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 not 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 particularly 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

REVIEWS IN FUTURE TRANSACTOR PUBLICATIONS

The Transactor will begin, on a limited basis, to review significant Pet related hardware and software products. The following is an outline of items under consideration:

Software: Programs

Adventures	Automated Simulations
	Creative Computing Software
Assemblers	Moser's MAE
	Skyles' MacroTea
Basic Extensions	Programmer's Toolkit
	Softside Structured Basic
Compilers	Graphics
	Music
Micro-Go	aid-com
Music Generation	AB Computers
	Bonnycastle
	MTU/Chamberlain/Covitz
Osborne Accounts Rec/Pay	
Osborne General Ledger	
Visi-Calc Data Base Manager	
Wordprocessors	CBM Wordpros I, II & III
	CMC version 2
	Medit

Software: Books

New Osborne PET and IEEE publications	
PET' Machine Language Guide	Abacus
The PET Manual	G. Yob
New H.Sams 6502 Programming publication	
New Scelbi 6502 publication	

Hardware:

Bus expansion (S-100)	Betsi
	California Computer Systems
DAC's	AB Computers
	MTU
Digitizers	* Bit-Pad
	Presto-Digitizer
High resolution Graphics	MTU
Memory expansion	MTU
	Skyles
Modems	* Hayes
	Novation
	TNW
Music synthesis	Ackerman/G.I.Sound Generator
Rom expansion	Kobotek
	Skyles
	Small Systems Services, Inc.
Prom programming	Optimal Technology Inc.
Plotters	* Houston Instr.
* Sylvanhills	
RS-232-C	Computer Associates
	TNW
Speech recognition	* Hueristics
Speech synthesis	* Computalker
	* Votrax
Spectrum analysis	Eventide Clock Works
Video pen	Ez-Mark

* When available

CALL FOR ARTICLES / A PRIORIZATION OF INTERESTS

The following is an outline of PET related applications The Transactor would like to publish in future issues (many ideas have been noted in your correspondence to The Transactor). Please forward this page (or a copy) to indicate a specific application(s) you would like to see published soon. As an inducement we provide a life size PET poster (limited supply) for each reply. In reviewing this outline you may discover you are already working on a particular application and would like to share it with other Transactor readers (and perhaps, through feedback, learn something new in the process.)

CATEGORY	APPLICATION	PREFERENCE
Business/Finance	Accounts Rec/Pay & General Ledger for Canadian Practice	( )
	Data Base Management Techniques	( )
	Income Tax Records Maintenance	( )
	Income Tax Returns, Simple and Complex	( )
	Mailing Lists	( )
	Mathematics	( )
	Payroll Methods	( )
	Property Management	( )
	Real Estate Analysis	( )
	Sorting and Filing Techniques	( )
	Statistics for Business and Finance	( )
	Stock Market Analysis	( )
	Wordprocessors	( )
Commun./Media	Computer Aids for Vision, Hearing and Motive Impairment	( )
	Computer Bulletin Board Systems and Techniques	( )
	Dark-Room Micro-processor Applications	( )
	Encryptography	( )
	Language translation, English to French etc.	( )
	Modem Methods and Techniques	( )
	Motion Picture Camera & Projection Control Systems	( )
	Photography and Film Index Methods	( )
	Satelite Reception	( )
	Slow-scan TV Reception	( )
	Studio and Home Recording and Mixing Control	( )
	Telidon	( )
Education	Computer Aided Instruction	( )
	Foreign Language Instruction	( )
	School/College Student Timetable Scheduler	( )
	Student Attendance & Academic Performance Stats. Main.	( )
Games	Backgammon	( )
	Bridge	( )
	Checkers	( )
	Chess	( )
	Gaming	( )
	Go	( )
	Role playing	( )

