tiny basic

TWO TINY BASIC MODS

Michael E. Day
2590 DeBok Rd.
West Linn, Or 97068

Tom Pittman's TINY BASIC TB651K V.1K may have a bug!!!

The following program has the ability to lock you out of your computer:

```
1 RUN
```

What happens, is that when you type RUN, TINY begins execution, and the first statement it sees is RUN; which causes TINY to begin execution again. During all of this there is no test for a BREAK, which leaves the computer running away happily ignoring you.

This is no big deal, unless your computer happens to be located in a remote location (Like across town!), then it becomes a pain.

I found this bug late one night when nothing else was going right, (MY keyboard has not been the same since) and I typed it in by mistake.

Normally, I wouldn't care about it, but due to the circumstances it 'bugged' me, so I decided to do something about it. The following is the cure, and is located in the execute routine (XQ).

```
053F  A5 2A     LDA  2A     Get IL pointer (ADL)
0541  85 C4     STA  C4     Save it
0543  A5 2B     LDA  2B     Get IL pointer (ADH)
0545  85 C5     STA  C5     Save it
0547  4C 0F 05  JMP  050F   GOTO NX routine
054A  EA        NOP         Not used
054B  EA        NOP         Not used
```

This replaces the previous data, and allows a break test on execution.

The multiple statements per line modifications consists of changing the address of the Branch End routine to the new address, changing the name of the old NX IL code to NS (address remains the same), and the addition of the new NX IL code and address. NX retains the old meaning and description of Next Line. The new NS code searches for the Next Statement by looking for a colon (:) or carriage return, and passing control depending on what it has found.

The ML routine for the NS code is a modification of the old NX routine with a subroutine located at $0AE8. This routine causes execution of the next statement if a colon is found, it goes to the next line if a carriage return is found and in the run mode, otherwise it returns to the command mode.

The new ML routine for the BE code tests for a carriage return or colon to indicate statement end.

A modification to the IL is needed at $09B4 in order to use the colon (:) as a terminator, as this character is used to produce an X-OFF (DC3) after a print statement. This is modified to produce the X-OFF on an exclamation point (!) instead.

Another modification to the IL must be made at $09F5. This is required to make TINY begin execution on the next line rather than next statement following GOSUB RETURN. This is required due to the fact that TINY only remembers the line number for the return link, so if the GOSUB was not the first statement in the line, a hard loop would be set up. With this modification however, execution will begin on the next line, and not the next statement after a GOSUB has been executed.

A modification is made to the IL at $0A26 which causes execution to begin on the next line after a REM statement instead of beginning with the next statement. This allows colons to be in REM statements. It allows for more powerfull IF THEN statements. I.E.: IF A=0 THEN REM: LET A=1: PRINT A,: GOTO 20. In the above example if A is equal to 0, then execution begins on the next line, otherwise the rest of the present line is executed.

The colon may not be used in a print statement that is the second part of an IF THEN statement, since if the test is not true, then a search for the next statement is begun, and termination of the search will be prematurely done upon deteciton of the colon in the print statement. The colon may be in any other print statement however, even on the same line as the IF THEN statement. It just can not be used as the second part of an IF THEN statement.

The GOSUB will always be the last statement executed in a line. I.E.:
IF A=0 THEN GOSUB 20: LET A=1: PRINT A: GOTO 10
In the above example if A is equal to 0, then the GOSUB 20 is executed, and execution continues with the next line following the example upon RETURN from the GOSUB. If A is not equal to 0, then the GOSUB is skipped, and the rest of the line is executed.

IL ADDRESS CHANGES

```
CHANGE TO      WAS

022C  F2       FD
022D  0A       03  Branch End (BE)
025A  E0       9F
025B  0A       05  Next Line (NX)
```

Old IL code NX now becomes NX (Next Statement) there is no address change however.

IL ROUTINE CHANGES

```
09B4  83  A1   !          X-OFF On (!) exclamation
          (3)  BC 09B8    point instead of (:) colon
09F5  1E       NX         NX on Return instead of NS
```

ML ROUTINE ADDITIONS

NEW NX ROUTINE

```
0AE0  20 14 04  JSR 0414  Search for "CR"
0AE3  D0 FB     BNE 0AE0  Con't until found
0AE5  4C 0B 05  JMP 050B  Get new line
```

NEW NS ROUTINE

```
0AE8  20 14 04  JSR 0414  Search for terminator
0AEB  F0 04     BEQ 0AF1  Return if "CR"
0AED  C9 3A     CMP #3A   Return if ":"
0AEF  D0 F7     BNE 0AE8  Otherwise try again
0AF1  60        RTS
```

NEW BE ROUTINE

```
0AF2  20 25 04  JSR 0425  Read BASIC character
0AF5  C9 0D     CMP #0D   If it is a "CR"
0AF7  F0 F8     BEQ 0AF1  Return
0AF9  C9 3A     CMP #3A   or a ":"
0AFB  F0 F4     BEQ 0AF1  Return
0AFD  4C 64 03  JMP 0364  Otherwise go branch
```

ML ROUTINE CHANGES

NS ROUTINE

```
0506  20 E8 0A  JSR 0AE8  Find terminator
0509  B0 0C     BCS 0517  End line?
050B  A5 BE     LDA BE
050D  F0 23     BEQ 0532  Run mode?
```

RAMBLINGS ABOUT PITTMAN TINY BASIC by

Lew Edwards

Bought Tom Pittman's TINY BASIC, also his "Experimenter's Kit". Perhaps you might be interested in the following comments.

Things "not in the book" or at least not too clear.

Saving and loading basic programs using KIM cassette routines---Use the values in $0020 & $0021 for SAL & SAH and use the values in $0024 & $0025 for EAL & EAH when dumping to cassette. When loading the saved programs, transfer the values in $17ED & $17EE to $0024 & $0025 and enter TINY via the "warm start". Of course before loading the tape, you should have previously done a "cold start" to initialize the basic pointers, etc. Expect your whole system to crash if you try to make program changes without setting 24 & 25 to the correct values. You can append a second program to the one in memory if the second program has line numbers higher than the first. I have written a line renumbering program if anyone is interested. The second program is loaded in starting at the address in $0024 & $0025 minus four. Again, transfer values from $17ED & EE to $0024 & 25. I am using a tape loading subroutine callable as a USER function, which directly uses 24 & 25 as a pointer for storing recovered data so that it is automatically set up as end pointer for user programs.

