
CEF8-CF5F	Perform comparisons, string or numeric
CF60-CF6C	Perform DIM
CF6D-CFF6	Search for variable location in memory
CFF7-D000	Check if ASCII character is alphabetic
D001-D077	Create new Basic variable
D078-D088	Array pointer subroutine
D089-D08C	32768 in floating binary
D08D-D0AB	Evaluate expression for positive integer
D0AC-D227	Find or create array
D228-D258	Compute array subscript size
D259	Perform FRE
D26D-D279	Converts fixed-point to floating-point
D27A-D27F	Perform POS
D280-D28C	Check if direct command, print ILLEGAL DIRECT
D28D-D2BA	Perform DEF
D2BB-D2CD	Check FNx syntax
D2CE-D33E	Evaluate FNx
D33F-D34E	Perform STR$
D34F-D360	Calculate string vector
D361-D3CD	Scan and set up string
D3CE-D3FF	Subroutine to build string vector
D400-D496	Garbage collection subroutine
D497-D4DF	Check for most eligible string for collection
D4E0-D516	Collect a string
D517-D553	Perform string concatenation
D554-D57C	Build string into memory
D57D-D5B4	Discard unwanted string
D5B5-D5C5	Clean the descriptor stack
D5C6-D5D9	Perform CHR$
D5DA-D605	Perform LEFT$
D606-D610	Perform RIGHT$
D611-D63A	Perform MID$
D63B-D655	Pull string function parameters from stack
D656-D65B	Perform LEN
D65C-D664	Move from string-mode to numeric-mode
D665-D674	Perform ASC
D675-D686	Input byte parameter
D687-D6C5	Perform VAL
D6C6-D6D1	Get two parameters for POKE or WAIT
D6D2-D6E7	Convert floating-point to fixed-point
D6E8-D706	Perform PEEK
D707-D70F	Perform POKE
D710-D72B	Perform WAIT
D72C-D732	Add 0.5 to accumulator#1
D733-D744	Perform subtraction
D745-D76D	Microsoft joke

```
D76E-D852 Perform addition
D853-D889 Complement accumulator#1
D88A-D88E Print OVERFLOW and exit
D88F-D8C7 Multiply-a-byte subroutine
D8C8-D8F5 Function constants:  1, SQR(.5), SQR(2), -0.5, etc.
D8F6      Perform LOG
D937-D964 Perform multiplication
D965-D997 Multiply-a-bit subroutine
D998-D9C2 Load accumulator #2 from  memory
D9C3-D9DF Test and adjust accumulators #1 and #2
D9E0-D9ED Handle overflow and underflow
D9EE-DA04 Multiply by 10
DA05-DA09 10 in floating binary
DA0A      Divide by 10
DA13      Perform divide-into
DA1E-DAAD Perform divide-by
DAAE-DAD2 Load Accumulator #1 from memory
DAD3-DB07 Store accumulator #1 into memory
DB08-DB17 Copy accumulator #2 into accumulator #1
DB18-DB26 Copy accumulator #1 into accumulator #2
DB27-DB36 Round off accumulator #1
DB37-DB44 Compute SGN value of accumulator #1
DB45-DB63 Perform SGN
DB64-DB66 Perform ABS
DB67-DBA6 Compare accumulator #1 to memory
DBA7-DBD7 Convert floating-point to fixed-point
DBD8-DBFE Perform INT
DBFF-DC89 Convert string to floating-point
DC8A-DCBE Get new ASCII digit
DCBF-DCCD String conversion constants:  99999999, 999999999, 1E+9
DCCE      Print IN, followed by:
DCD9-DCE8 Print Basic line number
DCE9-DE1C Convert number or TI$ to ASCII
DE1D-DE5D Constants for numeric conversion
DE5E      Perform SQR
DE68      Perform power function
DEA1-DEAB Perform negation
DEAC-DED9 Constants for string evaluation
DEDA-DF2C Perform EXP
DF2D-DF76 Function series evaluation subroutines
DF77-DF7E Manipulation constants for RND
DF7F-DFD7 Perform RND
DFD8      Perform COS
DFDF-E027 Perform SIN
E028-E053 Perform TAN
E054-E08B Constants for trig evaluation: pi/2, 2*pi, .25, etc.
E08C-E0BB Perform ATN
E0BC-E0F8 Constants for ATN series evaluation
E0F9-E110 Subroutine to be moved to zero page ($70 to $87)
E111-E115 Initial RND seed
E116-E1B6 Initialize Basic system
E1B7-E1DD Messages:  BYTES FREE, ### COMMODORE BASIC ###
E1DE      Initialize I/O registers, and:
E229      Clear screen, and:
E257-E284 Home cursor
```

```
E285-E2F3 Input from screen or keyboard; wait for input completion
E2F4-E33E Input from screen
E33F-E34B Test for quotation mark and reverse quote-flag
E34C-E38A Set up screen print parameters
E38B-E395 Prevent 80-character line from getting any longer
E396-E3B3 Extend 40-character line to 80 characters
E3B4-E3D7 Back into the previous line (via DEL or CURSOR LEFT key)
E3D8-E518 Handle ASCII character for screen output
E519-E53E Go to next line on screen
E53F-E5B9 Scroll the screen
E5BA-E61A Open a line on the screen (via INSERT key)
E61B-E62D Main interrupt entry point
E62E-E6E9 Hardware interrupt:  service clock, keyboard, cassettes
E6EA-E6F7 Print character on screen
E6F8-E769 Table:  decoder for keyboard matrix
E76A-E796 MLM subroutine:  output hex digits
E797-E7A6 MLM subroutine:  swap TMPO and TMP2
E7A7-E7F6 MLM subroutine:  input hex digits
E7F7-E7FF MLM subroutine:  print ?
F000-F0B5 Monitor messages, mostly for Input/Output
F0B6      Set up IEEE for Listen, Talk, etc.
F0EE-F127 Send character to IEEE-488 bus
F128-F135 Output character immediate mode to IEEE-488
F136-F155 Send errors:  WRITE TIMEOUT, DEVICE NOT PRESENT, etc.
F156-F163 Send canned I/O message
F164-F16E Send immediate Listen command, then secondary address
F16F-F17E Output character deferred mode to IEEE-488
F17F-F18B Drop IEEE channel:  send Unlisten or Untalk
F18C-F1D0 Input character from IEEE-488 bus
F1D1-F1E0 GET a character
F1E1-F231 INPUT from any device
F232-F26D OUTPUT a character to any device
F26E      Abort all files, and;
F284-F28C Restore normal I/O devices
F28D-F2A8 Find file table entry; set parameters from file table
F2A9-F300 Perform CLOSE
F301-F30E Test stop key
F30F-F314 Action stop key
F315-F31C Send message if direct mode
F31D-F321 Test if direct mode
F322-F3C1 Perform program loading
F3C2-F409 Perform LOAD
F40A-F43D Subroutines:  Print SEARCHING... ; Print LOADING or VERIFYING
F43E-F45F Get Load or Save parameters
F460-F465 Get a byte parameter
F466-F493 Send program name to IEEE-488 bus
F494-F4B6 Find a specific tape header
F4B7-F4CD Perform VERIFY
F4CE-F50D Get parameters for OPEN, CLOSE
F50E-F515 Abort calling subroutine if end-of-line (default parameters)
F516-F520 Confirm comma, else send SYNTAX ERROR
F521-F5A5 Perform OPEN
F5A6-F5D9 Find any tape header
F5DA-F63B Write tape header
```

```
F63C-F655 Get start & end program addresses from tape header
F656-F66B Set cassette buffer address according to device number
F66C-F683 Set tape start & end addresses from buffer address
F684-F68C Perform CMD
F68D-F69D Set tape start & end addresses from Basic pointers
F69E-F728 Perform SAVE
F729-F76C Update TI and TI$, and copy STOP key to work area
F76D-F76F TI constant:  limit of clock (24 hours)
F770-F7BB Set input device
F7BC-F805 Set output device
F806-F811 Advance tape buffer pointer (for INPUT#, GET#, and PRINT#)
F812-F834 Wait:  PRESS PLAY ON TAPE#
F835-F846 Test if cassette button(s) pressed
F847-F854 Wait:  PRESS PLAY & RECORD ON TAPE#
F855      Initiate tape read
F886-F8E5 Initiate tape write
F8E6-F8EF Test for I/O interrupt completion
F8F0-F8FF Test stop key
F900-F930 Set expected timing for next input bit from tape
F931      Interrupt entry:  Read tape bits
FA57-FB75 Store received tape characters
FB76-FB7E Set tape read/write address back to starting point
FB7F-FB83 Flag I/O error into ST
FB84-FB92 Reset 8-count and flags for a new byte
FB93-FBAE Write a transition to cassette tape
FBAF-FC40 Write interrupt 2:  write data to tape
FC41-FC7A Write interrupt 1:  write tape shorts (leader)
FC7B-FC95 Terminate tape:  restore normal interrupt vector
FC96-FCA5 Set interrupt vector from table
FCA6-FCB3 Turn off cassette motors
FCB4-FCC5 Perform running checksum calculation
FCC6-FCD0 Advance read/write pointer
FCD1-FCFD Power-on reset entry point
FCFE-FD00 NMI interrupt entry point
FD01-FD10 Table of interrupt vectors
FD11-FFB0 Machine Language Monitor (MLM) - see Commodore documentation
FFB1-FFBF CBM copyright statement
 *****Jump Table*****
FFC0 OPEN
FFC3 CLOSE
FFC6 Set input device
FFC9 Set output device
FFCC Restore default I/O devices
FFCF Input character
FFD2 Output character
FFD5 LOAD
FFD8 SAVE
FFDB VERIFY
FFDE SYS
FFE1 Test stop key
FFE4 Get character
FFE7 Abort all I/O activity
FFEA Clock update
FFF0-FFF9 unused
FFFA-FFFF Hardware vectors:  NMI, Reset, Interrupt
```

Memory map: Original ROM to Upgrade ROM

Jim Butterfield

To identify a function of PET's original ROM, and/or convert it to the equivalent upgrade ROM location, use this table.

All addresses are given in hexadecimal.

OLD ADDRS	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
0000:	0000	0001	0002	000E	**	**	**	**
0008:	0011	0012	0200	0201	0202	0203	0204	0205
0010:	0206	0207	0208	0209	020A	020B	020C	020D
0018:	020E	020F	0210	0211	0212	0213	0214	0215
0020:	0216	0217	0218	0219	021A	021B	021C	021D
0028:	021E	021F	0220	0221	0222	0223	0224	0225
0030:	0226	0227	0228	0229	022A	022B	022C	022D
0038:	022E	022F	0230	0231	0232	0233	0234	0235
0040:	0236	0237	0238	0239	023A	023B	023C	023D
0048:	023E	023F	0240	0241	0242	0243	0244	0245
0050:	0246	0247	0248	0249	024A	024B	024C	024D
0058:	024E	024F	0003	0004	0005	0006	0007	0008
0060:	0009	000A	000B	000C	000D	0013	0014	0015
0068:	0016	0017	0018	0019	001A	001B	001C	001D
0070:	001E	001F	0020	0021	0022	0023	0024	0025
0078:	0026	0027	0028	0029	002A	002B	002C	002D
0080:	002E	002F	0030	0031	0032	0033	0034	0035
0088:	0036	0037	0038	0039	003A	003B	003C	003D
0090:	003E	003F	0040	0041	0042	0043	0044	0045
0098:	0046	0047	0048	0049	004A	004B	004C	004D
00A0:	004E	004F	0050	0051	0052	0053	0054	0055
00A8:	0056	0057	0058	0059	005A	005B	005C	005D
00B0:	005E	005F	0060	0061	0062	0063	0064	0065
00B8:	0066	0067	0068	0069	006A	006B	006C	006D
00C0:	006E	006F	0070	0071	0072	0073	0074	0075
00C8:	0076	0077	0078	0079	007A	007B	007C	007D
00D0:	007E	007F	0080	0081	0082	0083	0084	0085
00D8:	0086	0087	0088	0089	008A	008B	008C	**
00E0:	00C4	00C5	00C6	00C7	00C8	00C9	00CA	00CB
00E8:	00CC	00B4	00CD	00CE	00CF	00D0	00D1	00D2
00F0:	00D3	00D4	00D5	00D6	00D7	00D8	00D9	00FB
00F8:	00FC	00DA	00DB	00DC	00DD	00DE	00DF	**
0200:	008D	008E	008F	0097	0098	0099	009A	00F9
0208:	00FA	009B	009C	009D	0096	009E	009F	026F
0210:	0270	0271	0272	0273	0274	0275	0276	0277
0218:	0278	0090	0091	0092	0093	00A0	00A1	**
0220:	00A3	00A4	00A5	00A6	00A7	00A8	00A9	00AA
0228:	00AB	00E0	00E1	00E2	00E3	00E4	00E5	00E6
0230:	00E7	00E8	00E9	00EA	00EB	00EC	00ED	00EE
0238:	00EF	00F0	00F1	00F2	00F3	00F4	00F5	00F6
0240:	00F7	00F8	0251	0252	0253	.. etc.		
0260:	00AC	00AD	00AE	00AF	00B0	00B1	00B2	**
0268:	00B5	**	**	**	00B7	**	**	00B9
0270:	00BA	00BB	00BC	00BD	00BE	00BF	00C0	00C1