CATEGORY	APPLICATION	PREFERENCE
Graphics	Animation	( )
	Colour	( )
	High Resolution	( )
	3D Simulation	( )
	Video Synthesis & digitizing	( )
	Character Recognition	( )
Household	AC Control Systems	( )
	Automated Kitchens	( )
	Automated Shopping list	( )
	Cheque Booking, Plain and Fancy	( )
	Consumer Expenditure Decision Maker	( )
	Effective Dieting	( )
	Energy Conservation Method & Techniques	( )
	Menu Planning/Nutrition	( )
	Periodical/Bibliography Indexing	( )
	Personal Finance Methods & Techniques	( )
	Personal Fitness	( )
	Security Systems, Fire, Burglary	( )
Languages	Basic Dialect and Translation Strategies	( )
	Basic Extensions	( )
	Basic Programming Techniques, Beginner & Advanced	( )
	Machine Language Programming, Beginner & Advanced	( )
	Other Languages:Compilers, Focal, Forth, Lisp, Pascal, Pilot	( )
Music	Analog Synthesizer Control	( )
	CB2-Port Music Generation Techniques	( )
	DAC Hardware and Software Techniques	( )
	Digital Synthesizers	( )
	Fundamentals of Music Theory	( )
	Music (sound) Spectrum Analysis	( )
	Sound Processing: Reverb, Echo, Phasing, Phlanging	( )
Peripherals	A Commodore Disk Primer	( )
	A Commodore Printer Primmer	( )
	IEEE Uses	( )
	Memory Expansion Bus Uses	( )
	Parallel User Port	( )
	RS-232-C Applications	( )
PET	Monitor Techniques & Extensions	( )
	More on Interrupts	( )
	PET Operating System Primmer	( )
Science&Industry	Computer Oscilliscopes	( )
	Digital Logic Circuit Analysis	( )
	Interfacing Analog Measuremnt Instruments	( )
	Mathematics for Electronics and Science	( )
	Measurement of Capacitance, Current, Resistance & Voltage	( )
Speech Synthesis	Computer Aids for Speech Impairment	( )
	Speech Recognition	( )
	Speech Synthesis	( )

Name ____

Address ____

____

Comments ____

____

____

I would like to contribute an article on ___________

____

____

****

____

____

____

The Transactor
Commodore Bsuiness Machines
3370 Pharmacy Ave.,
Agincourt, Ont., M1W 2K4