HOW TINY STORES PROGRAMS:

User programs start at the address stored in $0020 & $0021 and lines are stored exactly as entered from the keyboard. The line number is stored as two hex bytes, all the rest as ASCII, ending with the carriage return, OD(hex). All lines are stored in sequence as numbered, with TINY doing the editing as each line is entered (or deleted, or replaced). TINY stores a ZERO line number in the two bytes follwoing the CR in the last line of the program. When TINY responds to a CLEAR command, it puts the zero line number in the first two bytes of the user program space and initializes the pointers. If you should accidently clear, say be using the "cold" start to re-enter basic, after having entered a program; you can salvage the program by loading a value in the first byte of user memory equal (in hex) to the original line number of the first line. Of course, if the number is over 255, you'll have to put the high order value into the second byte. This will let you list and run the program, but if you want to make any changes, you'd better restore the pointer at 24 & 25. You can search through memory to find the right address using the following rules. First, line numbers are contained in the two bytes immediately following a carriage return (ODhex). The last CR is followed by two zero value bytes. Add 5 to the address of the last CR and load the result into 24 & 25.

MACHINE LANGUAGE SUBROUTINES:

These can be used by calling a USER functions. If you want an ML subroutine to be included with your TB program, it can be "contained" within REM statements placed after the last line of your program. Make one or more REM statements using enough characters between the first REM and the last CR to accomodate your subroutine. The result will be garbage on a LIST, but that's immaterial. The ML subroutine can then be called by: X=USR(USR(S+20, 36)+USR(S+20,37)*256-n) where X is the result returned from the subroutine in the A & Y registers, S is the starting address of TINY BASIC, and n is the number of bytes reserved for the machine language code +6. If the ML subroutine is to be called more than once, a variable may be set to the value within the opening and closing parentheses. Second and third arguments may be included to pass parameters. The line renumber program I wrote in TB uses this technique to locate the line numbers. I had at first written it using only the TB built in USER routines for "peek" and "poke", but it ran too slowly to suit me. No, the renumber program does not renumber the goto's and the gosub's.

****

tiny basic

Oops! In issue #13, I left out the mod that must be made to the IL at $0A26. Here it is:

0A26 1E NX NX on REM instead of NS.

In the next issue, we'll be presenting a very comprehensive string capability for TB as well as a cassette save and load ability. (I ran out of room in this issue). Must be a good number of Tiny Basic users out there. Have you done anything neat with TB? Let us know.

tiny basic

TINY BASIC CASSETTE SAVE & LOAD

by William C. Clements, Jr.
Univ. of Alabama
Chem & Metal Eng.
Box 2662
University, Al 35486

I recently bought TINY BASIC and the accompanying experimenter's kit, and have enjoyed finding out how the BASIC statements are broken down and implemented. With a little study one can easily pick up the pseudolanguage used to program the inner interpreter, and then all sorts of possibilities exist for custom modifications to suit one's whim. I noticed the comments about transferring BASIC statements to and from cassette tape in Issue 13 (Lew Edwards, p. 14), and thought perhaps your readers might be interested in how I added the SAVE and LOAD commands to my version of TINY BASIC for the KIM-1. With my implementation, TINY can use the existing KIM monitor routines (or any others if one wishes) to save and load programs, and transfer of starting and ending addresses, etc. is handled by a machine language routine. The cassette file number is specified in the added BASIC commands: SAVE X or LOAD X, where X is any integer 0 _ X _ 255 corresponding to KIM file I.D.'s 00 through FF. My version of TINY is the one having the cold start at 2000 hex; corresponding address offsets can be added for other versions.

The patch to the Intermediate Interpreter is made at relative location 00B7, as shown on p.38 of the Experimenter's Manual. This is address 2827 absolute. The patch is as follows:

```
                  ;test for keyword SAVE
00B7 8B534156C5 TAPE BC LOAD "SAVE"
                  ;push start address of
00BC 09 29           LB 29
                  ;save routine onto stack
00BE 09 0E           LB 0E
                  ;do it again
00C0 0B               Q DS
                  ;error stop if file id not number
00C1 C0              BN
                  ;go to save routine at 290E_H
00C2 2E              US
                  ;test for keyword LOAD
00C3 8A4C4F41C4 LOAD BC DFLT "LOAD"
                  ;push start address of
00C8 09 29           LB 29
                  ;load routine onto stack
00CA 09 28           LB 28
                  ;go to load routine at
00CC 38 C0            J Q
                  ;2928_H via above instructions
00CE A0          DFLT BV *
  '                 '     ' (continue with
  '                 '     ' remaining IL code)
```

The constants after the LB commands specify the hex addresses of the machine language routines which handle the SAVE X and LOAD X functions. The line labeled DFLT is thus moved from relative location 00B7 to 00CE, resulting in an offset of 17_H or 23_D for remaining lines. This must be accommodated in the jump and jump subroutine commands in the I.L. The changes in destination for those instructions which jump beyond the patch are listed. All error messages originating beyond the patch will also be increased by 23_D.

My version jumps to a pair of machine language routines which initialize the file i.d., SAL, SAH, and the TINY BASIC registers. BASIC files are saved using a Hypertape routine stored in EPROM at location $C400_H$; if the user wishes to use the KIM tape dump routine, he should change the contents of location 2927_H to 18_H. Appropriate routines can of course be relocated anywhere the user wishes, so long as the correct entry points are provided for in the I.L. patch. After execution of a SAVE or LOAD, TINY must be manually re-entered at the warm start (the limits of memory for the BASIC statements are set for my system when BASIC is first entered). A jump to warm start could of course be placed at the end of the tape dump and load routines if ones stored in RAM instead of ROM were being used.

These alterations were worth their trouble in added convenience: SAVE 01 is a lot easier than exiting TINY, storing 01 in 17F9, and looking up the memory bounds for the BASIC statements to set SAL and SAH manually. I hope this modification will be of interest to other users of TINY BASIC.