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
0	00		@	end-line	BRK	0
1	01		A		ORA(I,X)	1
2	02		B			2
3	03		C			3
4	04		D			4
5	05		E		ORA Z	5
6	06		F		ASL Z	6
7	07		G			7
8	08		H		PHP	8
9	09		I		ORA #	9
10	0A		J		ASL A	10
11	0B		K			11
12	0C		L			12
13	0D	car ret	M		ORA	13
14	0E		N		ASL	14
15	0F		O			15
16	10		P		BPL	16
17	11	cur down	Q		ORA(I),Y	17
18	12	reverse	R			18
19	13	cur home	S			19
20	14	delete	T			20
21	15		U		ORA Z,X	21
22	16		V		ASL Z,X	22
23	17		W			23
24	18		X		CLC	24
25	19		Y		ORA Y	25
26	1A		Z			26
27	1B		[			27
28	1C		\			28
29	1D	cur right	]		ORA X	29
30	1E		↑		ASL X	30
31	1F		←			31
32	20	space	space	space	JSR	32
33	21	!	!	!	AND(I,X)	33
34	22	"	"	"		34
35	23	#	#	#		35
36	24	$	$	$	BIT Z	36
37	25	%	%	%	AND Z	37
38	26	&	&	&	ROL Z	38
39	27	'	'	'		39
40	28	(	(	(	PLP	40
41	29	)	)	)	AND #	41
42	2A	*	*	*	ROL A	42
43	2B	+	+	+		43
44	2C	,	,	,	BIT	44
45	2D	-	-	-	AND	45
46	2E	.	.	.	ROL	46
47	2F	/	/	/		47
48	30	Ø	Ø	Ø	BMI	48
49	31	1	1	1	AND(I),Y	49
50	32	2	2	2		50
51	33	3	3	3		51
52	34	4	4	4		52
53	35	5	5	5	AND Z,X	53
54	36	6	6	6	ROL Z,X	54
55	37	7	7	7		55
56	38	8	8	8	SEC	56
57	39	9	9	9	AND Y	57
58	3A	:	:	:	CLI	58
59	3B	;	;	;		59
60	3C	<	<	<		60
61	3D	=	=	=	AND X	61
62	3E	>	>	>	ROL X	62
63	3F	?	?	?		63

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
64	40	@	─	@	RTI	64
65	41	A	♠,a	A	EOR(I,X)	65
66	42	B	│,b	B		66
67	43	C	─,c	C		67
68	44	D	─,d	D		68
69	45	E	─,e	E	EOR Z	69
70	46	F	─,f	F	LSR Z	70
71	47	G	│,g	G		71
72	48	H	│,h	H	PHA	72
73	49	I	╮,i	I	EOR #	73
74	4A	J	╰,j	J	LSR A	74
75	4B	K	╯,k	K		75
76	4C	L	└,l	L	JMP	76
77	4D	M	╲,m	M	EOR	77
78	4E	N	╱,n	N	LSR	78
79	4F	O	┌,o	O		79
80	50	P	┐,p	P	BVC	80
81	51	Q	●,q	Q	EOR(I),Y	81
82	52	R	─,r	R		82
83	53	S	♥,s	S		83
84	54	T	│,t	T		84
85	55	U	╭,u	U	EOR Z,X	85
86	56	V	╳,v	V	LSR Z,X	86
87	57	W	○,w	W		87
88	58	X	♣,x	X	CLI	88
89	59	Y	│,y	Y	EOR Y	89
90	5A	Z	♦,z	Z		90
91	5B	[	┼	[		91
92	5C	\	▌	\		92
93	5D	]	│	]	EOR X	93
94	5E	↑	π,▒	↑	LSR X	94
95	5F	←	◥,▨	←		95
96	60		□		RTS	96
97	61		▌		ADC(I,X)	97
98	62		▄			98
99	63		▔			99
100	64		▁			100
101	65		▏		ADC Z	101
102	66		▒		ROR Z	102
103	67		▕			103
104	68		▄		PLA	104
105	69		◤,▨		ADC #	105
106	6A		▕		ROR A	106
107	6B		├			107
108	6C		▗		JMP(I)	108
109	6D		└		ADC	109
110	6E		┐		ROR	110
111	6F		▂			111
112	70		┌		BVS	112
113	71		┴		ADC(I),Y	113
114	72		┬			114
115	73		┤			115
116	74		▎			116
117	75		▍		ADC Z,X	117
118	76		▐		ROR Z,X	118
119	77		▔			119
120	78		▀		SEI	120
121	79		▃		ADC Y	121
122	7A		□,☑			122
123	7B		▖			123
124	7C		▝			124
125	7D		┘		ADC X	125
126	7E		▘		ROR X	126
127	7F		▚			127

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
128	80		r-@	END		128
129	81		r-A	FOR	STA(I,X)	129
130	82		r-B	NEXT		130
131	83		r-C	DATA		131
132	84		r-D	INPUT#	STY Z	132
133	85		r-E	INPUT	STA Z	133
134	86		r-F	DIM	STX Z	134
135	87		r-G	READ		135
136	88		r-H	LET	DEY	136
137	89		r-I	GOTO		137
138	8A		r-J	RUN	TXA	138
139	8B		r-K	IF		139
140	8C		r-L	RESTORE	STY	140
141	8D	car ret	r-M	GOSUB	STA	141
142	8E		r-N	RETURN	STX	142
143	8F		r-O	REM		143
144	90		r-P	STOP	BCC	144
145	91	cur up	r-Q	ON	STA(I),Y	145
146	92	rvs off	r-R	WAIT		146
147	93	clear	r-S	LOAD		147
148	94	insert	r-T	SAVE	STY Z,X	148
149	95		r-U	VERIFY	STA Z,X	149
150	96		r-V	DEF	STX Z,Y	150
151	97		r-W	POKE		151
152	98		r-X	PRINT#	TYA	152
153	99		r-Y	PRINT	STA Y	153
154	9A		r-Z	CONT	TXS	154
155	9B		r-[	LIST		155
156	9C		r-\	CLR		156
157	9D	cur left	r-]	CMD	STA X	157
158	9E		r-↑	SYS		158
159	9F		r-←	OPEN		159
160	A0		■	CLOSE	LDY #	160
161	A1	▌	r-!	GET	LDA(I,X)	161
162	A2	▄	r-"	NEW	LDX #	162
163	A3	▔	r-#	TAB(		163
164	A4	▁	r-$	TO	LDY Z	164
165	A5	▏	r-%	FN	LDA Z	165
166	A6	▒	r-&	SPC(	LDX Z	166
167	A7	▕	r-'	THEN		167
168	A8	▄	r-(	NOT	TAY	168
169	A9	◤,▨	r-)	STEP	LDA #	169
170	AA	▕	r-*	+	TAX	170
171	AB	├	r-+	-		171
172	AC	▗	r-,	*	LDY	172
173	AD	└	r--	/	LDA	173
174	AE	┐	r-.	↑	LDX	174
175	AF	▂	r-/	AND		175
176	B0	┌	r-Ø	OR	BCS	176
177	B1	┴	r-1	>	LDA(I),Y	177
178	B2	┬	r-2	=		178
179	B3	┤	r-3	<		179
180	B4	▎	r-4	SGN	LDY Z,X	180
181	B5	▍	r-5	INT	LDA Z,X	181
182	B6	▐	r-6	ABS	LDX Z,Y	182
183	B7	▔	r-7	USR		183
184	B8	▀	r-8	FRE	CLV	184
185	B9	▃	r-9	POS	LDA Y	185
186	BA	□,☑	r-:	SQR	TSX	186
187	BB	▖	r-;	RND		187
188	BC	▝	r-<	LOG	LDY X	188
189	BD	┘	r-=	EXP	LDA X	189
190	BE	▘	r->	COS	LDX Y	190
191	BF	▚	r-?	SIN		191

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
192	C0	─	↑	TAN	CPY #	192
193	C1	♠,a		ATN	CMP(I),X	193
194	C2	│,b		PEEK		194
195	C3	─,c		LEN		195
196	C4	─,d		STR$	CPY Z	196
197	C5	─,e		VAL	CMP Z	197
198	C6	─,f		ASC	DEC Z	198
199	C7	│,g		CHR$		199
200	C8	│,h		LEFT$	INY	200
201	C9	╮,i		RIGHT$	CMP #	201
202	CA	╰,j		MID$	DEX	202
203	CB	╯,k				203
204	CC	└,l			CPY	204
205	CD	╲,m			CMP	205
206	CE	╱,n			DEC	206
207	CF	┌,o	Reverse of ASCII			207
208	D0	┐,p			BNE	208
209	D1	●,q			CMP(I),Y	209
210	D2	─,r				210
211	D3	♥,s				211
212	D4	│,t				212
213	D5	╭,u			CMP Z,X	213
214	D6	╳,v			DEC Z,X	214
215	D7	○,w				215
216	D8	♣,x			CLD	216
217	D9	│,y			CMP Y	217
218	DA	♦,z				218
219	DB	┼				219
220	DC	▌				220
221	DD	│			CMP X	221
222	DE	π,▒			DEC X	222
223	DF	◥,▨	↓			223
224	E0		█		CPX #	224
225	E1		▐		SBC(I),X	225
226	E2		▀			226
227	E3		▆			227
228	E4		▇		CPX Z	228
229	E5		▊		SBC Z	229
230	E6		▒		INC Z	230
231	E7		▋			231
232	E8		▀		INX	232
233	E9		◢,▨		SBC #	233
234	EA		▋		NOP	234
235	EB		┤			235
236	EC		▛		CPX	236
237	ED		┐		SBC	237
238	EE		└		INC	238
239	EF		▆			239
240	F0		┘		BEQ	240
241	F1		┬		SBC(I),Y	241
242	F2		┴			242
243	F3		├			243
244	F4		▊			244
245	F5		▋		SBC Z,X	245
246	F6		▌		INC Z,X	246
247	F7		▆			247
248	F8		▄		SED	248
249	F9		▀		SBC Y	249
250	FA		■,☑			250
251	FB		▜			251
252	FC		▙			252
253	FD		┌		SBC X	253
254	FE		▟		INC X	254
255	FF		▞			255

Memory locations for ROM upgrade on PET computers

Jim Butterfield, Toronto

0000-0002	0-2	USR Jump instruction
0003	3	Search character
0004	4	Scan-between-quotes flag
0005	5	Basic input buffer pointer; # subscripts
0006	6	Default DIM flag
0007	7	Type: FF=string, 00=numeric
0008	8	Type: 80=integer, 00=floating point
0009	9	DATA scan flag; LIST quote flag; memory flag
000A	10	Subscript flag; FNx flag
000B	11	0=input; 64=get; 152=read
000C	12	ATN sign flag; comparison evaluation flag
000D	13	input flag; suppress output if negative
000E	14	current I/O device for prompt-suppress
0011-0012	17-18	Basic integer address (for SYS, GOTO etc)
0013	19	Temporary string descriptor stack pointer
0014-0015	20-21	Last temporary string vector
0016-001E	22-30	Stack of descriptors for temporary strings
001F-0020	31-32	Pointer for number transfer
0021-0022	33-34	Misc. number pointer
0023-0027	35-39	Product staging area for multiplication
0028-0029	40-41	Pointer: Start-of-Basic memory
002A-002B	42-43	Pointer: End-of-Basic, Start-of-Variables
002C-002D	44-45	Pointer: End-of-Variables, Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: Bottom-of-Strings (moving down)
0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit of Basic Memory
0036-0037	54-55	Current Basic line number
0038-0039	56-57	Previous Basic line number
003A-003B	58-59	Pointer to Basic statement (for CONT)
003C-003D	60-61	Line number, current DATA line
003E-003F	62-63	Pointer to current DATA item
0040-0041	64-65	Input vector
0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/NEXT
0048	72	Y save register; new-operator save
004A.	74	Comparison symbol accumulator
004B-004C	75-76	Misc numeric work area
004D-0050	77-80	Work area; garbage yardstick
0051-0053	81-83	Jump vector for functions
0054-0058	84-88	Misc numeric storage area
0059-005D	89-93	Misc numeric storage area
005E-0063	94-99	Accumulator#1: E,M,M,M,M,S
0064	100	Series evaluation constant pointer
0065	101	Accumulator hi-order propogation word
0066-006B	102-107	Accumulator#2
006C	108	Sign comparison, primary vs. secondary
006D	109	low-order rounding byte for Acc#1
006E-006F	110-111	Cassette buffer length/Series pointer

Memory map, upgrade ROM, contd.

0070-0087	112-135	Subrtn: Get Basic Char; 77,78=pointer
0088-008C	136-140	RND storage and work area
008D-008F	141-143	Jiffy clock for TI and TI$
0090-0091	144-145	Hardware interrupt vector
0092-0093	146-147	Break interrupt vector
0094-0095	148-149	NMI interrupt vector
0096	150	Status word ST
0097	151	Which key depressed: 255=no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Correction clock
009B	155	Keyswitch PIA: STOP and RVS flags
009C	156	Timing constant buffer
009D	157	Load=0, Verify=1
009E	158	# characters in keyboard buffer
009F	159	Screen reverse flag
00A0	160	IEEE-488 output flag: FF=character waiting
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row, column)
00A5	165	IEEE-488 output character buffer
00A6	166	Key image
00A7	167	0=flashing cursor, else no cursor
00A8	168	Countdown for cursor timing
00A9	169	Character under cursor
00AA	170	Cursor blink flag
00AB	171	EOT bit received
00AC	172	Input from screen/input from keyboard
00AD	173	X save flag
00AE	174	How many open files
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally 3
00B1	177	Tape character parity
00B2	178	Byte received flag
00B4	180	Tape buffer character
00B5	181	Pointer in filename transfer
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Countdown for tape write
00BB	187	Tape buffer#1 count
00BC	188	Tape buffer#2 count
00BD	189	Write leader count; Read pass1/pass2
00BE	190	Write new byte; Read error flag
00BF	191	Write start bit; Read bit seq error
00C0	192	Pass 1 error log pointer
00C1	193	Pass 2 error correction pointer
00C2	194	0=Scan; 1-15=Count; $40=Load; $80=End
00C3	195	Checksum for read; Leader length for write
00C4-00C5	196-197	Pointer to screen line
00C6	198	Position of cursor on above line

Memory map, upgrade ROM, contd.