****

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
```

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.

MICRO-GO 9L: A REVIEW

W.T.Garbutt
Mississauga,Ontario
Canada

Last March I read an intriguing article by Milton Bradley on the game of GO (Creative Computing -March 1979 vol 5, no 3). At the time I made a mental note:who would be the first to meet the challenge of marketing a micro-computer version and for which system ?

Recently those questions were answered. Mr. H. Mueller, a teacher at the Oakville Ontario Canada campus of Sheridan College has the first commercial honors. Mr. Mueller's program is written primarily in Basic to run on an 8k or 16/32k PET, old or upgrade ROM .

First, for all the gentle readers who may not remember, a brief description of GO is in order. GO has a history at least as old as chess. Originating in the East it did not receive widespread exposure in the West until the end of WW II.

The game is played on a 19 x 19 board. Two opposing players add pieces, called stones, to the intersections of the playing grid. The move scenario is extremely simple. The object of the game is to occupy more intersections, surrounded empty intersections and prisoners than your opponent at the games end. An opponent can be captured and removed from the board by surrounding that stone horizontally and vertically (see Table 1).

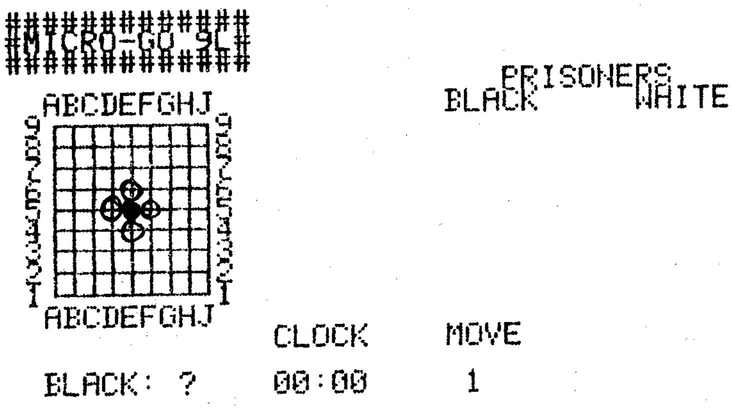


Table 1

A capture configuration:
Black's stone is captured

A player cannot make a move thhat results in an identical previous board position--the Rule of KO. They may however, pass and give up a stone. When both players pass in succession the game is over. The winner is the player with the most number of intersections, surrounded intersections and prisoners.

The perceptive reader will have noted the immense complexity of the game given the 19 x 19 playing grid (the permutations and combinations are staggering). They will also have noted that by stringing together intersections, called chaining, capture strategies are particularly complex and difficult. So difficult in fact that professional and amateur GO comprises some 17 levels (or Dan as they are called).

Now to Mr. Mueller's implementation.

Mr. Mueller uses a 9 x 9 grid. Ha!ha!you say , this certainly cannot compete with the 19 x 19 version. Of course not! What micro-computer can presently handle that complexity? However for the beginning GO player this implementation is ideal.

Let's take a look at the PET screen after GO has been loaded and initialized (all automatic and note the cassette is protected against copying but a back-up copy is provided. Mr. Mueller will of course replace a defective tape).

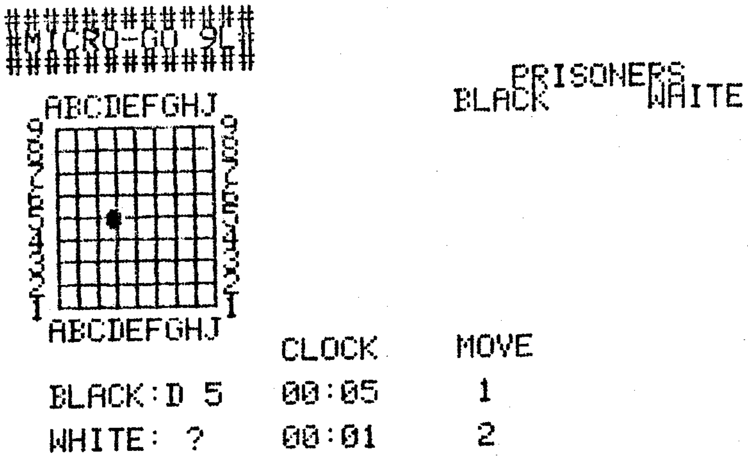


Table 2

PET screen on game start

As you can see in Table 2 the Pet screen is segmented into three distinct areas; the game board; the move, move clocks and current move number block; and the prisioner area.

The game starts with the computer, playing black, entering the cordinates of the first move. The player then selects the counter move, enters the cordinates (in any order), presses RETURN and the computer repeats the sequence. The computer will not allow an illegal move, say a violation of the Rule of KO or a move to an occupied intersection. Initially the computer responds in under a minute. However as the game progresses the computer's reaction time slows down. In the end game the computer can take as long as ten minutes! (An ideal length of time to read a few pages of a GO strategy text and improve one's knowledge for the next encounter.) Moves of both players are monitored on screen clock displays.

Mr. Mueller has allowed for computer or player handicaps by permitting the addition of extra stones (an advantage) at the beginning of the game. In addition the board can be set up in a pre-determined configuration, after a game interruption or for analysis of a particular move strategy.

The game comes with complete and above average documentation. It explains in considerable detail all operating instructions as well as stepping through highlights of a typical game (complete with screen printouts).

As I mentioned earlier Mr. Mueller has written the first implementation in Basic. He has used a 'brute force' approach. Naturally this approach requires some concessions. The major penalty is speed. As the game progresses the computer takes longer to respond. Further as Mr. Mueller has used the screen board as an array (to conserve memory) the prisoner capture routine requires additional time after each computer move.

Close examination of the screen display clocks will reveal small hesitancies. These are created by interrupts while the computer analyses the move strategy. They in no way affect the clock accuracies.

Incorporated in the algorithm is a thoughtful 'stop key disable' provision to prevent game stoppage in the event of accidental stop key contact.

For $18.95 the beginning GO player will acquire a quality program with good documentation. For that investment they will find a willing partner and the means to develop a sound foundation for advanced play. Naturally a quicker, machine language version, with a larger playing grid and perhaps a look ahead feature for levels of difficulty would appeal to the advanced GO enthusiast. How about it Mr. Mueller!

The program is available from:

"aidcom",
P.O. Box 165,
Clarkson Postal Station,
Mississauga, Ontario,
Canada L5J 3Y1
(Ontario residents please add 7% PST)

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.

S U P E R M O N 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 2000 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.

S U P E R M O N 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, 4501
```

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",08,0400,0D45

...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
.:  0D26 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.