MACHINE LANGUAGE ROUTINES USED BY THE PATCH

```
2906  8D F9 17  00     STA 17F9_H   STEPS COMMON TO
      A9 00            LDA $00      BOTH
      85 F1            STA 00F1_H   ROUTINES
      60               RTS

290E  20 06 29  SAVE   JSR 00       FILE SAVE ROUTINE
      A5 20            LDA 0020_H
      8D F5 17         STA 17F5_H
      A5 21            LDA 0021_H
      8D F6 17         STA 17F6_H   INITIALIZATION
      A5 24            LDA 0024_H
      8D F7 17         STA 17F7_H
      A5 25            LDA 0025_H
      8D F8 17         STA 17F8_H
      4C 00 C4         JMP HYPERTAPE
2928  20 06 29  LOAD   JSR QQ set 17F9_H, 00F1_H
      4C 73 18         JMP TPLOAD read tape
292E  AD ED 17  ENTER  LDA EAL set address
      85 24            STA 0024_H at end
      AD EE 17         LDA EA_H  of BASIC
      85 25            STA 0075_H program file
      4C 03 20         JMP BASIC go to warm start
```

Restart BASIC at ENTER (loc. $292E_H$) after loading. Restart at warm start (2003_H in my version) after saving.

Summary of additional modifications to I.L. Code (new transfer statement destination caused by insertion of patch)

Relative Location (See pp. 36-40 TINY BASIC Experimenter's Manual)	New Instruction
0014	30 D3
001F	30 D3
0029	30 D3
004B	30 D3
0052	30 D3
0054	31 4B
0056	30 D3
0073	30 D3
009E	30 D3
00BE	30 EA
00C4	30 EA
00C8	30 EA
00CE	30 EA
00D3	30 F9
00D7	30 F9
00F7	31 47
0114	30 D3
0116	31 41
0118	31 41
0125	30 D3
012C	38 D3

TINY BASIC STRINGS

by Michael E Day
2590 DeBok Rd
West Linn, Or 97068

Here is the string mod I've been using which I access thru the USR verb. This requires 512 bytes of memory, and is relocatable and will run out of ROM or protected memory except for the storage area which operates out of RAM, however it can be located in any 256 byte block of free memory.

PEEK $ USR(2816,ADDRESS)
PEEK at string at the string relative address ADDRESS. Returns decimal value of addressed byte.

POKE $ USR(2822,ADDRESS,DATA)
POKE data byte DATA into the string relative address ADDRESS. Returns string relative address plus one.

INPUT SP$ USR(2832,BEGIN,END)
INPUT a string of characters beginning with string relative address BEGIN, echoing back a space with each input character, until a carriage return is encountered, or the ending address END is reached. Returns the string relative ending address plus one.

INPUT $ USR(2839,BEGIN,END)
INPUT a string of characters as in INPUT SP$, but without the space echo. Returns the string relative ending address plus one.

PRINT SP$ USR(2905,BEGIN,END)
PRINT the character string beginning with the string relative address BEGIN, and print a space after each character, until a carriage return is encountered, or the ending address END is reached. Returns the string relative ending address plus one.

PRINT $ USR(2912,BEGIN,END)
PRINT the character string as in PRINT SP$, but without the space echo. Returns the string relative ending address plus one.

SEARCH $ USR(2946,BEGIN,DATA)
SEARCHes for the BCD equivalent of decimal value DATA, beginning at string relative address BEGIN, until a match is found, or the ending address of variable "L" is reached. Returns the string relative ending address plus one.
If a match is not found the return address will be 0 (zero). Variable "L" is decremented once per test until match is found, or it is 0.

MOVE $ USR(2966,FROM,TO) (Length in variable "L")

MOVEs a group of characters of the length in variable "L" beginning at the relative string address FROM, and moving them to relative string address TO, for the length of variable "L". Returns the FROM ending address plus one. Variable "L" is zeroed. (Lower 8 bits only, see notes on addressing of strings).

SET POINTERS

These are memory formating routines that are addressed by the other routines, and are listed with USR statements only for reference. They do not need to be accessed by TINY.

OPERATIONAL NOTES

Addressing is limited to 0-256 (8 bit addressing) and the upper bits are ignored (I.E. 512 will appear as a 0, and 513 will appear as a 1).

The string array table is perminently fixed to 256 bytes in length, and dedicated for this purpose. This table may be located anyplace in RAM so long as intrusion from other sources is not allowed. Relocation is done by changing the page location address at 0BAA (0BAA A0 0C LDY #0C). The routines that access the table are clean. (They are relocatable, and will operate out of ROM or protected memory.)

All data passed through the USR statements both to and from is in decimal. The data inside the routines however, remain in BCD.

In the PRINT and INPUT routines, if the BEGIN address is less than the END address, an error exit will occur which causes the exit address to be 0, and the funciton asked for is not performed.

If only one address is given, the second address will be assumed to be equal to the first address given (I.E. USR(2912 0) will print out a single character at location 0 and return an address value of 1 to TINY.

As with any USR statement in TINY, the address and data information passed through the USR statement can be calculated from any expression. (Such as USR(2912,B,E-2) can be used to print a string starting at the address in variable "B", and using the E-2 to suppress the ending carriage return, and another variable can be used to pickup the returning ending address.)

The routines given have been located at the end of TINY, as this allows for easy isolation from TINY by revising the user memory starting address located at 028B.

```
028B   A9 0B     LDA  #0B  Old starting address
028B   A9 0D     LDA  #0D  New starting address
```

This is the only place that TINY references this, so it is the only thing that needs to be changed. NOTE: A <u>cold</u> start <u>MUST</u> be done after this change to <u>set the pointers</u>, or else they will have to be set by hand.

The entire string mod requires less than 512 bytes of memory (256 bytes for the array, and 187 bytes for the routines.)

A possible mod would be to place the array page address in zero page memory, and modify it with TINY before going into the routines. This would allow for greater than 256 bytes, but program management must be closely followed, or strange things might happen!!!

The cancel code used in TINY will terminate an INPUT $ without putting the character into the array, therefore this code can not be used directly. All previous characters will have been inserted however.