00C7-00C8	199-200	Utility pointer: tape buffer, scrolling
00C9-00CA	201-202	Tape end address/end of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	00=direct cursor, else programmed cursor
00CE	206	Timer 1 enabled for tape read; 00=disabled
00CF	207	EOT signal received from tape
00D0	208	Read character error
00D1	209	# characters in file name
00D2	210	Current logical file number
00D3	211	Current secondary addrs, or R/W command
00D4	212	Current device number
00D5	213	Line length (40 or 80) for screen
00D6-00D7	214-215	Start of tape buffer, address
00D8	216	Line where cursor lives
00D9	217	Last key input; buffer checksum; bit buffer
00DA-00DB	218-219	File name pointer
00DC	220	Number of keyboard INSERTs outstanding
00DD	221	Write shift word/Receive input character
00DE	222	#blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	Screen line table: hi order address & line wrap
00F9	249	Cassette#1 status switch
00FA	250	Cassette#2 status switch
00FB-00FC	251-252	Tape start address
0100-010A	256-266	Binary to ASCII conversion area
0100-013E	256-318	Tape read error log for correction
0100-01FF	256-511	Processor stack area
0200-0250	512-592	Basic input buffer
0251-025A	593-602	Logical file number table
025B-0264	603-612	Device number table
0265-026E	613-622	Secondary address, or R/W cmd, table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape#1 buffer
033A-03F9	826-1017	Tape#2 buffer
03FA-03FB	1018-1019	Vector for Machine Language Monitor
0400-7FFF	1024-32767	Available RAM including expansion
8000-8FFF	32768-36863	Video RAM
9000-BFFF	36864-49151	Available ROM expansion area
C000-E0F8	49152-57592	Microsoft Basic interpreter
E0F9-E7FF	57593-59391	Keyboard, Screen, Interrupt programs
E810-E813	59408-59411	PIA1 - Keyboard I/O
E820-E823	59424-59427	PIA2 - IEEE488 I/O
E840-E84F	59456-59471	VIA - I/O and Timers
F000-FFFF	61440-65535	Reset, tape, diagnostic monitor

PET MEMORY LOCATIONS

September 1978

0000-0002	0-2	USR Jump instruction
0003	3	Current I/O Device for prompt-suppress
0005	5	Cursor position for Input / Print
0008-0009	8-9	Integer address from Basic (for SYS, GOTO, etc.)
000A-0059	10-89	Basic input buffer; # of array subscripts
005A	90	Search character (usually ':' or end-of-line)
005B	91	Scan-between-quotes flag
005C	92	Basic input buffer pointer; number of subscripts
005D	93	First-character of array-name; default DIM flag
005E	94	Type: FF=string; 00=numeric
005F	95	Type: 80=integer; 00=floating point
0060	96	'DATA' scan flag; LIST quote flag; memory flag
0061	97	Subscript flag; FNx flag
0062	98	0=input, 64=get, 152=read (flag)
0063	99	flag for trigonometric signs/comparison evaluation flag
0064	100	input flag (suppress output if negative)
0065	101	variable pseudo-stack pointer
0066	102	fixed-point pseudo-stack pointer
0067	103	dummy value (0)
0068-0070	104-112	variable # pseudo-stack
0071-0072	113-114	pointer for number transfer
0073-0074	115-116	number pointer
0075-0078	117-120	product staging area for multiplication
007A-007B	122-123	start of basic pointer
007C-007D	124-125	end of basic/start of varibles pointer
007E-007F	126-127	end of variables/startof arrays
0080-0081	128-129	start of available space pointer
0082-0083	130-131	bottom of strings (moving down) pointer
0084-0085	132-133	top of strings (moving down) pointer
0086-0087	134-135	limit of Basic memory pointer
0088-0089	136-137	current program line number
008A-008B	138-139	previous line number
008C-008D	140-141	previous line address (for CONT)
008E-008F	142-143	line number of DATA line
0090-0091	144-145	memory address of DATA line
0092-0093	146-147	input vector (DATA etc.)
0094-0095	148-149	current variable name
0096-0097	150-151	current variable address
0098-0099	152-153	variable pointer for current FOR/NEXT
009A	154	Y save register; new operator save
009C	156	comparison symbol accumulator: < 1 =2 > 4
009D-00A1	157-161	number work area for SQR, etc.
00A2	162	pseudo-stack yardstick (3 or 7)
00A3-00A5	163-165	jump vector for functions
00A6-00AA	166-170	numeric store area
00AB-00AF	171-175	numeric store area
00B0-00B5	176-181	primary accumulator E,M,M,M,M,S
00B6	182	Taylor series constant counter
00B7	183	accumulator high-order propogation word
00B8-00BD	184-189	secondary accumulator
00BE	190	sign comparison, primary/secondary
00BF	191	low-order rounding byte for primary acc

00C0-00C1	192-193	Cassette buffer length/Taylor constant pointer
00C2-00D9	194-217	Subrtn: Get Basic Char; C9,CA=pointer
00DA-00DE	218-222	RND storage and work area
00E0-00E1	224-225	Pointer to screen cursor line
00E2	226	Porsition of cursor on line
00E3-00E4	227-228	Utility pointer; tape buffer,scrolling
00E5-00E6	229-230	End of current program/tape end address
00E7-00E8	231-232	Tape timing constants
00E9	233	Tape buffer character
00EA	234	Direct/programmed cursor; 00=direct
00EB	235	Tape read/verify flag
00EC	236	Tape write flag
00ED	237	
00EE	238	Number of # characters in file name
00EF	239	Logical file number
00F0	240	File command (from OPEN)
00F1	241	Device number
00F2	242	Maximum line length (40 or 80)
00F3-00F4	243-244	Tape buffr address (start of buffer)
00F5	245	Line where cursor lives
00F6	246	Last key pushed (ASCII); buffer checksum
00F7-00F8	247-248	Tape start address/tape pointer
00F9-00FA	249-250	File name pointer
00FB	251	Number of "insert" keys pushed
00FC	252	Serial bit shift word
00FD	253	# blocks remaining to write
00FE	254	Serial word buffer
0100-010A	256-266	Binary to ASCII conversion area
0105-01FF	267-511	Stack area
0200-0202	512-514	TI and TI$ clock - jiffies
0203	515	Which key depressed: 255 = no key
0204	516	Shift key: 1 if depressed
0205-0206	517-518	Clock (unused?)
0207	519	Cassette 1 status switch
0208	520	Cassette 2 status switch
0209	521	Keyswitch PIA: STOP & RVS flags, etc.
020A	522	
020B	523	Load=0, Verify=1
020C	524	Status
020D	525	# characters in keyboard buffer
020E	526	Reverse flag
020F-0218	527-536	Keyboard buffer
0219-021A	537-538	Hardware interrupt vector
021B-021C	539-540	Break interrupt vector
021D	541	
021E	542	End-of-line-for -input pointer
0220-0221	544-545	Cursor log (row, column)
0222	546	PBD image for tape I/O
0223	547	Key image
0224	548	0=flashing cursor; else no cursor shows
0225	549	Cursor timing countdown
0226	550	Character under cursor
0227	551	Cursor blink flag
0228	552	Tape write
[illegible]	553-[illegible]	[illegible]e addr[illegible] screen[illegible]n table

0242-024B	578-587	Logical numbers of open files
024C-0255	588-597	Device numbers of open files
0256-025F	598-607	Command/Secondary address of open files
0260	608	Input from screen/input from keyboard
0261	609	X-save flag
0262	610	How many open files
0263	611	Input device, normally 0
0264	612	Output CMD device, normally 3
0265	613	Tape parity
0266	614	
0268	616	Pointer in filename transfer
026A	618	
026C	620	Serial bit count
026F	623	
0270	624	Tape write countdown
0271	625	Tape buffer #1 count
0272	626	Tape buffer #2 count
0273	627	Leader counter
0274	628	Flag for tape error
0275	629	0 if 1st ½-byte cntr not written
0276	630	2nd ½-byte cntr/tape error count
0277	631	
0278	632	Cassette read flag
0279	633	Checksum working word
027A-0339	634-825	Tape #1 buffer
033A-03F9	826-1017	Tape #2 buffer
0400-7FFF	1024-32767	Available RAM including expansion
8000-8FFF	32768-36863	Video RAM
9000-BFFF	36864-49151	Available ROM expansion area
C000-E077	49152-57463	Microsoft Basic
E078-E7F8	57464-59384	Keyboard/Screen/Interrupt monitor
E810	59408	PIA1 - Keyboard A register; (Direction with CRA2=1)
E811	59409	PIA1 - Keyboard A control
E812	59410	PIA1 - Keyboard B register; (Direction with CRB2=1)
E813	59411	PIA1 - Keyboard B control
E820	59424	PIA2 - IEEE A register; (Direction with CRA2=1)
E821	59425	PIA2 - IEEE A control
E822	59426	PIA2 - IEEE B register; (Direction with CRB2=1)
E823	59427	PIA2 - IEEE B control
E840	59456	VIA I/O register B
E841	59457	VIA I/O register A with handshake
E842-E843	59458-59459	VIA Data Direction regs, A and B
E844-E845	59460-59461	VIA Timer 1
E846-E847	59462-59463	VIA Timer 1 latch
E848-E849	59464-59465	VIA Timer 2
E84A	59466	VIA shift register
E84B	59467	ACS: T1.T1.T2.SR.SR.SR.PB.PA
E84C	59468	PCR: B2.B2.B2.B1.A2.A2.A2.A1
E84D-E84E	59469-59470	IFR, IER: T1.T2.CB1.BC2.SR.CA1.CA2
E84F	59471	I/O Register A without handshake
F000-FFFF	61440-65535	Reset/tape/diagnostic monitor

BASIC 4.0 MEMORY MAP

Compiled by Jim Butterfield

There are some differences between usage between the 40- and 80-column machines.

Hex	Decimal	Description
0000-0002	0-2	USR jump
0003	3	Search character
0004	4	Scan-between-quotes flag
0005	5	Input buffer pointer; # of subscripts
0006	6	Default DIM flag
0007	7	Type: FF=string, 00=numeric
0008	8	Type: 80=integer, 00=floating point
0009	9	Flag: DATA scan; LIST quote; memory
000A	10	Subscript flag; FNX flag
000B	11	0=INPUT; $40=GET; $98=READ
000C	12	ATN sign/Comparison Evaluation flag
000D-000F	13-15	Disk status DS$ descriptor
0010	16	Current I/O device for prompt-suppress
0011-0012	17-18	Integer value (for SYS, GOTO etc)
0013-0015	19-21	Pointers for descriptor stack
0016-001E	22-30	Descriptor stack(temp strings)
001F-0022	31-34	Utility pointer area
0023-0027	35-39	Product area for multiplication
0028-0029	40-41	Pointer: Start-of-Basic
002A-002B	42-43	Pointer: Start-of-Variables
002C-002D	44-45	Pointer: Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: String-storage(moving down)
0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit-of-memory
0036-0037	54-55	Current Basic line number
0038-0039	56-57	Previous Basic line number
003A-003B	58-59	Pointer: Basic statement for CONT
003C-003D	60-61	Current DATA line number
003E-003F	62-63	Current DATA address
0040-0041	64-65	Input vector
0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/NEXT
0048-0049	72-73	Y-save; op-save; Basic pointer save
004A	74	Comparison symbol accumulator
004B-0050	75-80	Misc work area, pointers, etc
0051-0053	81-83	Jump vector for functions
0054-005D	84-93	Misc numeric work area
005E	94	Accum#1: Exponent
005F-0062	95-98	Accum#1: Mantissa
0063	99	Accum#1: Sign
0064	100	Series evaluation constant pointer
0065	101	Accum#1 hi-order (overflow)
0066-006B	102-107	Accum#2: Exponent, etc.
006C	108	Sign comparison, Acc#1 vs #2
006D	106	Accum#1 lo-order (rounding)
006E-006F	110-111	Cassette buff len/Series pointer
0070-0087	112-135	CHRGET subroutine; get Basic char
0077-0078	119-120	Basic pointer (within subrtn)
0088-008C	136-140	Random number seed.
008D-008F	141-143	Jiffy clock for TI and TI$
0090-0091	144-145	Hardware interrupt vector
0092-0093	146-147	BRK interrupt vector
0094-0095	148-149	NMI interrupt vector
0096	150	Status word ST
0097	151	Which key down; 255=no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Correction clock
009B	155	Keyswitch PIA: STOP and RVS flags
009C	156	Timing constant for tape
009D	157	Load=0, Verify=1
009E	158	Number of characters in keybd buffer
009F	159	Screen reverse flag
00A0	160	IEEE output; 255=character pending
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row, column)
00A5	165	IEEE output buffer
00A6	166	Key image
00A7	167	0=flash cursor
00A8	168	Cursor timing countdown
00A9	169	Character under cursor
00AA	170	Cursor in blink phase
00AB	171	EOT received from tape
00AC	172	Input from screen/from keyboard
00AD	173	X save
00AE	174	How many open files
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally 3
00B1	177	Tape character parity
00B2	178	Byte received flag
00B3	179	Logical Address temporary save
00B4	180	Tape buffer character; MLM command
00B5	181	File name pointer; MLM flag, counter
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Tape writer countdown
00BB-00BC	187-188	Tape buffer pointers, #1 and #2
00BD	189	Write leader count; read pass1/2
00BE	190	Write new byte; read error flag
00BF	191	Write start bit; read bit seq error
00C0-00C1	192-193	Error log pointers, pass1/2
00C2	194	0=Scan/1-15=Count/$40=Load/$80=End
00C3	195	Write leader length; read checksum
00C4-00C5	196-197	Pointer to screen line
00C6	198	Position of cursor on above line
00C7-00C8	199-200	Utility pointer: tape, scroll
00C9-00CA	201-202	Tape end addrs/End of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	0=direct cursor, else programmed
00CE	206	Tape read timer 1 enabled
00CF	207	EOT received from tape
00D0	208	Read character error
00D1	209	# characters in file name
00D2	210	Current file logical address
00D3	211	Current file secondary addrs
00D4	212	Current file device number
00D5	213	Right-hand window or line margin
00D6-00D7	214-215	Pointer: Start of tape buffer
00D8	216	Line where cursor lives
00D9	217	Last key/checksum/misc.
00DA-00DB	218-219	File name pointer
00DC	220	Number of INSERTs outstanding
00DD	221	Write shift word/read character in
00DE	222	Tape blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	(40-column) Screen line wrap table
00E0-00E1	224-225	(80-column) Top, bottom of window
00E2	226	(80-column) Left window margin
00E3	227	(80-column) Limit of keybd buffer
00E4	228	(80-column) Key repeat flag
00E5	229	(80-column) Repeat countdown
00E6	230	(80-column) New key marker
00E7	231	(80-column) Chime time
00E8	232	(80-column) HOME count
00E9-00EA	233-234	(80-column) Input vector
00EB-00EC	235-236	(80-column) Output vector
00F9-00FA	249-250	Cassette status, #1 and #2
00FB-00FC	251-252	MLM pointer/Tape start address
00FD-00FE	253-254	MLM, DOS pointer, misc.
0100-010A	256-266	STR$ work area, MLM work
0100-013E	256-318	Tape read error log
0100-01FF	256-511	Processor stack
0200-0250	512-592	MLM work area; Input buffer
0251-025A	593-602	File logical address table
025B-0264	603-612	File device number table
0265-026E	613-622	File secondary adds table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape#1 input buffer
033A-03F9	826-1017	Tape#2 input buffer
033A	826	DOS character pointer
033B	827	DOS drive 1 flag
033C	828	DOS drive 2 flag
033D	829	DOS length/write flag
033E	830	DOS syntax flags
033F-0340	831-832	DOS disk ID
0341	833	DOS command string count
0342-0352	834-850	DOS file name buffer
0353-0380	851-896	DOS command string buffer
03EE-03F7	1006-1015	(80-column) Tab stop table
03FA-03FB	1018-1019	Monitor extension vector
03FC	1020	IEEE timeout defeat
0400-7FFF	1024-32767	Available RAM including expansion
8000-83FF	32768-33791	(40-column) Video RAM
8000-87FF	32768-34815	(80-column) Video RAM
9000-AFFF	36864-45055	Available ROM expansion area
B000-DFFF	45056-57343	Basic, DOS, Machine Lang Monitor
E000-E7FF	57344-59391	Screen, Keyboard, Interrupt programs
E810-E813	59408-59411	PIA 1 - Keyboard I/O
E820-E823	59424-59427	PIA 2 - IEEE-488 I/O
E840-E84F	59456-59471	VIA - I/O and timers
E880-E881	59520-59521	(80-column) CRT Controller
F000-FFFF	61440-65535	Reset, I/O handlers, Tape routines

PET 4.0 ROM Routines

Jim Butterfield, Toronto

The 40-character and 80-character machines are the same except for addresses $E000-$E7FF.