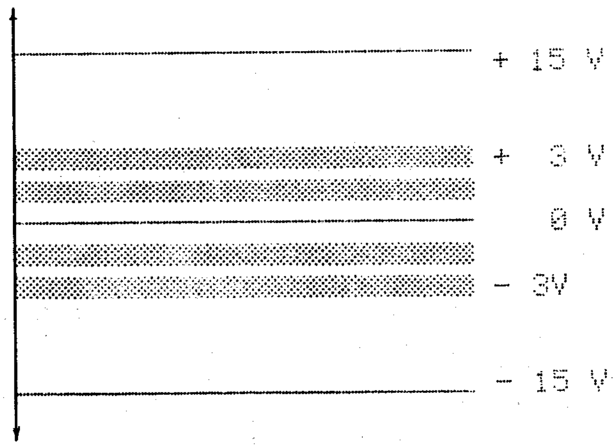


FIGURE 1

BIT REPRESENTATION

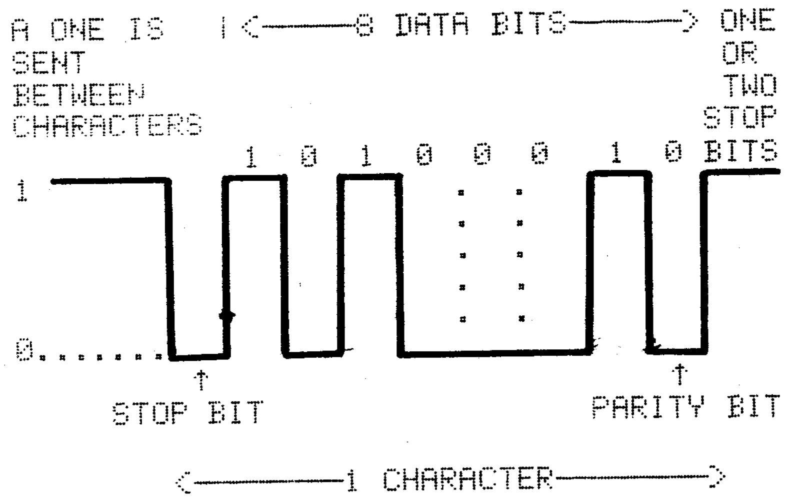
The transmission can be synchronous or asynchronous. Synchronous transmission requires that a clock signal be present (usually transmitted on a seperate line) to mark the start of each bit of information. Optionally, special data patterns are used to define the start of a message. Data must of course 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.



<u>FIGURE 2</u>

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 Electrical 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

<u>Manufacturers of PET compatible RS-232C Interface:</u>

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. M1W 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).

<u>Within BASIC</u>

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
```

In File Header

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

Applications

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.

```
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
```

B. Brown
Austin O'Brien School
6110 95th Ave.
Edmonton, Alta.
Feb. 4, 1980

Commodore Business Systems
3370 Pharmacy Ave.
Agincourt, Ont.

Editor Transactor:

I am enclosing some material in the hopes that it might be of some use to other PET users. First of all I have some outlines of routines for allowing reading and writing of files without having to go into the program to change file names in the OPEN statement every time the program is used. This is especially important if non-programmers are to use the disk system.

In the following program, a program file named by the user is to be LOADed. Generality is allowed for both file name and drive number.

```
100 INPUT "FILE NAME" ; F$
110 INPUT "ON DRIVE#" ; D$
120 Z$ = D$ + ":" + F$
130 LOAD Z$,8
140 END
```

The purpose of the following program is to allow the user to read in a data file from drive 1. It is interesting to note that CHR$(34) is needed so that the string explicitly includes quotes. (see note 1) I was very puzzled by this for a long time as the LOAD and OPEN commands both indicate the need for use of quotes in the same manner, yet the LOAD does not need explicit quotes whereas the OPEN does. The only other comment I want to make about this program is that flexibility in choice of drives is possible though it was not done here.