PEEK $ USR(2816,ADDRESS)

```
0B00   20 A8 0B   JSR   0BA8      Set pointers A
0B03   B1 18      LDA   (18),Y    Pick up data
0B05   60         RTS             Return to TINY
```

POKE $ USR(2822,ADDRESS,DATA)

```
0B06   20 A8 0B   JSR   0BA8      Set pointers A
0B09   91 18      STA   (18),Y    Store data
0B0B   E6 18      INC   18        Increment pointer
0B0D   A5 18      LDA   18        Return address to TINY
0B0F   60         RTS             Return to TINY
```

INPUT SP$ USR(2832,BEGIN,END)

```
0B10   20 B5 0B   JSR   0BB5      Set pointers B
0B13   84 1B      STY   1B        Clear 1B
0B15   B0 03      BCS   0B1A      Goto Input routine
```

INPUT $ USR(2839,BEGIN,END)

```
0B17   20 B5 0B   JSR   0BB5      Set pointers B
0B1A   A9 3F      LDA   #3F
0B1C   20 09 02   JSR   0209      Print a "?"
0B1F   A9 20      LDA   #20
0B21   20 09 02   JSR   0209      Print a "SP"
0B24   20 06 02   JSR   0206      Get a character
0B27   CD 10 02   CMP   0210      Is it "ESC"?
0B2A   F0 28      BEQ   0B54      If so return to TINY
0B2C   CD 0F 02   CMP   020F      Is it "BS"?
0B2F   D0 11      BNE   0B42      If so back up
0B31   A5 1A      LDA   1A
0B33   C5 18      CMP   18        Is it begin of array?
0B35   F0 E8      BEQ   0B1F      If so restart
0B37   C6 18      DEC   18        Decrement pointer
0B39   A5 1B      LDA   1B        Input SP$ ?
0B3B   D0 E7      BNE   0B24      If not get next
                                    character
0B3D   AD 0F 02   LDA   020F      Get "BS"
0B40   90 DF      BCC   0B21      Print it
0B42   91 18      STA   (18),Y    Store data
```

INPUT $ USR(2839,BEGIN,END) Con't.

```
0B44   E4 18      CPX   18        Is it end of array?
0B46   F0 0C      BEQ   0B54      If so return to TINY
0B48   E6 18      INC   18        Increment pointer
0B4A   C9 0D      CMP   #0D       Is it a "CR"
0B4C   F0 08      BEQ   0B56      If so return to TINY
0B4E   A5 1B      LDA   1B        Print a "SP"?
0B50   D0 D2      BNE   0B24      If not get next byte
0B52   F0 CB      BEQ   0B1F      Print a "SP"
0B54   E6 18      INC   18        Increment pointer
0B56   A5 18      LDA   18        Return exit address
                                    to TINY
0B58   60         RTS             Return to TINY
```

PRINT SP$ USR(2905,BEGIN,END)

```
0B59   20 B5 0B   JSR   0BB5      Set pointers B
0B5C   84 1B      STY   1B        Clear 1B
0B5E   B0 03      BCS   0B63      Goto print routine
```

PRINT $ USR(2912,BEGIN,END)

```
0B60   20 B5 0B   JSR   0B5B      Set pointers B
0B63   B1 18      LDA   (18),Y    Pick up data
0B65   20 09 02   JSR   0209      Print character
0B68   E4 18      CPX   18        Is it end of array?
0B6A   F0 11      BEQ   0B7D      If end return to TINY
0B6C   E6 18      INC   18        Increment pointer
0B6E   C9 0D      CMP   #0D       Is it a "CR"?
0B70   F0 0D      BEQ   0B7F      If so return to TINY
0B72   A5 1B      LDA   1B        Print a "SP"?
0B74   D0 ED      BNE   0B63      If not get next byte
0B76   A9 20      LDA   #20
0B78   20 09 02   JSR   0209      Print a "SP"
0B7B   D0 E6      BNE   0B63      Go get next byte
0B7D   E6 18      INC   18        Increment pointer
0B7F   A5 18      LDA   18        Get exit address
0B81   60         RTS             Return to TINY
```

SEARCH $ USR(2946,BEGIN,DATA) (Length in variable "L")

```
0B82   02 A8 0B   JSR   0BA8      Set pointers A
0B85   B1 18      LDA   (18),Y    Pick up test byte
0B87   E6 18      INC   18        Increment pointer
0B89   C5 1A      CMP   1A        Found match?
0B8B   F0 06      BEQ   0B93      If so return to TINY
0B8D   C6 98      DEC   98        Decrement variable'L'
0B8F   D0 F4      BNE   0B85      If not get next byte
0B91   84 18      STY   18        Clear 18 (pointer)
0B93   A5 18      LDA   18        Return exit address
                                    to TINY
0B95   60         RTS             Return to TINY
```

MOVE $ USR(2966,FROM,TO) (Length in variable "L")

```
0B96   20 A8 0B   JSR   0BA8      Set pointers A
0B99   B1 18      LDA   (18),Y    Pick up byte
0B9B   91 1A      STA   (1A),Y    Store it
0B9D   E6 18      INC   18
0B9F   E6 1A      INC   1A        Increment pointers
0BA1   C6 98      DEC   98        Decrement variable'L'
```

```
0BA3  D0 F4      BNE  0B99    If end return to TINY
0BA5  A5 18      LDA  18      Return exit address
                                to TINY
0BA7  60         RTS          Return to TINY

SET POINTERS A   USR(2984,Y,A)
0BA8  84 18      STY  18      Save begin
0BAA  A0 0C      LDY  #0C     Set array page
0BAC  84 19      STY  19      Store array page
0BAE  84 1B      STY  1B      Store array page
0BB0  A0 00      LDY  #00     Clear Y
0BB2  85 1A      STA  1A      Save A
0BB4  60         RTS          Exit
```

```
SET POINTERS B   USR(2997,Y,A)
0BB5  20 A8 0B   JSR  0BA8    Set pointers A
0BB8  AA         TAX          Save end
0BB9  A5 18      LDA  18      Recapture begin
0BBB  85 1A      STA  1A      Save it
0BBD  E4 18      CPX  18      Bad address?
0BBF  B0 03      BCS  0BC4    If so go error
0BC1  68         PLA
0BC2  68         PLA          Discard string link
0BC3  98         TYA          Clear A
0BC4  60         RTS          Exit

READ KEY    USR(3064)
0BF8  AD 00 C0   LDA  0C00    Pick up data
0BFB  29 7F      AND  #7F     Clear bit 8 (Strobe)
0BFD  A0 00      LDY  #00     Clear Y
0BFF  60         RTS          Return to TINY
```

tiny basic

TINY BASIC   Editors note

Several of you were apparently confused as to how to add the Tiny Basic mods from #15 to your systems. I wholeheartedly recommend you pick up the Tiny Basic Experimenters Kit mentioned in one of the articles. (It's available for $15 from 6502 Program Exchange, 2920 Moana Ln, Reno NV 859C9.