This map shows where various routines lie. The first address is not necessarily the proper entry point for the routine. Similarly, many routines require register setup or data preparation before calling.

	Description
B000-B065	Action addresses for primary keywords
B066-B093	Action addresses for functions
B094-B0B1	Hierarchy and action addresses for operators
B0B2-B20C	Table of Basic keywords
B20D-B321	Basic messages, mostly error messages
B322-B34F	Search the stack for FOR or GOSUB activity
B350-B392	Open up space in memory
B393-B39F	Test: stack too deep?
B3A0-B3CC	Check available memory
B3CD	Send canned error message, then:
B3FF-B41E	Warm start; wait for Basic command
B41F-B4B5	Handle new Basic line input
B4B6-B4E1	Rebuild chaining of Basic lines
B4E2-B4FA	Receive line from keyboard
B4FB-B5A2	Crunch keywords into Basic tokens
B5A3-B5D1	Search Basic for given line number
B5D2	Perform NEW, and;
B5EC-B621	Perform CLR
B622-B62F	Reset Basic execution to start
B630-B6DD	Perform LIST
B6DE-B784	Perform FOR
B785-B7B6	Execute Basic statement
B7B7-B7C5	Perform RESTORE
B7C6-B7ED	Perform STOP or END
B7EE-B807	Perform CONT
B808-B812	Perform RUN
B813-B82F	Perform GOSUB
B830-B85C	Perform GOTO
B85D	Perform RETURN, then:
B883-B890	Perform DATA: skip statement
B891	Scan for next Basic statement
B894-B8B2	Scan for next Basic line
B8B3	Perform IF, and perhaps:
B8C6-B8D5	Perform REM: skip line
B8D6-B8F5	Perform ON
B8F6-B92F	Accept fixed-point number
B930-BA87	Perform LET
BA88-BA8D	Perform PRINT#
BA8E-BAA1	Perform CMD
BAA2-BB1C	Perform PRINT
BB1D-BB39	Print string from memory
BB3A-BB4B	Print single format character
BB4C-BB79	Handle bad input data
BB7A-BBA3	Perform GET
BBA4-BBBD	Perform INPUT#
BBBE-BBF4	Perform INPUT
BBF5-BC01	Prompt and receive input
BC02-BCF6	Perform READ
BCF7-BD18	Canned Input error messages
BD19-BD71	Perform NEXT
BD72-BD97	Check type mismatch
BD98	Evaluate expression
BEE9	Evaluate expression within parentheses
BEEF	Check parenthesis, comma
BF00-BF0B	Syntax error exit
BF8C-C046	Variable name setup
C047-C085	Set up function references
C086-C0B5	Perform OR, AND
C0B6-C11D	Perform comparisons
C11E-C12A	Perform DIM
C12B-C1BF	Search for variable
C1C0-C2C7	Create new variable
C2C8-C2D8	Setup array pointer
C2D9-C2DC	32768 in floating binary
C2DD-C2FB	Evaluate integer expression
C2FC-C4A7	Find or make array
C4A8	Perform FRE, and:
C4BC-C4C8	Convert fixed-to-floating
C4C9-C4CE	Perform POS
C4CF-C4DB	Check not Direct
C4DC-C509	Perform DEF
C50A-C51C	Check FNx syntax
C51D-C58D	Evaluate FNx
C58E-C59D	Perform STR$
C59E-C5AF	Do string vector
C5B0-C61C	Scan, set up string
C61D-C669	Allocate space for string
C66A-C74E	Garbage collection
C74F-C78B	Concatenate
C78C-C7B4	Store string
C7B5-C810	Discard unwanted string
C811-C821	Clean descriptor stack
C822-C835	Perform CHR$
C836-C861	Perform LEFT$
C862-C86C	Perform RIGHT$
C86D-C896	Perform MID$
C897-C8B1	Pull string data
C8B2-C8B7	Perform LEN
C8B8-C8C0	Switch string to numeric
C8C1-C8D0	Perform ASC
C8D1-C8E2	Get byte parameter
C8E3-C920	Perform VAL
C921-C92C	Get two parameters for POKE or WAIT
C92D-C942	Convert floating-to-fixed
C943-C959	Perform PEEK
C95A-C962	Perform POKE
C963-C97E	Perform WAIT
C97F-C985	Add 0.5
C986	Perform subtraction
C998-CA7C	Perform addition
CA7D-CAB3	Complement accum#1
CAB4-CAB8	Overflow exit
CAB9-CAF1	Multiply-a-byte
CAF2-CB1F	Constants
CB20	Perform LOG
CB5E-CBC1	Perform multiplication
CBC2-CBEC	Unpack memory into accum#2
CBED-CC09	Test & adjust accumulators
CC0A-CC17	Handle overflow and underflow
CC18-CC2E	Multiply by 10
CC2F-CC33	10 in floating binary
CC34	Divide by 10
CC3D	Perform divide-by
CC45-CCD7	Perform divide-into
CCD8-CCFC	Unpack memory into accum#1
CCFD-CD31	Pack accum#1 into memory
CD32-CD41	Move accum#2 to #1
CD42-CD50	Move accum#1 to #2
CD51-CD60	Round accum#1
CD61-CD6E	Get accum#1 sign
CD6F-CD8D	Perform SGN
CD8E-CD90	Perform ABS
CD91-CDD0	Compare accum#1 to memory
CDD1-CE01	Floating-to-fixed
CE02-CE28	Perform INT
CE29-CEB3	Convert string to floating-point
CEB4-CEE8	Get new ASCII digit
CEE9-CEF8	Constants
CF78	Print IN, then:
CF7F-CF92	Print Basic line #
CF93-D0C6	Convert floating-point to ASCII
D0C7-D107	Constants
D108	Perform SQR
D112	Perform power function
D14B-D155	Perform negation
D156-D183	Constants
D184-D1D6	Perform EXP
D1D7-D220	Series evaluation
D221-D228	RND constants
D229-D281	Perform RND
D282	Perform COS
D289-D2D1	Perform SIN
D2D2-D2FD	Perform TAN
D2FE-D32B	Constants
D32C-D35B	Perform ATN
D35C-D398	Constants
D399-D3B5	CHRGET sub for zero page
D3B6-D471	Basic cold start
D472-D716	Machine Language Monitor
D717-D7AB	MLM subroutines
D7AC-D802	Perform RECORD
D803-D837	Disk parameter checks
D838-D872	Dump disk control messages
D873-D919	Perform CATALOG or DIRECTORY
D91A-D92E	Output
D92F-D941	Find spare secondary address
D942-D976	Perform DOPEN
D977-D990	Perform APPEND
D991-D9D1	Get disk status
D9D2-DA06	Perform HEADER
DA07-DA30	Perform DCLOSE
DA31-DA64	Set up disk record
DA65-DA7D	Perform COLLECT
DA7E-DAA6	Perform BACKUP
DAA7-DAC6	Perform COPY
DAC7-DAD3	Perform CONCAT
DAD4-DB0C	Insert command string values
DB0D-DB39	Perform DSAVE
DB3A-DB65	Perform DLOAD
DB66-DB98	Perform SCRATCH
DB99-DB9D	Check Direct command
DB9E-DBD6	Query ARE YOU SURE?
DBD7-DBE0	Print BAD DISK
DBE1-DBF9	Clear DS$ and ST
DBFA-DC67	Assemble disk command string
DC68-DE29	Parse Basic DOS command
DE2C-DE48	Get Device number
DE49-DE86	Get file name
DE87-DE9C	Get small variable parameter

** Entry points only for E000-E7FF **

E00C	Register/screen initialization
E0A7	Input from keyboard
E116	Input from screen
E202	Output character
E442	Main Interrupt entry
E455	Interrupt: clock, cursor, keyboard
E600	Exit from Interrupt

**   **

F000-F0D1	File messages
F0D2	Send 'Talk'
F0D5	Send 'Listen'
F0D7	Send IEEE command character
F109-F142	Send byte to IEEE
F143-F150	Send byte and clear ATN
F151-F16B	Option: timeout or wait
F16C-F16F	DEVICE NOT PRESENT
F170-F184	Timeout on read, clear control lines
F185-F192	Send canned file message
F193-F19D	Send byte, clear control lines
F19E-F1AD	Send normal (deferred) IEEE char
F1AE-F1BF	Drop IEEE device
F1C0-F204	Input byte from IEEE
F205-F214	GET a byte
F215-F265	INPUT a byte
F266-F2A1	Output a byte
F2A2	Abort files
F2A6-F2C0	Restore default I/O devices
F2C1-F2DC	Find/setup file data
F2DD-F334	Perform CLOSE
F335-F342	Test STOP key
F343-F348	Action STOP key
F349-F350	Send message if Direct mode
F351-F355	Test if Direct mode
F356-F400	Program load subroutine
F401-F448	Perform LOAD
F449-F46C	Print SEARCHING
F46D-F47C	Print LOADING or VERIFYING
F47D-F4A4	Get Load/Save parameters
F4A5-F4D2	Send name to IEEE
F4D3-F4F5	Find specific tape header
F4F6-F50C	Perform VERIFY
F50D-F55F	Get Open/Close parameters
F560-F5E4	Perform OPEN
F5E5-F618	Find any tape header
F619-F67A	Write tape header
F67B-F694	Get start/end addrs from header
F695-F6AA	Set buffer address
F6AB-F6C2	Set buffer start & end addrs
F6C3-F6CB	Perform SYS
F6CC-F6DC	Set tape write start & end
F6DD-F767	Perform SAVE
F768-F7AE	Update clock
F7AF-F7FD	Connect input device
F7FE-F84A	Connect output device
F84B-F856	Bump tape buffer pointer
F857-F879	Wait for PLAY
F87A-F88B	Test cassette switch
F88C-F899	Wait for RECORD
F89A	Initiate tape read
F8CB	Initiate tape write
F8E0-F92A	Common tape I/O
F92B-F934	Test I/O complete
F935-F944	Test STOP key
F945-F975	Tape bit timing adjust
F976-FA9B	Read tape bits
FA9C-FBBA	Read tape characters
FBBB-FBC3	Reset tape read address
FBC4-FBC8	Flag error into ST
FBC9-FBD7	Reset counters for new byte
FBD8-FBF3	Write a bit to tape
FBF4-FC85	Tape write
FC86-FCBF	Write tape leader
FCC0-FCDA	Terminate tape; restore interrupt
FCDB-FCEA	Set interrupt vector
FCEB-FCF8	Turn off tape motor
FCF9-FD0A	Checksum calculation
FD0B-FD15	Advance load/save pointer
FD16-FD4B	Power-on Reset
FD4C-FD5C	Table of interrupt vectors

** Jump table: **

FF93-FF9E	CONCAT,DOPEN,DCLOSE,RECORD
FF9F-FFAA	HEADER,COLLECT,BACKUP,COPY
FFAB-FFB6	APPEND,DSAVE,DLOAD,CATALOG
FFB7-FFBC	RENAME,SCRATCH
FFBD	Get disk status
FFC0	OPEN
FFC3	CLOSE
FFC6	Set input device
FFC9	Set output device
FFCC	Restore default I/O devices
FFCF	INPUT a byte
FFD2	Output a byte
FFD5	LOAD
FFD8	SAVE
FFDB	VERIFY
FFDE	SYS
FFE1	Test stop key
FFE4	GET byte
FFE7	Abort all files
FFEA	Update clock
FFFA-FFFF	Hard vectors: NMI, Reset, INT

A Few Entry Points, 1.0 / 2.0 / 4.0 ROM: Jim Butterfield

Entry points seen in various programmer's machine language programs. The user is cautioned to check out the various routines carefully for proper setup before calling, registers used, etc.

