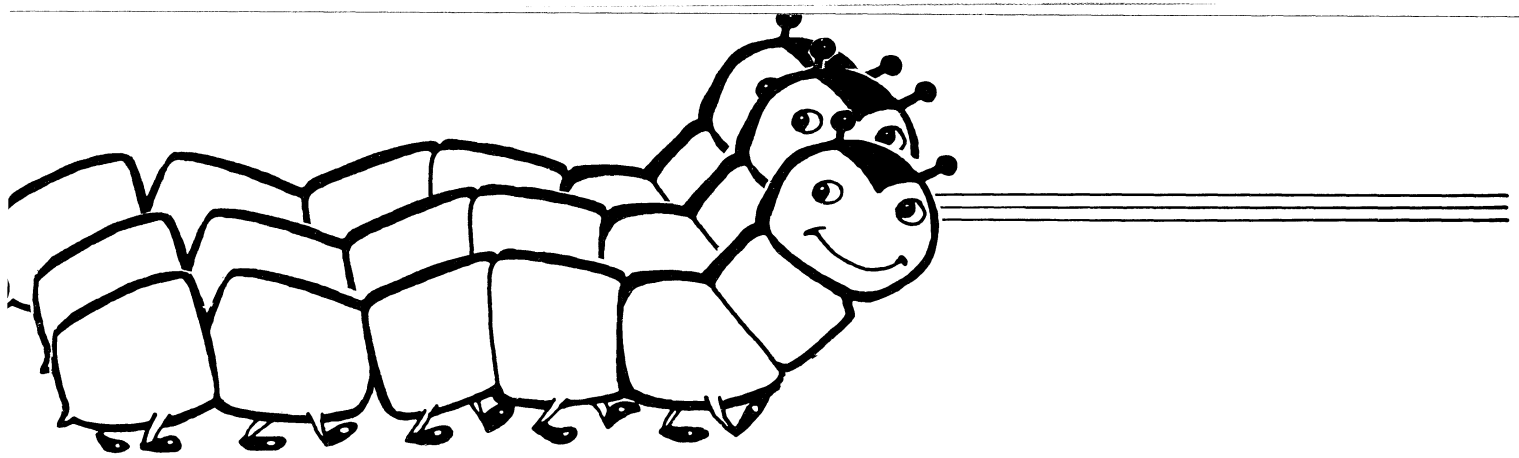


Beat the bugs in Applesoft's error handling

By ANTHONY WICKSTEAD

THERE is a bug in Applesoft's error handling routine which causes it to "forget" about FOR . . . NEXT loops and subroutine calls whenever an error occurs. Although this is documented in the Applesoft manual it makes the writing of programs, which are both well-structured and which do their own error handling, very difficult. To take a very simple example the following program:

```
10 HGR : HCOLOR  = 7
20 FOR J = 0 TO 279
30 ONERR GOTO 50
40 HPLOT J, SQR
   ((J - 140) / 140)
50 POKE 216,0 : NEXT J
```

will halt with the error message NEXT WITHOUT FOR after going through the loop once instead of plotting a graph of the square root function on the right hand side of the screen.

The ITT2020 is almost identical to the Apple II except for its high resolution graphics and timing. One difference between its version of Applesoft, Palsoft and Applesoft itself, is that this error handling bug has been fixed. Nevertheless, the Palsoft manual contains the same warning about restarting FOR . . . NEXT loops and subroutines that the Applesoft manual contains, so this correction is relatively little known.

Information about subroutine calls and FOR . . . NEXT loops is placed on the 6502 stack by the Basic interpreter. We can use the existence of Palsoft to compare the state of the stack after an error occurs in Applesoft with what it should be. Luckily it turns out that Applesoft has not thrown away the information about FOR . . . NEXT loops and GOSUB return addresses, but has merely placed "garbage" after them, making them inaccessible.

If we run the program:

```
 10 POKE 768,186 : POKE
    769,0:ONERR GOTO 100
 20 PRINT 1/0
100 CALL 768
```

the machine will halt in monitor with the X-register containing the stack pointer at the time of entry to this machine code. The stack pointer itself will have been changed by the monitor's break handling routine, so we need the POKEs (186 decimal = $BA is the TSX instruction and 0 is a BRK) to establish a short program to keep a note of it for us. Running this program in Applesoft we find that the stack pointer is EF (we give all subsequent numbers in hexadecimal) so the next stack position available is 1EF and the stack currently occupies 1F0 to 1FF. The data in this range is

01F0– 22 D8 D9 DD 00 ED DA 22
01F8– D8 C1 F1 00 01 01 0F 00

Running the program under Palsoft the stack pointer is F6 and the same area contains

01F0– 95 BA FD F8 FE 84 FF 22
01F8– D8 C1 F1 00 01 01 0F 00

The stack in use by Palsoft has the same values as are in the same locations when Applesoft is in use, so replacing the stack pointer by F6 will allow Applesoft to return with the stack in the same state as it would be if Palsoft were in use.

So far this does not help us much as this is only going to be correct if no FOR . . . NEXT loops or subroutine calls are active. To establish a general correction routine we need to know the following facts, (which can be established by examining the stack in a manner similar to that used above):

(1) Each subroutine call places seven bytes on the stack, the last of which is B0.
(2) Each FOR . . . NEXT loop places 23 bytes on the stack, the last of which is 81.
(3) Each CALL to a machine code routine places only the return address (22 D8) on the stack.
(4) The garbage on the stack is of variable length (depending on the nature of the error) but always starts with the address 22 D8 and does not contain that address anywhere else.