```
2500 INPUT "INPUT FILE NAME" ; F$
2510 Z$ = CHR$(34) + "1:" + F$ + ",SEQ,READ" + CHR$(34)
2520 OPEN 2,8,2,Z$
```

The last two program segments are used to read in a file and then rewrite it at a later date when some editing has been completed. The reason for this long drawn out procedure was simply that the @ does not seem to work for data files, however @ does work with the SAVE command. (see note 2)

Essentially what is done is to OPEN a scratch file for writing the edited data and then having done so, scratching the file F of lines 2500-2520, followed by a renaming of the scratchingfile to F$. I hope somebody has found an easier fix, however for now this has had to suffice.

```
5300 Z$ = CHR$(34) + "@1:TEMP,SEQ,WRITE" + CHR$(34)
5310 OPEN 2,8,2,Z$
5320 W (0) = M : W(1) = N
5330 FOR I = 0 TO N+1
   .
   .
   .
   .
5700 NEXT I : CLOSE 2
5710 OPEN 1,8,15
5720 Z$ = "S1:" + F$
5730 PRINT #1,Z$       :REM SCRATCH FILE
5740 Z$ = "R1:" + F$ + "=1:TEMP"
5750 PRINT #1,Z$       :REM RENAME TEMP FILE
```

Finally the last program can be used to renumber a program (see note 3) under one restriction that line numbers greater than 62999 are not allowed. Of course modification of line 63090 and renumbering of this program would allow for any upper limit you want within the limits allowed by BASIC. I intend to convert this into machine code when I have the time but I am hoping someone will beat me to it and publish their results. To use this program procede as follows:

1. clear the screen
2. LIST the renumbering program
3. LOAD the program to be renumbered
4. using the screen editor append the renumber program to your program
5. RUN 63000

```
63000 INPUT "[CS DN DN RV]INPUT STARTING LINE" ; LI
63010 INPUT "[DN DN RV] INPUT INCREMENT" ; IN
63020 PO = 1025
63030 PL = PEEK(PO) : PH = PEEK(PO+1)
63040 IF PL=0 AND PH=0 THEN END
63050 PO = PO + 2
63060 LH = INT(LI/256)
63070 LL = LI - LH * 256
63080 P = PO + 1 : A = PEEK(PO) : B = PEEK(P)
63090 IF B * 256 + A = 63000 THEN END
63100 POKE PO,LL : POKE P,LH : LI = LI + IN
63110 PO = PH * 256 + PL : GOTO 63030
63120 END
```

Editor's Notes:
1 Explicit quotes don't seem to be always necessary when passing strings to the disk. An interesting note nonetheless.
2 See Disk Notes page
3 This renumber will not renumber GOTOs and GOSUBs. One that will may be published in a later Transactor

Commodore Education Advisory Board

Commodore Business Machines is pleased to announce the formation of the CEAB. This board will provide suggestions for Commodore in promoting the use of microcomputers in education.

The board members are:

Mr. Wes Graham, University of Waterloo
Mr. Al Lott, University of Western Ontario
Mr. Don Whitewood, Toronto Board of Education
Mr. Frank Winter, Sheridan College

Commodore, upon advice of these gentlemen, will be preparing an education newsletter and will be promoting the sharing of software.

Educators are encouraged to send any software or articles to:

Commodore Business Machines
3370 Pharmacy Ave.
AGINCOURT, Ontario
M1W 2K4
(416) 499 4292

With your help this program can be a success!

Schools - The first CEAB software release is now available for copying at your local dealer.

Disk Notes

Four items to note when using disk:

1. Try not to use identical ID numbers. If a disk is inserted that has the same ID number as the disk there previously, it can be written on without an Initialize. That can be hazardous!

2. If a WRITE PROTECT ON error occurs, power down the disk and re-Initialize everything.

3. As a precaution, always Initialize both drives before doing a drive-to-drive Duplicate. If a raw disk is in the destination drive, New it with formatting first.

4. Avoid using the "@" symbol for write-and-replace. It has an intermittent tendency to throw a wrench into the system. Instead write the replacement file to a temporary file. Then Scratch the file to be replaced and Rename the temporary file to the name of the file just Scratched.

Try the following:

1. PET and 2040 on; Commodore diskette in Drive 0
2. Clear Screen
3. Type 5-10 spaces then: "*",8 (HOME)
4. Now hit a shifted RUN/STOP

Index Transactor #8