ORIG	UPGR	4.0	DESCRIPTION
	C2D8	B350	Open space in BASIC text
	C328	B3A0	Check available memory
C357	C355	B3CD	?OUT OF MEMORY
C359	C357	B3CF	Send Basic error message
C38B	C389	B3FF	Warm start, Basic
	C39B	B40F	Main CHRGET entry
C3AC	C3AB	B41F	Crunch & insert line
C430	C439	B4AD	Fix chaining & READY.
C433	C442	B4B6	Fix chaining
	C46F	B4E2	Receive line from keyboard
C48D	C495	B4FB	Crunch tokens
C522	C52C	B5A3	Find line in Basic
C553	C55D	B5D4	Do NEW
C567	C572	B5E9	Reset Basic and do CLR
C56A	C575	B5EC	Do CLR
C59A	C5A7	B622	Reset Basic to start
C6B5	C6C4	B74A	Continue Basic execution
C863	C873	B8F6	Get fixed-point number from Basic.
C9CE	C9DE	BADB	Send Return,LF if in screen mode
C9D2	C9E2	BADF	Send Return, Linefeed
CA27	CA1C	BB1D	Print string
CA2D	CA22	BB23	Print precomputed string
CA47	CA43	BB44	Print "?"
CA49	CA45	BB46	Print character (output .A to device)
CE11	CDF8	BEF5	Check for comma
CE13	CDFA	BEF7	Check for specific character
CE1C	CE03	BF00	'SYNTAX ERROR'
CFD7	CFC9	C187	Find fl-pt variable, given name
D079	D069	C2B9	Bump Variable Address by 2
D0A7	D09A	C2EA	Float to Fixed conversion
D278	D26D	C4BC	Fixed to Float conversion
na	D472	FD11	Entry to m.l.m. (dec. 54386 & 64785 resp.)
D679	D67B	C8D7	Get byte to X reg
D68D	D68F	C8EB	Evaluate String
D6C4	D6C6	C921	Get two parameters
D73C	D773	C99D	Add (from memory)
D8FD	D934	CB5E	Multiply by memory location
D9B4	D9EE	CC18	Multiply by ten
DA74	DAAE	CCD8	Unpack memory variable to Accum #1
DAA9	DAE3	CD0D	Copy Acc #1 to (X,Y) location
DB1B	DB55	CD7F	Completion of Fixed to Float conversion
DC9F	DCD9	CF83	Print fixed-point value
DCA9	DCE3	CF8D	Print floating-point value
DCAF	DCE9	CF93	Convert number to ASCII string
E3EA	E3D8	E202	Print a character
na	E76A	D717	Output 4 ASCII hex chars from $FB,FC
na	E775	D722	Output .A as 2 hex digits
na	E7A7	D754	Input 2 hex digits to A
na	E7E0	D78D	Transfer 1 ASCII hex digit to A in binary
na	E7B6	D763	Input 1 hex digit to A
E7DE	F156	F185	Print system message
F0B6	F0B6	F0D2	Send 'talk' to IEEE
F0BA	F0BA	F0D5	Send 'listen' to IEEE
F12C	F128	F143	Send Secondary Address
E7DE	F156	F185	Send canned message
F167	F16F	F19E	Send character to IEEE
F17A	F17F	F1B6	Send 'untalk'
F17E	F183	F1B9	Send 'unlisten'
F187	F18C	F1C0	Input from IEEE
F2C8	F2A9	F2DD	Close logical file
F2CD	F2AE	F2E2	Close logical file in A
F32A	F301	F335	Check for Stop key
F33F	F315	F349	Send message if Direct mode
na	F322	F356	LOAD subroutine
F3DB	F3E6	F425	?LOAD ERROR
F3E5	F3EF	F42E	Print READY & reset Basic to start
F3FF	F40A	F449	Print SEARCHING...
F411	F41D	F45C	Print file name
F43F	F447	F486	Get LOAD/SAVE type parameters
F462	F466	F4A5	Open IEEE channel for output.
F495	F494	F4D3	Find specific tape header block
F504	F4FD	F53C	Get string
F52A	F521	F560	Open logical file from input parameters
F52D	F524	F563	Open logical file
F579	F56E	F5AD	?FILE NOT FOUND, clear I/O
F57B	F570	F5AF	Send error message
F5AE	F5A6	F5E5	Find any tape header block
F64D	F63C	F67B	Get pointers for tape LOAD
F667	F656	F695	Set tape buffer start address
F67D	F66C	F6AB	Set cassette buffer pointers
F6E6	F6F0	F72F	Close IEEE channel
F78B	F770	F7AF	Set input device from logical file number
F7DC	F7BC	F7DF	Set output device from LFN.
F83B	F812	F857	PRESS PLAY..; wait
F85E	F835	F87A	Sense tape switch
F87F	F855	F89A	Read tape to buffer
F88A	F85E	F8A3	Read tape
F8B9	F886	F8CB	Write tape from buffer
F8C1	F88E	F8D3	Write tape, leader length in A
F913	F8E6	F92B	Wait for I/O complete or Stop key
FBDC	FB76	FBBB	Reset tape I/O pointer
FD1B	FC9B	FCE0	Set interrupt vector
FFC6	FFC6	FFC6	Set input device
FFC9	FFC9	FFC9	Set output device
FFCC	FFCC	FFCC	Restore default I/O devices
FFCF	FFCF	FFCF	Input character
FFD2	FFD2	FFD2	Output character
FFE4	FFE4	FFE4	Get character

MNE.	IMMED	ZPAGE	ZPG.X	ZPG.Y	(I,X)	(I),Y	ABSOL	ABS.X	ABS.Y	INDAB	RELAT	IMPL.
ORA	09	05	15		01	11	0D	1D	19			
AND	29	25	35		21	31	2D	3D	39			
EOR	49	45	55		41	51	4D	5D	59			
ADC	69	65	75		61	71	6D	7D	79			
STA		85	95		81	91	8D	9D	99			
LDA	A9	A5	B5		A1	B1	AD	BD	B9			
CMP	C9	C5	D5		C1	D1	CD	DD	D9			
SBC	E9	E5	F5		E1	F1	ED	FD	F9			
ASL		06	16				0E	1E				
ROL		26	36				2E	3E				
LSR		46	56				4E	5E				
ROR		66	76				6E	7E				
STX		86		96			8E					
LDX	A2	A6		B6			AE		BE			
DEC		C6	D6				CE	DE				
INC		E6	F6				EE	FE				
BIT		24					2C					
STY		84					8C					
LDY	A0	A4	B4				AC					
CPY	C0	C4					CC					
CPX	E0	E4					EC					
BPL											10	
BVC											50	
BCC											90	
BNE											D0	
BMI											30	
BVS											70	
BCS											B0	
BEQ											F0	
JSR							20					
JMP							4C			6C		
BRK												00
RTI												40
RTS												60
PHP												08
CLC												18
PLP												28
SEC												38
PHA												48
CLI												58
PLA												68
SEI												78
DEY												88
TYA												98
TAY												A8
CLV												B8
INY												C8
CLD												D8
INX												E8
SED												F8
ASL												0A
ROL												2A
LSR												4A
ROR												6A
TXA												8A
TXS												9A
TAX												AA
TSX												BA
DEX												CA
NOP												EA

DS & DS$: Disk Status Variables

DS returns the CBM disk error number & DS$ returns a string consisting of the error number, error description and track & sector, if applicable.

DS	Error Description
0	OK, no error exists
2-19	Unused error messages: can occur, should be ignored
20	read error; block header not found
21	read error; sync character not found
22	read error; data block not present
23	read error; checksum error in data
24	read error; byte decoding error
25	write error; write verify error
26	write protect on
27	read error; checksum error in header
28	write error; data extends into next block
29	disk id mismatch
30	syntax error; general syntax
31	syntax error; invalid command
32	syntax error; command line greater than 58 chars
33	syntax error; invalid filename
34	syntax error; no filename given
39	syntax error; command file not given
50	record not present
51	overflow in record
52	file too large
60	file open for write
61	file not open
62	file not found
63	file exists
64	file type mismatch
65	no block; t,s is next available block
66	illegal track or sector
67	illegal system track or sector
70	no channels (available)
71	dir error (directory error)
72	disk full (could indicate directory full)
73	cdm dos v2 (or v2.5 for 8050); power up message, also indicates write attempt with DOS mismatch, DOS 2.0 & 2.5 only
74	drive not ready (8050 only)

Note: After files are SCRATCHed, the number of files scratched will be returned with a "files scratched" error message. This is not an error condition.

ST: The Status Word

ST returns the CBM status corresponding to the last I/O operation, whether over cassette, screen, keyboard or IEEE.

ST bit	ST value	Cassette Read	IEEE	Tape Verify and Load
na	0	OK	OK	OK
0	1		time out on write	
1	2		time out on read	
2	4	Short block		Short block
3	8	Long block		Long block
4	16	Unrecoverable read error		Any mismatch
5	32	Checksum error		Checksum error
6	64	End of file	EOI	
7	-128	End of tape	Device not present	End of tape

80 Column Screen, Line Start Addresses

Ln#	Dec	Hex	Notes
0	32768	$8000	
1	32848	8050	
2	32928	80A0	
3	33008	80F0	
4	33088	8140	
5	33168	8190	
6	33248	81E0	
7	33328	8230	
8	33408	8280	
9	33488	82D0	
10	33568	8320	
11	33648	8370	
12	33728	83C0	
13	33808	8410	
14	33888	8460	
15	33968	84B0	
16	34048	8500	
17	34128	8550	
18	34208	85A0	
19	34288	85F0	
20	34368	8640	
21	34448	8690	
22	34528	86E0	
23	34608	8730	
24	34688	8780	

8032 Control Characters

This table is a summary of the 8032 screen control functions. The ESC/RVS characters will display as lower/upper case or upper case/graphics, depending on which mode you're in. POKE59468,X (where X=12 for graphics, 14 for lower case) still changes modes without changing the gap between the lines. Notice that complimentary functions differ by 128 using CHR$. See the Commodore BASIC 4.0 manual for details on functions.

Function	CHR$(value)	ESC/RVS char.	Keyboard Combination
BELL	7	g	
GRAPHICS	142	shift n	
TEXT	14	n	BOTHShifts / "
SCROLL DOWN	153	shift y	LeftShift / TAB / I
SCROLL UP	25	y	
SET BOTTOM	143	shift o	Shift / Z / A / L
SET TOP	15	o	Z / A / L
INSERT LINE	149	shift u	Shift / RVS / A / L
DELETE LINE.	21	u	RVS / A / L
ERASE BEGIN	150	shift v	Shift / TAB / ← / DEL
ERASE END	22	v	/ TAB / ← / DEL
SET/CLR TAB	137	shift i	Shift / TAB
TAB	9	i	TAB

← is the leftarrow key, not cursor right.

Window POKEs

Screen TOP:	224,T	where T=0 to 24
BOTTOM:	225,B	where B=T to 24
LEFT:	226,L	where L=0 to 79
RIGHT:	213,R	where R=L to 79

32768

2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799	2800	2801	2802	2803	2804	2805	2806	2807
2808	2809	2810	2811	2812	2813	2814	2815	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879	2880	2881	2882	2883	2884	2885	2886	2887
2888	2889	2890	2891	2892	2893	2894	2895	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959	2960	2961	2962	2963	2964	2965	2966	2967
2968	2969	2970	2971	2972	2973	2974	2975	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039	3040	3041	3042	3043	3044	3045	3046	3047
3048	3049	3050	3051	3052	3053	3054	3055	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119	3120	3121	3122	3123	3124	3125	3126	3127
3128	3129	3130	3131	3132	3133	3134	3135	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199	3200	3201	3202	3203	3204	3205	3206	3207
3208	3209	3210	3211	3212	3213	3214	3215	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279	3280	3281	3282	3283	3284	3285	3286	3287
3288	3289	3290	3291	3292	3293	3294	3295	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359	3360	3361	3362	3363	3364	3365	3366	3367
3368	3369	3370	3371	3372	3373	3374	3375	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439	3440	3441	3442	3443	3444	3445	3446	3447
3448	3449	3450	3451	3452	3453	3454	3455	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519	3520	3521	3522	3523	3524	3525	3526	3527
3528	3529	3530	3531	3532	3533	3534	3535	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599	3600	3601	3602	3603	3604	3605	3606	3607
3608	3609	3610	3611	3612	3613	3614	3615	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679	3680	3681	3682	3683	3684	3685	3686	3687
3688	3689	3690	3691	3692	3693	3694	3695	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759	3760	3761	3762	3763	3764	3765	3766	3767

33767

The printing mode (standard or lower case) is set by POKEing an address. So as not to disturb any of the other bits in the peripheral control register a safe way to set the lower case mode would be: POKE 59468,PEEK(59468) OR 14 and reset it to standard mode with POKE 59468, PEEK(59468) AND 253 OR 12.