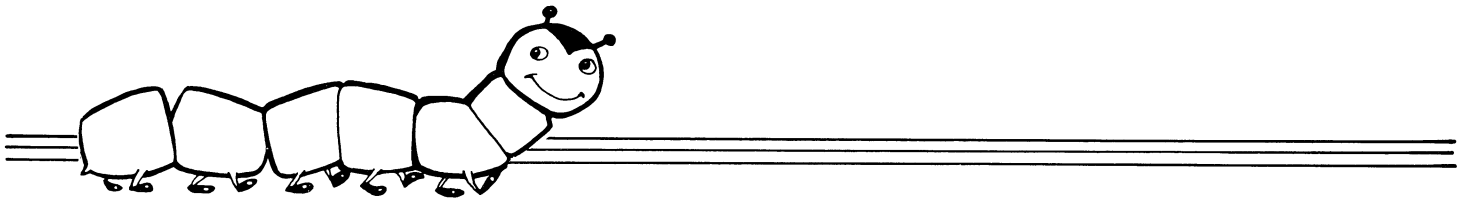
The variable length of the garbage means that we cannot just cut the stack back by a fixed amount. On the other hand the fact that the garbage starts with the same address as the return address for a CALL means that we need not worry about saving a return address for our correction routine.

The routine in Listing 1 simply searches back through the stack until it finds one of the sequences 22 D8 B0 or 22 D8 81 and sets the stack pointer to point at the next location (i.e. one less than the least significant byte of the address of the 22) before its own RTS takes the 22 and D8 off the stack.

If no such pattern is found then no subroutine calls or FOR . . . NEXT loops are active and the stack pointer is set to F6 (1F7 and 1F8 will contain the correct return address whether or not an error has occurred). Although slightly more complex than cutting the stack by a fixed amount, this process means that the correction routine will leave the stack in the correct state if no garbage is there, either because no error has occurred or because the program is running on an ITT2020.

To use this routine, type it into memory using monitor and then BSAVE it (its length is $26). It is written in position independent code so it can be placed anywhere. At the start of your program BLOAD the routine to a convenient position (the area starting at $300 (=768 decimal) is safe and is large enough, provided it is not being used for any other purpose). Now make the first statement in any error handling routine be a call to the

→



```
START    BA          TSX             ;copy stack pointer
FINDKEY  BD 00 01    LDA $100,X      ;look at byte on stack
         C9 B0       CMP #$B0        ;keyword for GOSUB
         F0 0B       BEQ KEYFND      ;if found check for correct return
                                     ;address after it
         C9 81       CMP #$81        ;keyword for FOR
         F0 07       BEQ KEYFND      ;if found check for correct return
                                     ;address after it
NEXTPOS  E8          INX             ;previous stack position is one
                                     ;address higher
         D0 F2       BNE FINDKEY     ;look again
         A2 F6       LDX #$F6        ;if no GOSUB or FOR...NEXT loop
         9A          TXS             ;active stack pointer should be F6
         60          RTS             ;for a correct return
KEYFND   BD FF 00    LDA $FF,X       ;if keyword found
         C9 D8       CMP #$D8        ;last two bytes should be D8
         D0 F2       BNE NEXTPOS
         BD FE 00    LDA $FE,X
         C9 22       CMP #$22        ;and 22
         D0 EB       BNE NEXTPOS
         CA          DEX             ;if so, push pointer past keyword
         CA          DEX             ;D8
         CA          DEX             ;and 22
         9A          TXS             ;before placing in stack pointer
         60          RTS             ;and returning to BASIC
```

Listing 1: Assembler listing of Applefix

```
10 S=768 : REM START OF ERROR HANDLING BUG FIX. CAN BE ANYWHERE.
20 FOR J=0 TO 37:READ A:POKE S+J,A:NEXT J
30 DATA 186,189,0,1,201,176,240,11,201,129,240,7,232,208,242,162,
   246,154,96,189,255,0,201,216,208,242,189,254,0,201,34,208,235,
   202,202,202,154,96
```

Listing 2: Applesoft program lines to POKE Applefix into memory

first location of this code. Thus if we have saved it on disc under the name Applefix, the program:

```
 5 PRINT CHR$(4);"BLOAD
   APPLEFIX,A$300"
10 HGR : HCOLOR   = 7
20 FOR J = 0 TO 279
30 ONERR GOTO 50
40 HPLOT J, 80 - 80 * SQR
   ((J - 140) / 140)
50 CALL 768 : POKE
   216,0 : NEXT J
```

will produce a graph of the square root function.

As far as I know the routine should work with a cassette-based system as well as the disc-based system on which it was developed. Cassette users, as well as some disc users, will prefer to use Applesoft program lines to POKE the code into memory.

Listing 2 gives such lines to place the routine into memory starting at address decimal 768.

It might be worth mentioning here another bug in Applesoft (and in Palsoft) which affects writing your own error handling routines. After executing an ONERR GOTO statement, program execution resumes with the next line of the program rather than the next statement.

The solution to this problem is simply to ensure that such a statement is always at the end of a multi-statement line or is in a line of its own.

Finally let me reiterate the need to restore normal error handling (by POKE 216,0) as soon as expected errors have been dealt with. If this were omitted in the first program in this text then it would hang, as the NEXT WITHOUT FOR error causes a jump to line 50, which causes a NEXT WITHOUT FOR error, which causes ... 🍎