MICHAEL DAY

TINY BASIC PAGE 0 MEMORY MAP
for TOM PITTMAN's TINY BASIC TB651K V.1K

```
0000 - 000F   UNUSED
0010 - 001F   USED IN PROTO VERSIONS ONLY
0020 - 0021   USER SPACE LOW ADDRESS
0022 - 0023   USER SPACE HIGH ADDRESS
0024 - 0025   PROGRAM END + STACK RESERVE
0026 - 0027   TOP OF GOSUB STACK
0028 - 0029   CURRENT BASIC LINE #
002A - 002B   IL PROGRAM COUNTER
002C - 002D   BASIC POINTER
002E - 002F   SAVED POINTER
0030 - 007F   INPUT BUFFER AND COMPUTATION STACK
0080 - 0081   RANDOM NUMBER SEED
0082 - 0083   VARIABLE 'A'
0084 - 0085   VARIABLE 'B'
....          ....
00B4 - 00B5   VARIABLE 'Z'
00B6 - 00B7   TRANSFER WORK POINTER
00B8 - 00B9   MISC WORK REGISTER
00BA - 00BB   MISC WORK REGISTER
00BC - 00BD   TEMPORARY STORAGE REGISTER
00BE          RUN MODE FLAG
00BF          PRINT CONTROL
00C0          INPUT BUFFER POINTER
00C1          COMPUTATION STACK POINTER
00C2          2nd ½ OF STACK POINTER (ALWAYS 00)
00C3          COUNTER (USED IN PN ONLY)
00C4 - 00C5   IL XQ POINTER
00C6 - 00C7   GOSUB STACK WORK POINTER
00C8 - 00D7   USED IN SPHERE VERSIONS ONLY
00D8 - 00FF   UNUSED
```

There are the major use of these registers only they may be used for other purposes on an availability basis.

TVT-6/TINY BASIC INTERFACE

by Michael Allen
6025 Kimbark
Chicago IL 60637

I had a lot of trouble getting Tom Pittman's Tiny Basic to work with the KIM-1/TVT-6 combination. Now, looking back, the input and output routines included below seem fairly simple and straight-forward. So I thought I should share these with you to help those who may be making the same mistakes I was.

The T. B. version I have resides in memory locations 0200 to 0AC6. You must change six bytes within T.B. as follows;

1. Set 0207 to C7 and 0208 to 0A. This is a jump to a subroutine to input a character. The input routine saves the return address to T.B. then jumps to the SCAN program and stays there until interrupted by a strobe signal from a key being pressed on the keyboard. If the IRQ vector has been properly set to 0AD3, a character is sent to the cursor subroutine. Then a return is made to T.B. Note that a CLI (clear interrupt status) instruction was inserted in SCAN (underlined in the hex dump).

2. Set 020A to F3 and 020B to 0A. This is a jump to the output subroutine where the miscellaneous characters T.B. sends for the benefit of a teletype are trapped before falling through to the cursor subroutine.

3. Set 020F to 08. This allows T.B. to recognize the ASCII backspace.

4. Set 028C to 0E. When starting T.B. at 0200 (cold start), this byte determines how T.B. defines the lowest address of program space.

5. Also be sure to set 17FE to D3 and 17FF to 0A.

I relocated SCAN to be able to reload T.B. from tape in one load. The version of SCAN shown is from Don Lancaster's Popular Electronics article except for bytes 0BA4 and 0BCC which were changed in order to display pages 0C00 and 0D00.

The Cursor program is adapted from Don's but is much shorter as it only supports backspace and carriage return controls--all you really need with T.B. (also INPUT sets lowercase to uppercase so you don't have to shift back and forth.)

KIM's Memory map now appears thus:

0020-00B9	Used by tiny BASIC
00E8-00EE	Used by I/O routines
0200-0AC6	Tiny BASIC
0AC7-0B79	INPUT & OUTPUT Subroutines
0B7A-0BDC	SCAN
0BDD-0BFF	34 bytes for USR subroutines (I put Don Box's subscripted variable SBR's here; see KUN #5.)
0C00-0DFF	TVT-6 display area
0E00-13FF	1.5K program area