Standard Mode: Location 59468 = XXXX110X

	OFF	RVS	CHR$
@	64 0	192 128	192 64
A	65 1	193 129	193 65
B	66 2	194 130	194 66
C	67 3	195 131	195 67
D	68 4	196 132	196 68
E	69 5	197 133	197 69
F	70 6	198 134	198 70
G	71 7	199 135	199 71
H	72 8	200 136	200 72
I	73 9	201 137	201 73
J	74 10	202 138	202 74
K	75 11	203 139	203 75
L	76 12	204 140	204 76
M	77 13	205 141	205 77
N	78 14	206 142	206 78
O	79 15	207 143	207 79
RETURN			141 13

	OFF	RVS	CHR$
P	80 16	208 144	208 80
Q	81 17	209 145	209 81
R	82 18	210 146	210 82
S	83 19	211 147	211 83
T	84 20	212 148	212 84
U	85 21	213 149	213 85
V	86 22	214 150	214 86
W	87 23	215 151	215 87
X	88 24	216 152	216 88
Y	89 25	217 153	217 89
Z	90 26	218 154	218 90
[	91 27	219 155	219 91
\	92 28	220 156	220 92.
]	93 29	221 157	221 93
↑	94 30	222 158	222 94
←	95 31	223 159	223 95
CLR SCREEN HOME			147 19
OFF RVS ON			146 18

	OFF	RVS	CHR$
SP	96 32	224 160	160 32
!	97 33	225 161	161 33
"	98 34	226 162	162 34
#	99 35	227 163	163 35
$	100 36	228 164	164 36
%	101 37	229 165	165 37
&	102 38	230 166	166 38
'	103 39	231 167	167 39
(	104 40	232 168	168 40
)	105 41	233 169	169 41
*	106 42	234 170	170 42
+	107 43	235 171	171 43
,	108 44	236 172	172 44
−	109 45	237 173	173 45
.	110 46	238 174	174 46
/	111 47	239 175	175 47
CURSOR ↑↓			145 17
CURSOR ←→			157 29

	OFF	RVS	CHR$
0	112 48	240 176	176 48
1	113 49	241 177	177 49
2	114 50	242 178	178 50
3	115 51	243 179	179 51
4	116 52	244 180	180 52
5	117 53	245 181	181 53
6	118 54	246 182	182 54
7	119 55	247 183	183 55
8	120 56	248 184	184 56
9	121 57	249 185	185 57
:	122 58	250 186	186 58
;	123 59	251 187	187 59
<	124 60	252 188	188 60
=	125 61	253 189	189 61
>	126 62	254 190	190 62
?	127 63	255 191	191 63
INST DEL			148 20
RUN STOP			131 3

Lower Case Mode: Location 59468 / XXXX110X, Same Except 193 to 218 Prints as Lower Case a to z Plus Different Graphics:

▨	105	233	169
✓	122	250	186
▦	94	222	222
▧	95	223	223

0: End of line
1-31: unused
32: space
33: !
34: "
35: #
36: $
37: %
38: &
39: '
40: (
41:)
42: *
43: +
44: ,
45: -
46: .
47: /
48: 0
49: 1
50: 2
51: 3
52: 4
53: 5
54: 6
55: 7
56: 8
57: 9
58: :
59: ;
60: <
61: =
62: >
63: ?
64: @
65: A
66: B
67: C
68: D
69: E
70: F
71: G
72: H
73: I
74: J
75: K
76: L
77: M
78: N
79: O
80: P
81: Q
82: R
83: S
84: T
85: U
86: V
87: W
88: X

89: Y
90: Z
91: [
92: \
93:]
94: ↑
95: ←
96: space
97: !
98: "
99: #
100: $
101: %
102: &
103: '
104: (
105:)
106: *
107: +
108: ,
109: -
110: .
111: /
112: 0
113: 1
114: 2
115: 3
116: 4
117: 5
118: 6
119: 7
120: 8
121: 9
122: :
123: ;
124: <
125: =
126: >
127: ?
128: END
129: FOR
130: NEXT
131: DATA
132: INPUT#
133: INPUT
134: DIM
135: READ
136: LET
137: GOTO
138: RUN
139: IF
140: RESTORE
141: GOSUB
142: RETURN
143: REM
144: STOP
145: ON
146: WAIT
147: LOAD

148: SAVE
149: VERIFY
150: DEF
151: POKE
152: PRINT#
153: PRINT
154: CONT
155: LIST
156: CLR
157: CMD
158: SYS
159: OPEN
160: CLOSE
161: GET
162: NEW
163: TAB(
164: TO
165: FN
166: SPC(
167: THEN
168: NOT
169: STEP
170: +
171: -
172: *
173: /
174: ↑
175: AND
176: OR
177: >
178: =
179: <
180: SGN
181: INT
182: ABS
183: USR
184: FRE
185: POS
186: SQR
187: RND
188: LOG
189: EXP
190: COS
191: SIN
192: TAN
193: ATN
194: PEEK
195: LEN
196: STR$
197: VAL
198: ASC
199: CHR$
200: LEFT$
201: RIGHT$
202: MID$
203-254: unused
255: π

Basic Commands and Statements

COMMAND/ STATEMENT	EXAMPLE	PURPOSE
CLR	CLR	Sets variables to zero or null.
CMD	CMD D	Keep IEEE device D open to monitor bus.
CONT	CONT	Continue program execution after a STOP command. No program changes permitted.
GOTO	GOTO L	Continue program execution at line L after a STOP command. Program changes are permitted.
FRE	PRINT FRE (0)	Returns number of bytes of available memory.

Basic Commands and Statements (Continued)

COMMAND/ STATEMENT	EXAMPLE	PURPOSE
DATA	10 DATA 1,2,3,4	Specifies data to be read from left to right.
	20 DATA TOM,SUE	Alphabetics do not need to be enclosed in quotes.
	30 DATA "TOM DOE"	If strings contain spaces, commas, colons, or graphic characters, the string must be enclosed in quotes.
DIM	10 DIM A(n)	Specifies maximum number of elements in an array or matrix.
	20 DIM A(n,m,o,p)	Specifies maximum number of dimensions in an array.
	30 DIM A(n),B(m)	Number of arrays limited by memory.
	40 DIM A(N)	May be dimensioned dynamically.
	50 DIM A$(n)	Strings may be dimensioned.
END	999 END	Terminates program execution.
GET	10 GET C	Accepts single character from keyboard.
	20 GET C$	Accepts single string character from keyboard.
	30 GET #D,C	Accepts single character from specified logical file.
	40 GET #D,C$	Accepts specified single string character from logical file.
INPUT	10 INPUT A	Accepts value of A from keyboard.
	20 INPUT A$	Accepts value of string variable A from keyboard. The string does not have to be enclosed in quotes.
	30 INPUT A,A$,B,B$	Accepts specified values from keyboard.
	40 INPUT #D,A	Accepts value of A from logical file D.
	50 INPUT #D,A$	Accepts specified string from logical file D.
	60 INPUT #D,A,A$,B,B$	Accepts specified values and strings from logical file D. Strings do not have to be enclosed in quotes.
LOAD	10 LOAD	Loads next encountered program or file, on built-in tape unit, into PET's memory.
	20 LOAD "NAME"	Loads program or file NAME into memory from built-in tape unit.
	30 LOAD "NAME",D	Loads specified file NAME from device D.
OPEN	10 OPEN A	Opens logical file A for read only from built-in tape unit.
	20 OPEN A,D	Opens logical file A for read only from device D.
	30 OPEN A,D,C	Opens logical file A for command C from device D.
	40 OPEN A,D,C,"NAME"	Opens logical file A on device D. If device D accepts formatted files, file NAME is positioned for command.
POS	10 PRINT POS(0)	Prints next available print position (position of cursor on screen).
PRINT	10 PRINT A	Prints value of A on display screen.
	20 PRINT A$	Prints specified string on screen.
	30 PRINT A,A$	Prints specified values or strings on screen, beginning in next available print position (pre-TABbed positions are in columns 10,20,30,40, etc.).

COMMAND/ STATEMENT	EXAMPLE	PURPOSE
LIST	LIST	Lists current program.
	LIST -L	Lists current program through line L.
	LIST L-M	Lists lines L through M of current program.
	LIST L-	Lists current program from line L to end.
LOAD	LOAD	Loads next encountered program from built-in tape unit.
	LOAD "NAME"	Loads file NAME from built-in tape unit.
	LOAD "NAME," D	Loads file NAME from device D.
NEW	NEW	Deletes current program from memory, sets variables to zero.
PEEK	PEEK(A)	Returns byte value from address A.
POKE	POKE A,B	Loads byte B into address A.
PRINT	PRINT A	Prints value of A on display screen.
	PRINT A$	Prints specified string on screen.
	PRINT #D,A	Prints value of A on device D.
	PRINT #D,A$	Prints specified string on device D.
RUN	RUN	Begins execution of program at lowest line number.
	RUN L	Begins execution of program at line L.
SAVE	SAVE	Saves current program on built-in tape unit.
	SAVE "NAME"	Saves current file or program NAME on built-in tape unit.
	SAVE "NAME," D	Saves current program or file NAME on device D.
	SAVE "NAME," D,C	Saves file NAME on device D. C specifies EOF or EOT.
STOP	STOP	Stops program execution.
SYS	SYS(X)	Complete control of PET is transferred to a subsystem at decimal address contained in the argument.
TI$	TI$="HHMMSS"	Sets PET's internal clock to real time.
	PRINT TI	Displays number of 'jiffies' since PET was powered up or clock was zeroed. (A jiffy = 1/60 of a second.)
USR	USR(X)	Transfers program control to a program whose address is at locations 1 and 2. X is a parameter passed to and from the machine language program.
WAIT	WAIT A,B,C	Stops execution of BASIC until contents of A, ANDed with B and exclusive ORed with C, is not equal to zero. C is optional and defaults to zero.
CLOSE	10 CLOSE N	Closes logical file N.

Basic Commands and Statements (Continued)

COMMAND/ STATEMENT	EXAMPLE	PURPOSE
GOTO	10 GOTO L	Transfers control (jumps) to specified line, skipping over intervening lines.
GOSUB	10 GOSUB L	Begins execution of a subroutine which begins on a specified line.
ON . . . GOTO	10 ON A GOTO L,M,N	Transfers control to specified line (in this example, L,M, or N, depending on value of index A.
ON . . . GOSUB	10 ON A GOSUB L,M,N	Begins execution of subroutine which begins on line L,M, or N, depending on the value of index A.
RETURN	9990 RETURN	Subroutine exit; transfers control to the statement following most recent GOSUB directing transfer to the subroutine.

String Functions

FUNCTION	EXAMPLE	PURPOSE
ASC	10 A=ASC("XYZ")	Returns integer value corresponding to ASCII code of first character in string.
CHR$	10 A$=CHR$(N)	Returns character corresponding to ASCII code number.
LEFT$	10 ?LEFT$(X$,A)	Returns leftmost A characters from string.
LEN	10 ?LEN(X$)	Returns length of string.
MID$	10 ?MID$(X$,A,B)	Returns B characters from string, starting with the Ath character.
RIGHT$	10 ?RIGHT$(X$,A)	Returns rightmost A characters from string.
STR$	10 A$=STR$(A)	Returns string representation of number.
VAL	10 A=VAL(A$) 20 A=VAL("A")	Returns numeric representation of string. If string not numeric, returns "Ø".

ASC, LEN and VAL functions return numerical results. They may be used as part of an expression. Assignment statements are used here for examples only; other statement types may be used.

Arithmetic Functions

FUNCTION	EXAMPLE	PURPOSE
ABS	10 C=ABS(A)	Returns magnitude of argument without regard to sign.
ATN	10 C=ATN(A)	Returns arctangent of argument. C will be expressed in radians.
COS	10 C=COS(A)	Returns cosine of argument. A must be expressed in radians.
DEF FN	10 DEF FNA(B)=C*D	Allows user to define a function. Function label A must be a single letter; argument B is a dummy.

SYMBOL	EXAMPLE	PURPOSE
EXP	10 C=EXP(A)	Returns constant 'e' raised to power of the argument. In this example, e^A.
INT	10 C=INT(A)	Returns largest integer less than or equal to argument.
LOG	10 C=LOG(A)	Returns natural logarithm of argument. Argument must be greater than or equal to zero.
RND	10 C=RND(A)	Generates a random number between zero and one. If A is less than 0, the same random number is produced in each call to RND. If A = 0, the same sequence of random numbers is generated each time RND is called. If A is greater than 0, a new sequence is produced for each call to RND.
SGN	10 C=SGN(A)	Returns -1 if argument is negative, returns 0 if argument is zero, and returns +1 if argument is positive.
SIN	10 C=SIN(A)	Returns sine or argument. A must be expressed in radians.
SQR	10 C=SQR(A)	Returns square root of argument.
TAN	10 C=TAN(A)	Returns tangent of argument. A must be expressed in radians.

Arithmetic Operators

SYMBOL	EXAMPLE	PURPOSE
=	10 A=B 20 LET A=B	Assigns a value to a variable. Let is optional.
↑	30 PRINT A↑2	Exponentiation; in example, A^2.
/	35 C=A/8	Division
*	40 C=A*8	Multiplication
+	50 C=A+8	Addition
-	60 C=A-8	Subtraction
=	10 IF A=B THEN PRINT C	A 'equals' B.
<>	10 IF A<>B THEN C=4	A 'does not equal' B.
<	10 IF A<B THEN C$="X"	A 'is less than' B.
>	10 IF A>B THEN C$-D$+E$	A 'is greater than' B.
<=	10 IF A<=B THEN C=20	A 'is less than or equal to' B.

Arithmetic Operators (Continued)

SYMBOL	EXAMPLE	PURPOSE
>=	10 IF A>=B THEN C=D-1	A 'is greater than or equal to' B.
AND	10 IF A AND B THEN C=0	A and B must BOTH be true for statement 10 to be true.
OR	20 IF A OR B THEN C=90	A must be true or B must be true for statement 20 to be true.
NOT	30 IF NOT A THEN PRINT C	Expression is true if A is false.

**NOTE: The numerical values used in the evaluation of logical comparisons are: 'TRUE' is any non-zero number and 'FALSE' is zero.

Special Symbols, Commands and Statements

SYMBOLS, COMMANDS, STATEMENTS	EXAMPLE	PURPOSE
:	10A=1:B=2:C=3	Allows multiple statements on a line.
;	10PRINT A;B 20PRINT A$;B$	Allows same line printing. Elements are separated by 3 spaces. Allows same line printing. String elements are concatenated.
,	10PRINT A,B	Allows same line printing. Elements are separated and printed in pre-TABbed print positions (columns 10,20,30, etc.)
,	LOAD "NAME," D	Separates elements in LOAD, SAVE, OPEN, and VERIFY.
?	10?A	Abbreviation for PRINT. Stores as one character; lists as word PRINT.
$	10A$="ABCDEFG"	String identifier.
%	10A%=INT(X)	Integer identifier.
"	10A$="ABCDEF"	String enclosures.
carriage return		Must follow every command, statement, or data entry; causes cursor to return to leftmost position on next lowest line. Signals "END OF INPUT LINE."
π		Value of Pi: 3.1415927.

I/O Commands

SYMBOL	COMMAND	PURPOSE

L=1-255
C=0: READ
C=1: WRITE
C=2: WRITE AND PUT EOT at end of file.
D=1 CASSETTE
D=2 2ND CASSETTE
D=4-15 IEEE BUSS
	OPEN L,D,C	Note: PET will not read past an EOT (end of tape) marker.

Table Status Word (ST) values correlated with tape cassette, unit and IEEE bus read/write errors.