TVT6/TINY BASIC INTERFACE LISTING

SET 17FE = D3, 17FF = OA

```
OAC7 68           INPUT   PLA LOW     SAVE ...
OAC8 85 E8                STA TEMP          RETURN ...
OACA 68                   PLA HI                      ADDRESS.
OACB 85 E9                STA TEMP+1
OACD BA                   TSX         AND STACK ...
OACE 86 EA                STX TEMP+2              POINTER.
OADO 4C A7 OB             JMP SCAN

OAD3 AD 00 17     BREAK   LDA CHAR    GET CHARACTER.
OAD6 29 7F                AND #$7F    REMOVE PARITY.
OAD8 C9 61                CMP #$61    LOWER CASE LETTER?
OADA 90 02                BCC SKIP    NO; SKIP AHEAD.
OADC E9 20                SBC #$20    YES; MAKE UPPER CASE.
OADE 85 EB        SKIP    STA TEMP+3 SAVE CHARACTER.
OAEO C9 OD                CMP #$OD    CARRAGE RETURN?
OAE2 FO 03                BEQ RTN1    YES; RETURN.
OAE4 20 FB OA             JSR CURSOR NO; ENTER CHARACTER.
OAE7 A6 EA        RTN1    LDX TEMP=3 RESTORE ...
OAE9 9A                   TXS               STACK POINTER.
OAEA A5 E9                LDA TEMP+1  RESTORE ...
OAEC 48                   PHA                   RETURN ...
OAED A5 E8                LDA TEMP                        ADDRESS.
OAEF 48                   PHA
OAFO A5 EB                LDA TEMP+3   GET CHARACTER..
OAF2 60           RTN2    RTS          RETURN TO TINY.

OAF3 C9 OB        OUTPUT  CMP #$OB     TRAP ...
OAF5 30 FB                BMI RTN2           CONTROL ...
OAF7 C9 7F                CMP #$7F                   CHARACTERS.
OAF9 BO F7                BCS RTN2
OAFB 48           CURSOR  PHA          SAVE CHARACTER.
OAFC AO OO                LDY #O       RESET INDEX.
OAFE A5 EE                LDA EE       GET CURSOR HI ADDR.
OBOO C9 OD                CMP #$OD     IS CURSOR ON PAGE OD?
OBO2 FO 04                BEQ CONT     YES; CONTINUE.
OBO4 C9 OC                CMP #$OC     NO; OR ON PAGE OC?
OBO6 DO 2F                BNE SCROLL   NO; INITIALIZE CURSOR.
OBO8 B1 ED        CONT    LDA (ED),Y   GET OLD CHARACTER.
OBOA 29 7F                AND #$7F     REMOVE CURSOR.
OBOC 91 ED                STA (ED),Y   REPLACE.
OBOE 68                   PLA          RECALL NEW CHARACTER.
OBOF C9 20                CMP #$20     IS IT A CHARACTER?
OB11 BO 59                BCS ENTER    YES; ENTER IT.
OB13 C9 OD                CMP #$OD     CARRAGE RETURN?
OB15 DO OC                BNE SKIP1    NO; SKIP
OB17 A5 ED                LDA ED       YES; MOVE CURSOR ...
OB19 09 1F                ORA #$1F           TO RIGHT SIDE.
OB1B 85 ED                STA ED       AND REPLACE.
OB1D 20 61 OB             JSR INCR     INCREMENT CURSOR.
OB20 4C 6F OB             JMP END
```

```
0B23 C9 08      SKIP1   CMP #$08       BACKSPACE?
0B25 D0 4C              BNE RESTORE    NO; CONTINUE.
0B27 C6 ED              DEC ED         YES; DECREMENT CURSOR.
0B29 A9 FF              LDA #$FF       TEST FOR PAGE ...
0B2B C5 ED              CMP ED                   UNDERFLOW.
0B2D D0 44              BNE RESTORE    O.K. TO CONTINUE.
0B2F C6 EE              DEC EE         DECREMENT PAGE.
0B31 A9 0B              LDA #$0B       TEST FOR SCREEN ...
0B33 C5 EE              CMP EE                   UNDERFLOW.
0B35 D0 3C              BNE RESTORE    O.K.
0B37 A9 00      SCROLL  LDA #0         NOT O.K.; HOME CURSOR.
0B39 85 ED              STA ED         TO 0C00
0B3B A9 0C              LDA #$0C       (UPPER LEFT OF SCREEN)
0B3D 85 EE              STA EE
0B3F A0 20      LOOP    LDY #$20       ADD OFFSET TO INDEX.
0B41 B1 ED              LDA (ED),Y     MOVE ...
0B43 A0 00              LDY #0            CHARACTER ...
0B45 20 5F 0B           JSR STORE                    UP.
0B48 D0 F5              BNE LOOP       LOOP UNTIL END OF SCREEN.
0B4A 18                 CLC            CLEAR FLAG.
0B4B A9 E0      HOME    LDA #$E0       HOME CURSOR
0B4D 85 ED              STA ED          TO 0DE0
0B4F A9 0D              LDA #$0D       (LOWER LEFT OF SCREEN).
0B51 85 EE              STA EE
0B53 B0 1E              BCS RESTORE    FINISH IF FLAG SET.
0B55 A9 20      SPACE   LDA #$20       ELSE; CLEAR LAST LINE.
0B57 20 5F 0B           JSR STORE      ENTER SPACT TO ...
0B5A D0 F9              BNE SPACE        END OF LINE.
0B5C 38                 SEC            SET FLAG.
0B5D B0 EC              BCS HOME       TRY AGAIN.

0B5F 91 ED      STORE   STA (ED),Y     ENTER CHARACTER.
0B61 E6 ED      INCR    INC ED         INCREMENT CURSOR.
0B63 D0 06              BNE RTN        OVERFLOW?
0B65 E6 EE              INC EE         YES; INCR CURSOR TO NEXT PAGE.
0B67 A9 0E              LDA #$0E       TEST FOR SCREEN OVERFLOW.
0B69 C5 EE              CMP EE
0B6B 60         RTN     RTS

0B6C 20 5F 0B   ENTER   JSR STORE      ENTER CHARACTER.
0B6F D0 02      END     BNE RESTORE    END OF SCREEN?
0B71 F0 C4              BEQ SCROLL     YES; SCROLL UP.
0B73 B1 ED      RESTORE LDA (ED),Y     GET CHARACTER.
0B75 09 08              ORA #$80       ADD CURSOR.
0B77 91 ED              STA (ED),Y     REPLACE.
0B79 60                 RTS            RETURN TO I/O ROUTINES.
```

HEX DUMP OF RELOCATED "SCAN" PROGRAM:

```
0B7A EA 8D 84 0B 48 68
0B80 D0 00 20 00 80 69 08 C9 C0 90 F0 20 DA 0B 20 00
0B90 80 AA AD 83 0B 69 1F 8D 83 0B 8A D0 AA EA 69 C0
0BA0 20 00 80 C9 86 90 D3 AD D9 0B 49 80 30 05 8D D9
0BB0 4B A2 66 20 DA 0B 20 DA 0B 10 05 8D D9 4B A2 67
0BC0 20 1E 80 58 48 68 A9 00 8D 83 0B A9 84 8D 84 0B
0BD0 20 00 80 18 CA 30 A4 10 ED 80 B0 00 60
```

tiny basic

Ben Doutre
621 Doyle Rd
Mont St-Hilaire Que
Canada J3H 1M3

Dear Eric,

First, let me say that 6502 User Notes is top quality and getting better with each issue. Keep up the good work.