ST Bit Position	ST Numeric Value	Cassette Read	IEEE R/W	Tape Verify + Load
0	1		Time out on write	
1	2		Time out on read	
2	4	Short block		Short block
3	8	Long block		Long block
4	16	Unrecoverable read error		Any mismatch
5	32	Checksum error		Checksum error
6	64	End of file	EOI line	
7	-128	End of tape	Device not present	End of tape

FROM PET MAIN LOGIC ASSEMBLY BOARD

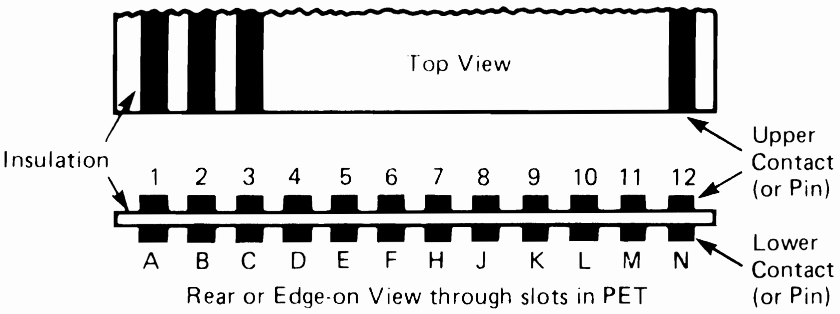


Figure 1-2. Simplified views of edge connectors J1 and J2 to illustrate contact identification convention.

Table: Parallel user port information.
PET pin identification characters, the corresponding → signal labels and their descriptions.

Pin Identification Character	Signal Label	Signal Description
1	Ground	Digital ground.
2	T.V. Video	Video output used for external display, used in diagnostic routine for verifying the video circuit to the display board.
3	IEEE-SRQ	Direct connection to the SRQ signal on the IEEE-488 port. It is used in verifying operation of the SRQ in the diagnostic routine.
4	IEEE-EOI	Direct connection to the EOI signal on the IEEE-488 port. It is used in verifying operation of the EOI in the diagnostic routine.
5	Diagnostic Sense	When this pin is held low during power up the PET software jumps to the diagnostic routine, rather than the BASIC routine.
6	Tape #1 READ	Used with the diagnostic routine to verify cassette tape #1 read function.
7	Tape #2 READ	Used with the diagnostic routine to verify cassette tape #2 read function.
8	Tape Write	Used with the diagnostic routine to verify operation of the WRITE function of both cassette ports.
9	T.V. Vertical	T.V. vertical sync signal verified in diagnostic. May be used for external TV display.
10	T.V. Horizontal	T.V. horizontal signal verified in diagnostic may be used for TV display.
11, 12	GND	Digital ground.
A	GND	Digital ground.
B	CA1	Standard edge sensitive input of 6522VIA.
C	PAØ	Input/output lines to peripherals, and can be programmed independently of each other for input or output.
D	PA1	
E	PA2	
F	PA3	
H	PA4	
J	PA5	
K	PA6	
L	PA7	
M	CB2	Special I/O pin of VIA.
N	GND	Digital ground.

Table: Second cassette interface port.
PET pin identification characters, labels and associated descriptions.

Note A-1, B-2, etc., imply a pin A to pin1, pin B to pin 2, connection. In some special units, pins 1 through 6 were not connected.

Pin Identification Characters	Label	Description
A-1	GND	Digital ground.
B-2	+5	Positive 5 volts to operate cassette circuitry only.
C-3	Motor	Computer controlled positive 6 volts for cassette motor.
D-4	Read	Read line from cassette.
E-5	Write	Write line to cassette.
F-6	Sense	Monitors closure of mechanical switch on cassette when any button is pressed.

Table: IEEE standard connectors

Connector Manufacturer	Identifier	Description
Cinch	571Ø24Ø	Solder-plug
Cinch	572Ø24Ø	Solder-receptacle
Amp	5523Ø1-1	Insulation displacement plug
Amp	5523Ø5-1	Insulation displacement receptacle

Table: Receptacles recommended for PET IEEE-488 connectors or parallel user port.

Manufacturer	Part Number
Cinch	251-12-9Ø-16Ø
Sylvania	6AGØ1-12-1A1-Ø1
Amp	53Ø657-3
Amp	53Ø658-3
Amp	53Ø654-3

Table: A selection of suitable receptacles for connecting with the PET second cassette edge connector J3.

Manufacturer	Identifier
Sylvania	6AJØ7-6-1A1-Ø1
Viking	2KH6/1AB5
Viking	2KH6/9AB5
Viking	2KH6/21AB5
Amp	53Ø692-1
Sullins	ESM6-SREH
Cinch	25Ø-Ø6-9Ø-17Ø

Table: IEEE-488 bus signal.

Bus Group	Signal Abbrev.	Name	Functional Description
Manager	ATN	Attention	The PET (controller) sets this signal low while it is sending commands on the data bus. When ATN is low, only peripheral addresses and control messages are on the data bus. When ATN is high, only preveiously assigned devices can transfer data.
Transfer	DAV	Data Valid	When DAV is low, this signifies that data is valid on data bus.
Manager	EOI	End or Identify	When the last byte of data is being transferred, the talker has the option of setting EOI low. The PET always sets EOI low while the last data byte is being transferred from the PET.
Manager	IFC	Interface Clear	The PET sends its internal reset signal as IFC low (true) to initialize all devices to the idle state. When PET is switched on or reset, IFC goes low for about 100 milliseconds.
Transfer	NDAC	Data Not Accepted	This signal is held low (true) by the listener while reading. When the data byte has been read, the listener sets NDAC high. This signals the talker that data has been accepted.
Transfer	NRFD	Not Ready for Data	When NRFD is low (true), one or more listeners are not ready for the *next* byte of data. When all devices are ready, NRFD goes high.
Manager	SRQ	Service Request	Not implemented in BASIC, but available to the PET user.
Manager	REN	Remote Enable	REN is held low by the bus controller. The PET has a pin grounded that keeps REN permanently low.
Data	DI01-8	Data input/ output lines 1 through 8	These signals represent the bits of information on the data bus. When a DI0 signal is low, it represents 1 and when high Ø.
General	GND	Ground	Ground connections: There are six control and management signal ground returns, one data signal ground return and one chassis shield ground lead.

PET Pin Characters	Standard IEEE Connector Pin Numbers	IEEE Signal Mnemonic	Signal Definition/Label
Upper Pins			
1	1	DI01	Data input/output line #1
2	2	DI02	Data input/output line #2
3	3	DI03	Data input/output line #3
4	4	DI04	Data input/output line #4
5	5	EOI	End or identify
6	6	DAV	Data valid
7	7	NRFD	Not ready for data
8	8	NDAC	Data not accepted
9	9	IFC	Interface clear
10	10	SRQ	Service request
11	11	ATN	Attention
12	12	GND	Chassis ground and IEEE cable shield drain wire
Lower Pins			
A	13	DI05	Data input/output line #5
B	14	DI06	Data input/output line #6
C	15	DI07	Data input/output line #7
D	16	DI08	Data input/output line #8
E	17	REN	Remote enable
F	18	GND	DAV ground
H	19	GND	NRFD ground
J	20	GND	NDAC ground
K	21	GND	IFC ground
L	22	GND	SRQ ground
M	23	GND	ATN ground
N	24	GND	Data ground (DI01-8)

Original ROM Board

Table: **Memory expansion connector. PET pin numbers. Line labels and line descriptions.**

Side A Pin Numbers	Line Labels	Line Description
A1	BAØ	Address bit Ø, used for memory expansion. Buffered.
A2	BA1	Address bit 1, used for memory expansion. Buffered.
A3	BA2	Address bit 2, used for memory expansion. Buffered.
A4	BA3	Address bit 3, used for memory expansion. Buffered.
A5	BA4	Address bit 4, used for memory expansion. Buffered.
A6	BA5	Address bit 5, used for memory expansion. Buffered.
A7	BA6	Address bit 6, used for memory expansion. Buffered.
A8	BA7	Address bit 7, used for memory expansion. Buffered.
A9	BA8	Address bit 8, used for memory expansion. Buffered.
A1Ø	BA9	Address bit 9, used for memory expansion. Buffered.
A11	BA1Ø	Address bit 1Ø, used for memory expansion. Buffered.
A12	BA11	Address bit 11, used for memory expansion. Buffered.
A13	NC	No connection.
A14	NC	No connection.
A15	NC	No connection.
A16	$\overline{\text{SEL 1}}$	4K byte page address select for memory locations 1ØØØ-1FFF.
A17	$\overline{\text{SEL 2}}$	4K byte page address select for memory locations 2ØØØ-2FFF.
A18	$\overline{\text{SEL 3}}$	4K byte page address select for memory locations 3ØØØ-3FFF.
A19	$\overline{\text{SEL 4}}$	4K byte page address select for memory locations 4ØØØ-4FFF.
A2Ø	$\overline{\text{SEL 5}}$	4K byte page address select for memory locations 5ØØØ-5FFF.
A21	$\overline{\text{SEL 6}}$	4K byte page address select for memory locations 6ØØØ-6FFF.
A22	$\overline{\text{SEL 7}}$	4K byte page address select for memory locations 7ØØØ-7FFF.
A23	$\overline{\text{SEL 9}}$	4K byte page address select for memory locations 9ØØØ-9FFF.
A24	$\overline{\text{SEL A}}$	4K byte page address select for memory locations AØØØ-AFFF.
A25	$\overline{\text{SEL B}}$	4K byte page address select for memory locations BØØØ-BFFF.
A26	NC	No connection.
A27	$\overline{\text{RES}}$	Reset for 6502 microprocessor. Note: connected to 74LS00 output.
A28	$\overline{\text{IRQ}}$	Interrupt request line to the microprocessor.
A29	BØ2	Buffered phase 2 clock.
A3Ø	R/W	Buffered read/write from 6502 microprocessor.
A31	NC	No connection.
A32	NC	No connection.
A33	BDØ	Data bit Ø. Buffered.
A34	BD1	Data bit 1. Buffered.
A35	BD2	Data bit 2. Buffered.
A36	BD3	Data bit 3. Buffered.
A37	BD4	Data bit 4. Buffered.
A38	BD5	Data bit 5. Buffered.
A39	BD6	Data bit 6. Buffered.
A4Ø	BD7	Data bit 7. Buffered.

Side B (top) pins are
Ground Returns for
corresponding Side A
(bottom) pins.

Upgrade ROM Board

Daughter board power connections

● ● ○ ● ● ● ●
1 2 3 4 5 6 7

J1Ø

pin #	function
1	-5V Raw power
2	-5V Raw power
3	key
4	+12V Raw power
5	+12V Raw power
6	Ground
7	Ground

● ○ ○ ● ● ● ●
1 2 3 4 5 6 7

J11

pin #	function
1	+9 unregulated
2	key
3	key
4	+9 unregulated
5	ground
6	9 unregulated
7	Ground

Manufacturer	contact grid	identifier
Spectra-strip	2x7	8Ø2-1Ø4
Spectra-strip	2x7	8Ø2-114
Spectra-strip	2x25	8Ø2-Ø5Ø
Spectra-strip	2x25	8Ø2-15Ø
Circuit-Assembly	2x7	CA-14-IDSC
Circuit-Assembly	2x25	CA-50-IDSC

Table 7.12. A selection of suitable receptacles for connecting with PET daughter board pin connectors J4, J9, J1Ø, and J11

Available at Deskin Sales
77 - D Steelcase Rd. W.
Toronto, Ont.
(416) 495 - 1412

Memory expansion bus

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Side A ●
Side B ●

J9

pin #	function	pin #	function
Side A1	ground	14	BA12
2	BAØ	15	BA13
3	BA1	16	BA14
4	BA2	17	BA15
5	BA3	13	SYNC
6	BA4	19	$\overline{IRQ}$
7	BA5	20	Memory Management
8	BA6	21	BØ2
9	BA7	22	BR/W
10	BA8	23	$\overline{BR/W}$
11	BA9	24	$\overline{DMA}$
12	BA1Ø	25	ground
13	BA11	Side B1-25	ground

J4

pin #	function	pin #	function
Side A1	ground	14	$\overline{SEL\ 6}$
2	BDØ	15	$\overline{SEL\ 7}$
3	BD1	16	$\overline{SEL\ 8}$
4	BD2	17	$\overline{SEL\ 9}$
5	BD3	18	$\overline{SEL\ A}$
6	BD4	19	$\overline{SEL\ B}$
7	BD5	20	$\overline{CAS}$
8	BD6	21	$\overline{RAS}$
9	BD7	22	$\overline{RES}$
10	$\overline{SEL\ 2}$	23	$\overline{RDY}$
11	$\overline{SEL\ 3}$	24	$\overline{NMI}$
12	$\overline{SEL\ 4}$	25	ground
13	$\overline{SEL\ 5}$	Side B1-25	ground

Connector Pin Numbers	Line Labels	Line Description
J9-1	GND	Logic Ground
J9-2	BAØ	Address bit Ø, used for memory expansion. Buffered.
J9-3	BA1	Address bit 1, used for memory expansion. Buffered.
J9-4	BA2	Address bit 2, used for memory expansion. Buffered.
J9-5	BA3	Address bit 3, used for memory expansion. Buffered.
J9-6	BA4	Address bit 4, used for memory expansion. Buffered.
J9-7	BA5	Address bit 5, used for memory expansion. Buffered.
J9-8	BA6	Address bit 6, used for memory expansion. Buffered.
J9-9	BA7	Address bit 7, used for memory expansion. Buffered.
J9-25	GND	Logic Ground.
J9-10	BA8	Address bit 8, used for memory expansion. Buffered.
J9-11	BA9	Address bit 9, used for memory expansion. Buffered.
J9-12	BA10	Address bit 1Ø, used for memory expansion. Buffered.
J9-13	BA11	Address bit 11, used for memory expansion. Buffered.
J9-14	BA12	Address bit 12, used for memory expansion. Buffered.
J9-15	BA13	Address bit 13, used for memory expansion. Buffered.
J9-16	BA14	Address bit 14, used for memory expansion. Buffered.
J9-17	BA15	Address bit 15, used for memory expansion. Buffered.
J9-19	$\overline{IRQ}$	Interrupt request line to the microprocessor.
J9-21	BØ2	Buffered phase 2 clock.
J9-22	BR/W	Buffered read/write from 6502 microprocessor.
J4-10	$\overline{SEL\ 2}$	4K byte page address select for memory locations 2ØØØ-2FFF.
J4-11	$\overline{SEL\ 3}$	4K byte page address select for memory locations 3ØØØ-3FFF.
J4-12	$\overline{SEL\ 4}$	4K byte page address select for memory locations 4ØØØ-4FFF.
J4-13	$\overline{SEL\ 5}$	4K byte page address select for memory locations 5ØØØ-5FFF.
J4-14	$\overline{SEL\ 6}$	4K byte page address select for memory locations 6ØØØ-6FFF.
J4-15	$\overline{SEL\ 7}$	4K byte page address select for memory locations 7ØØØ-7FFF.
J4-16	$\overline{SEL\ 8}$	4K byte page address select for memory locations 8ØØØ-8FFF.
J4-17	$\overline{SEL\ 9}$	4K byte page address select for memory locations 9ØØØ-9FFF.
J4-18	$\overline{SEL\ A}$	4K byte page address select for memory locations AØØØ-AFFF.
J4-19	$\overline{SEL\ B}$	4K byte page address select for memory locations BØØØ-Bfff.
J4-22	$\overline{RES}$	Reset for 6502 microprocessor. Note: connected to 74LS00 output.
J4-23	$\overline{RDY}$	Ready line to the microprocessor.
J4-24	$\overline{NMI}$	Non maskable interrupt to microprocessor.
J9-1	GND	Logic ground
J4-2	BDØ	Data bit Ø. Buffered.
J4-3	BD1	Data bit 1. Buffered.
J4-4	BD2	Data bit 2. Buffered.
J4-5	BD3	Data bit 3. Buffered.
J4-6	BD4	Data bit 4. Buffered.
J4-7	BD5	Data bit 5. Buffered.
J4-8	BD6	Data bit 6. Buffered.
J4-9	BD7	Data bit 7. Buffered.
J4-20	$\overline{RAS}$	Dynamic RAM RAS.
J4-21	$\overline{CAS}$	Dynamic RAM CAS.
J4-25	GND	Logic Ground.