I have been following the Tiny Basic items with particular interest and feel that Michael Day, Lew Edwards and William Clements are to be congradulated for their contributions in issues #13-15. The following comments may be of interest:

a) In Day's string mods, KIM owners who are using the TTY I/O routines GETCH and OUTCH will have problems, since these do not save the Y register. Rather than reassemble the code, you can set up a couple of buffer I/O routines as follows:

```
INPUT   JSR  GETCH        OUTPUT  JSR  OUTCH
        INY                       INY
        RTS                       RTS
```

and change your JMP vectors at $0206 and $0209 to wherever you tuck these routines in. There is also a pretty obvious typo at 0B82: 02 should be 20. These string features are really interesting to play with. (The BNE instruction at $0B7B in Tiny B must be changed to BEQ for this mod to work).

b) In Clements tape SAVE and LOAD mod, one item was omitted from the list of revised branches: at IL relative address 00DD, the "30E2" should be changed to "30F9". This mod also works great, although perosnally, I have reservations about adding IL workload (I seldom use "Let" expressions) for non-run-time extensions and prefer to use an input trap routine. But that is another story.

I have developed a small (74 bytes) utility program which makes it pretty easy and straight-forward to load machine-code routines. If you feel that your readers would be interested, the enclosed listing and example of use will make most of it clear, together with these additional comments.

My system is a KIM-1 with an additional 8K bytes of RAM, located at $2000 to $3FFF. My version of Tiny Basic is TB651T, V.1T, which loads at $2000 and extends to $28C6. Day's multiple statement per line mods are tucked into the remaining $2800 space, and the next 1K is allocated to utilities, like tape I/O (I use Lew Edwards' ZIPTAPE, the greatest thing to come along since sliced bread!), Selectric print routines, etc. User space is allocated starting at $2D00, but this can vary.

EZLOAD is an interface routine which scans the output stream looking for a unique prefix character. When it finds it, it then proceeds to convert each following pair of characters into a hex byte which is placed at the top (bottom?) of the Basic stack. Anyway, the bytes are shuffled along the stack, with the Basic stack pointer and variable "A" (an arbitrary choice) keeping up with the head of the code. The loading stops when a carriage return comes along, but may resume and stop several times. When the dust finally settles, the machine code is neatly arranged in execution order at the top of user space, with not a byte wasted, and with "A" all set to be used as the first parameter in a USR function call.

The machine code is written into REM statements, and will print in readable form when listed. It is, in fact, loaded by being LISTed, and is effectively wiped out by a warm start (the Basic stack pointer is reset) or by the execution of an END statement, which ends up doing a warm start for you. The best way to use a program with EZLOAD machine code is to do a command-mode END, list the program, then RUN it.

The code will not load when you are first typing it in, unless you have an I/O setup with external echo. You may be tempted to use the selected prefix character in a run-time PRINT "..." but this will clobber your stack when it is in use for other things. With some slight changes, though, this presents some intriguing possibilities. Obviously, the programs may be saved on tape, and later loaded with their machine-code still intact and usable. This is a considerable benefit.

EZLOAD was written with severe space constraints, consequently some niceties were left out, such as checking for stack over flow. In particular, it will not work as is unless some modifications are made to Tiny's memory grab code in the cold start areas. These are detailed below. Users with more bytes available might want to check for valid HEX code characters (KIM's PACKT will return with Zero bit set if valid, reset otherwise, assuming you enter with Y equal 0) and use the validity check to step over spaces and other readability aids. You could also use several of Tiny's variables to point to various code segments, or several different prefixes, etc etc.

The trouble with the cold start code, insofar as this program is concerned, is that it runs the top-of-user-space pointer ($0022-23) to the last real RAM location plus one. That plus one I didn't need! And contrary to what the Experimenter's Kit seems to say (top of page 6), the Basic stack pointer must be decremented before use, not after; these conditions presented severe problems in initializing EZLOAD, beyond resetting the load flag which is done by the first carriage return from a warm start. So that cute memory grab finally had to go!

In my version of TB, the cold start vector jump at $2000 points to $2085. The code from $2085 thru $20A9 initializes both the start and end of user space pointers ($0020-21 and $0022-23, respectively). The following code was substituted: (You should, of course, use your own start and end values):

```
2085 A9 00     COLDST LDA #$00
2087 85 20            STA $20
2089 A9 2D            LDA #$2D
208B 85 21            STA $21   ; user space start
                                  at $2D00
208D A9 FF            LDA #$FF
208F 85 22            STA $22
2091 A9 3F            LDA #$3F
2093 85 23            STA $23   ; user space end at
                                  $3FFF
2095 A0 00            LDY #$00  ; zero Y register
2097 4C AA 20         JMP $20AA ; for rest of init
     ......
20AA D8               CLD       ; existing code
20AB A5 20            LDA $20
        etc
```

In the following warm start code, the Basic stack pointer $0026-27 is made equal to top-of-user-space pointer $0022-23. The worse this mod can do (I hope!) is to prevent the use of byte $3FFF in the Basic stack.