Note that the 40 top edge "B" connections (or pins) are ground returns for the corresponding 40 lower edge "A" connections.

ADDRESSING MODES

ACCUMULATOR ADDRESSING - This form of addressing is represented with a one byte instruction, implying an operation on the accumulator.

IMMEDIATE ADDRESSING - In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

ABSOLUTE ADDRESSING - In absolute addressing, the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits. Thus, the absolute addressing mode allows access to the entire 65K bytes of addressable memory.

ZERO PAGE ADDRESSING - The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. Careful use of the zero page can result in significant increase in code efficiency.

INDEXED ZERO PAGE ADDRESSING - (X, Y indexing) - This form of addressing is used in conjunction with the index register and is referred to as "Zero Page, X" or "Zero Page, Y". The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally due to the "Zero Page" addressing nature of this mode, no carry is added to the high order 8 bits of memory and crossing of page boundaries does not occur.

INDEXED ABSOLUTE ADDRESSING - (X, Y indexing) - This form of addressing is used in conjunction with X and Y index register and is referred to as "Absolute, X", and "Absolute, Y". The effective address is formed by adding the contents of X or Y to the address contained in the second and third bytes on the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

IMPLIED ADDRESSING - In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

RELATIVE ADDRESSING - Relative addressing is used only with branch instructions and establishes a destination for the conditional branch.

The second byte of the instruction becomes the operand which is an "Offset" added to the contents of the lower eight bits of the program counter when the counter is set at the next instruction. The range of the offset is -128 to +127 bytes from the next instruction.

INDEXED INDIRECT ADDRESSING - In indexed indirect addressing (referred to as (indirect,X)), the second byte of the instruction is added to the contents of the X index register, discarding the carry. The result of the addition points to a memory location on page zero whose contents is the low order eight bits of the effective address. The next memory location in page zero contains the high order eight bits of the effective address. Both memory locations specifying the high and low order bytes of the effective address must be in page zero.

INDIRECT INDEXED ADDRESSING - In indirect indexed addressing (referred to as (Indirect),Y), the second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of the Y index register, the result being the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

ABSOLUTE INDIRECT - The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that memory location is contained in the third byte of the instruction. The contents of the fully specified memory location is the low order byte of the effective address which is loaded into the sixteen bits of the program counter.

Hexadecimal Conversion Table

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

	IMM	ZPAG	Z,X	(I,X)	(I),Y	ABS	A,X	A,Y
	2	2	2	2	2	3	3	3
ORA	09	05	15	01	11	0D	1D	19
AND	29	25	35	21	31	2D	3D	39
EOR	49	45	55	41	51	4D	5D	59
ADC	69	65	75	61	71	6D	7D	79
STA		85	95	81	91	8D	9D	99
LDA	A9	A5	B5	A1	B1	AD	BD	B9
CMP	C9	C5	D5	C1	D1	CD	DD	D9
SBC	E9	E5	F5	E1	F1	ED	FD	F9

Op Code ends in -1, -5, -9, or -D

	IMM	ZPAG	Z,X	Z,Y	ABS	A,X	A,Y
	2	2	2	2	3	3	3
ASL		06	16		0E	1E	
ROL		26	36		2E	3E	
LSR		46	56		4E	5E	
ROR		66	76		6E	7E	
STX		86		96	8E		
LDX	A2	A6		B6	AE		BE
DEC		C6	D6		CE	DE	
INC		E6	F6		EE	FE	

Op Code ends in -2, -6, or -E

BPL	10	BMI	30
BVC	50	BVS	70
BCC	90	BCS	B0
BNE	D0	BEQ	F0

Branches -0

	ABS	(IND)
JSR	20	
JMP	4C	6C

Jumps

	IMM	ZPAG	Z,X	ABS	A,X
	2	2	2	3	3
BIT		24		2C	
STY		84	94	8C	
LDY	A0	A4	B4	AC	BC
CPY	C0	C4		CC	
CPX	E0	E4		EC	

Misc. -0, -4, -C

	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	BRK				RTI		RTS									
-8	PHP	CLC	PLP	SEC	PHA	CLI	PLA	SEI	DEY	TYA	TAY	CLV	INY	CLD	INX	SED
-A	ASL-A		ROL-A		LSR-A		ROR-A		TXA	TXS	TAX	TSX	DEX		NOP	

Single-byte Op Codes -0, -8, -A

MCS6501-MCS6505 MICROPROCESSOR INSTRUCTION SET – ALPHABETIC SEQUENCE

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

INSTRUCTION ADDRESSING MODES AND RELATED EXECUTION TIMES (in clock cycles)

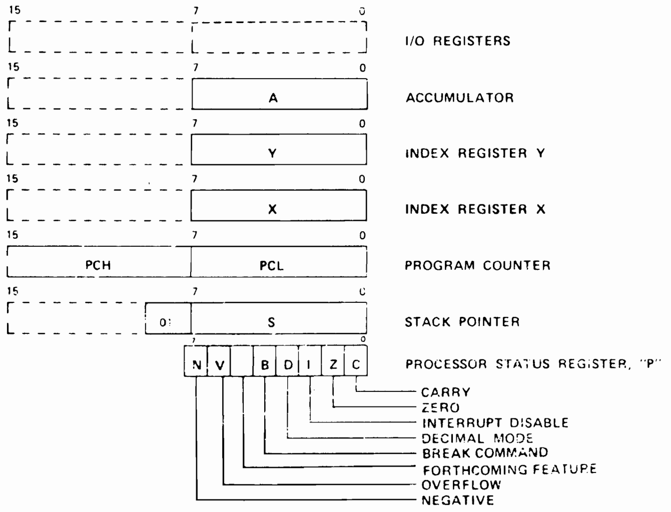
	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
ADC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
AND	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
ASL	2	.	5	6	.	6	7	.	.	.	.	.	.
BCC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BCS	.	.	.	.	.	.	.	.	.	2**	.	.	.
BEQ	.	.	.	.	.	.	.	.	.	2**	.	.	.
BIT	.	.	3	.	.	4	.	.	.	.	.	.	.
BMI	.	.	.	.	.	.	.	.	.	2**	.	.	.
BNE	.	.	.	.	.	.	.	.	.	2**	.	.	.
BPL	.	.	.	.	.	.	.	.	.	2**	.	.	.
BRK	.	.	.	.	.	.	.	.	.	.	.	.	.
BVC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BVS	.	.	.	.	.	.	.	.	.	2**	.	.	.
CLC	.	.	.	.	.	.	.	.	2	.	.	.	.
CLD	.	.	.	.	.	.	.	.	2	.	.	.	.
CLI	.	.	.	.	.	.	.	.	2	.	.	.	.
CLV	.	.	.	.	.	.	.	.	2	.	.	.	.
CMP	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
CPX	.	2	3	.	.	4	.	.	.	.	.	.	.
CPY	.	2	3	.	.	4	.	.	.	.	.	.	.
DEC	.	.	5	6	.	6	7	.	.	.	.	.	.
DEX	.	.	.	.	.	.	.	.	2	.	.	.	.
DEY	.	.	.	.	.	.	.	.	2	.	.	.	.
EOR	.	2	3	4	.	4	4*	4*	.	.	6	5	.
INC	.	.	5	6	.	6	7	.	.	.	.	.	.
INX	.	.	.	.	.	.	.	.	2	.	.	.	.
INY	.	.	.	.	.	.	.	.	2	.	.	.	.
JMP	.	.	.	.	.	3	.	.	.	.	.	.	5

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
JSR	.	.	.	.	.	6	.	.	.	.	.	.	.
LDA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
LDX	.	2	3	.	4	4	.	4*	.	.	.	.	.
LDY	.	2	3	4	.	4	4*	.	.	.	.	.	.
LSR	2	.	5	6	.	6	7	.	.	.	.	.	.
NOP	.	.	.	.	.	.	.	.	2	.	.	.	.
ORA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
PHA	.	.	.	.	.	.	.	.	3	.	.	.	.
PHP	.	.	.	.	.	.	.	.	3	.	.	.	.
PLA	.	.	.	.	.	.	.	.	4	.	.	.	.
PLP	.	.	.	.	.	.	.	.	4	.	.	.	.
ROL	2	.	5	6	.	6	7	.	.	.	.	.	.
ROR	2	.	5	6	.	6	7	.	.	.	.	.	.
RTI	.	.	.	.	.	.	.	.	6	.	.	.	.
RTS	.	.	.	.	.	.	.	.	6	.	.	.	.
SBC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
SEC	.	.	.	.	.	.	.	.	2	.	.	.	.
SED	.	.	.	.	.	.	.	.	2	.	.	.	.
SEI	.	.	.	.	.	.	.	.	2	.	.	.	.
STA	.	.	3	4	.	4	5	5	.	.	6	6	.
STX*	.	.	3	.	4	4	.	.	.	.	.	.	.
STY**	.	.	3	4	.	4	.	.	.	.	.	.	.
TAX	.	.	.	.	.	.	.	.	2	.	.	.	.
TAY	.	.	.	.	.	.	.	.	2	.	.	.	.
TSX	.	.	.	.	.	.	.	.	2	.	.	.	.
TXA	.	.	.	.	.	.	.	.	2	.	.	.	.
TXS	.	.	.	.	.	.	.	.	2	.	.	.	.
TYA	.	.	.	.	.	.	.	.	2	.	.	.	.

* Add one cycle if indexing across page boundary

** Add one cycle if branch is taken, Add one additional if branching operation crosses page boundary

PROGRAMMING MODEL MCS650X



* Solid line indicates currently available features
Dashed line indicates forthcoming members of family

Table: Code assignments for "Command Mode" of operation.

(SENT AND RECEIVED WITH ATN TRUE)

b7					0	① MSG	0	MSG	0	MSG	0	MSG	1	MSG	1	MSG	1	MSG	1	MSG
b6					0		0		1		1		0		0		1		1	
b5					0		1		0		1		0		1		0		1	
b4 ↓	**b3** ↓	**b2** ↓	**b1** ↓	**COLUMN →** **ROW ↓**	**0**		**1**		**2**		**3**		**4**		**5**		**6**		**7**	
0	0	0	0	0	NUL		DLE		SP	MLA ASSIGNED TO DEVICE	0	MLA ASSIGNED TO DEVICE	@	MTA ASSIGNED TO DEVICE	P	MTA ASSIGNED TO DEVICE	`	MEANING DEFINED BY PCG CODE	p	MEANING DEFINED BY PCG CODE
0	0	0	1	1	SOH	GTL	DC1	LLO	!		1		A		Q		a		q	
0	0	1	0	2	STX		DC2		"		2		B		R		b		r	
0	0	1	1	3	ETX		DC3		#		3		C		S		c		s	
0	1	0	0	4	EOT	SDC	DC4	DCL	$		4		D		T		d		t	
0	1	0	1	5	ENQ	PPC ③	NAK	PPU	%		5		E		U		e		u	
0	1	1	0	6	ACK		SYN		&		6		F		V		f		v	
0	1	1	1	7	BEL		ETB		'		7		G		W		g		w	
1	0	0	0	8	BS	GET	CAN	SPE	(		8		H		X		h		x	
1	0	0	1	9	HT	TCT	EM	SPD	)		9		I		Y		i		y	
1	0	1	0	10	LF		SUB		*		:		J		Z		j		z	
1	0	1	1	11	VT		ESC		+		;		K		[		k		{	
1	1	0	0	12	FF		FS		,		<		L		\		l		¦	
1	1	0	1	13	CR		GS		-		=		M		]		m		}	
1	1	1	0	14	SO		RS		.		>		N		^		n		~	
1	1	1	1	15	SI		US		/		?	■	O		—	■	o		DEL	

④

ADDRESSED COMMAND GROUP (ACG) — columns 0 and MSG

UNIVERSAL COMMAND GROUP (UCG) — columns 1 and MSG

LISTEN ADDRESS GROUP (LAG) — columns 2 and 3

TALK ADDRESS GROUP (TAG) — columns 4 and 5

PRIMARY COMMAND GROUP (PCG) — columns 0 through 5

SECONDARY COMMAND GROUP (SCG) — columns 6 and 7

NOTES: ① MSG = INTERFACE MESSAGE

② b_1 = DI01 ... b_7 = DI07

③ REQUIRES SECONDARY COMMAND

④ DENSE SUBSET (COLUMN 2 THROUGH 5). ALL CHARACTERS USED IN BOTH COMMAND & DATA MODES.