I have not yet had any problems in using EZLOAD, but Murphy syas that someone out there will, and probably the first time out. I would be interested in any comments or suggestions.

```
2CB2              EZLOAD ORG   $2CB2

                  ZERO PAGE LOCATIONS

2CB2              TOPL   *     $0022  TOP LIMIT OF
2CB2              TOPH   *     $0023  USER SPACE
2CB2              SPL    *     $0026  T-B STACK
2CB2              SPH    *     $0027  POINTER
2CB2              ALO    *     $0082  TINY'S
2CB2              AHI    *     $0083  VARIABLE "A"
2CB2              FLAG   *     $00F8  LOAD ON/OFF SW
2CB2              POINTL *     $00FA  POINTER FOR
2CB2              POINTH *     $00FB  LOAD ROUTINE

                  KIM SUBROUTINES

2CB2              PACKT  *     $1A00  CONV ASCII/HEX
2CB2              OUTCH  *     $1EA0  OUTPUT CHAR
2CB2              INCPT  *     $1F63  INCR LOAD PTR

                  SET T-B OUTPUT JMP VECTOR AT $2009
                  TO ADDRESS $2CB2

2CB2 48           ENTRY  PHA          SAVE CHAR
2CB3 20 A0 1E            JSR   OUTCH  THEN PRINT IT
2CB6 C8                  INY          ZERO Y-REG
2CB7 68                  PLA
2CB8 C9 0D               CMPIM $0D    WAS IT CR?
2CBA F0 0A               BEQ   SETFLG EXIT LOAD MODE
2CBC 24 F8               BITZ  FLAG   LOAD MODE ON?
2CBE 70 09               BVS   ALOAD  YES - 1ST CHAR
2CC0 30 0C               BMI   BLOAD  YES - 2ND CHAR
2CC2 C9 5C               CMPIM '\     PREFIX CHAR?
2CC4 D0 02               BNE   OUT    NO - SKIP
2CC6 85 F8        SETFLG STAZ  FLAG
2CC8 60           OUT    RTS

2CC9 06 F8        ALOAD  ASL   FLAG   TOGGLE BIT
2CCB 4C 00 1A            JMP   PACKT  1ST NYBBLE
2CCE 46 F8        BLOAD  LSR   FLAG
2CD0 20 00 1A            JSR   PACKT  CODE BYTE IN ACC
2CD3 91 22               STAIY TOPL   PARK IT
2CD5 A6 26               LDXZ  SPL    NOW DEC
2CD7 D0 02               BNE   SKIP   STACK PTR
2CD9 C6 27               DECZ  SPH
2CDB A5 27        SKIP   LDAZ  SPH    COPY TO
2CDD 85 FB               STAZ  POINTH LOAD PTR
2CDF 85 83               STAZ  AHI    & VAR "A"
2CE1 CA                  DEX
2CE2 86 26               STXZ  SPL
2CE4 86 FA               STXZ  POINTL
2CE6 86 82               STXZ  ALO
```

```
2CE8 C8          SHUFL  INY              MOVE ALL
2CE9 B1 FA              LDAIY  POINTL  BYTES DOWN
2CEB 88                 DEY              ONE PLACE
2CEC 91 FA              STAIY  POINTL
2CEE 20 63 1F           JSR    INCPT
2CF1 A5 FA              LDAZ   POINTL  CK IF
2CF3 C5 22              CMPZ   TOPL    ALL DONE?
2CF5 A5 FB              LDAZ   POINTH
2CF7 E5 23              SBCZ   TOPH
2CF9 90 ED              BCC    SHUFL   MORE
2CFB 60                 RTS            NEXT CHAR..
```

```
                SAMPLE ORG   $0200

                THIS IS A SAMPLE MACHINE-CODE ROUTINE
                TO ILLUSTRATE USES OF EZLOAD

                SET UP A NUMERICAL ARRAY OF 128
                16-BIT ELEMENTS IN MEMORY SPACE
                2A00-2AFF, INDEXED BY 0 TO 127

                READ ROUTINE, R=USR(A,I), WHERE R=CONTENTS
                OF ARRAY(I), A=ADDRESS, I=SUBSCRIPT

0200 98         READ   TYA             TRANSFER INDEX
0201 0A                ASLA            MULTIPLY BY 2
0202 AA                TAX             USE FOR INDEXING
0203 BD 00 2A          LDAAX $2A00     INTO ARRAY
0206 E8                INX             NOW GET
0207 BC 00 2A          LDYAX $2A00     HIGH BYTE
020A 60                RTS

                WRITE ROUTINE, Z=USR(B,W,I), WHERE Z=DUMMY
                B=ADDRESS, W=VAL TO BE STORED, I=SUBSCRIPT

020B 86 F9      WRITE  STXZ  $F9       PARK X FOR NOW
020D 0A                ASLA            SUBSCRIPT * 2
020E AA                TAX             USE FOR INDEXING
020F 98                TYA
0210 9D 00 2A          STAAX $2A00     STORE LO BYTE
0213 A5 F9             LDAZ  $F9       GET HI BYTE
0215 E8                INX             ..AND
0216 9D 00 2A          STAAX $2A00     STORE IT
0219 60                RTS
```

```
1 REM \980AAABD002AE8BC002A60
2 REM \86F90AAA989D002AA5F9E89D002A60
3 REM
4 REM PROGRAM TO DEMO USE OF EZLOAD
5 REM
6 REM MACHINE CODE CREATES ARRAY READ AND WRITE FUNCTIONS
7 REM BASIC PROGRAM LOADS 64 RANDOM NUMBERS AND PRINTS THEM
8 REM THEN SORTS THE ARRAY AND PRINTS THE RESULTS
9 REM
10 B=A+11:C=0
20 Z=USR(B,RND(1000),C):C=C+1:IF C<64 GOTO 20
30 GOSUB 100
40 REM SORT THEN PRINT
50 R=63
60 F=0:C=0:L=R
70 IF USR(A,C)<=USR(A,C+1)GOTO 90
80 T=USR(A,C):Z=USR(B,USR(A,C+1),C):Z=USR(B,T,C+1)
85 F=1:R=C
90 C=C+1:IF C<L GOTO 70:IF F=0 GOSUB 100:GOTO 60
95 END
100 C=0:PR
110 PR USR(A,C),:C=C+1:IF C-C/8*8=0 PR:IF C<64 GOTO 110
120 PR:RETURN
```

:RUN

985	633	946	338	310	186	51	816
230	248	700	186	143	[illegible]	[illegible]	456
126	831	161	173	233	681	268	869
344	477	673	609	187	981	597	496
244	58	256	541	142	917	365	183
210	263	510	333	967	420	560	145
370	774	487	919	46	838	342	614
340	606	534	318	995	326	614	695

46	51	58	126	142	143	145	161
173	183	186	186	187	210	230	230
244	248	256	263	268	310	318	326
333	338	340	342	344	365	370	420
456	477	487	495	498	510	534	541
560	597	606	609	614	614	633	666
673	681	695	700	774	816	831	838
869	917	919	946	967	981	985	995