APPLESOFT BANDAIDS

APPLE UTILITIES

A number of subtle and not-so-subtle bugs in Applesoft have surfaced over the years. Now you can have a version that corrects most of these bugs.

I recently decided to make some changes to the routines in my Apple II's ROM. While I was at it, I thought, why not fix a bug described by Cornelis Bongers in *All About Applesoft* [1]. My aid would be the symbolic disassembler included with Glen Bredon's Merlin program [2], which does a superb job disassembling Applesoft. I had not used it since I received it with my Big Mac.C Assembler, but without a doubt — now was the time.

I discovered that there was more wrong in Applesoft than I had previously imagined. Applesoft has several bugs that need to be patched up.

In this article fourteen bugs in Applesoft are discussed. The machine language program (**Listing 1**) fixes nine of them by copying Applesoft into the language card and installing patches. These bugs are:

1. The multiplication routine doesn't preserve the proper number of significant digits under some circumstances.
2. References to certain illegal line numbers cause a crash instead of an error.
3. The numeric string evaluation routine doesn't reject values with more than one decimal point.
4. All statements on the same line after an ONERR-GOTO statement are ignored.
5. When an ONERR trap is in effect, an error with a numeric GET statement causes improper error and line number information to be recorded, particularly when a second numeric GET occurs.
6. Under certain circumstances, a RETURN does not properly terminate an open FOR-NEXT loop.
7. An expression after the TO statement may be improperly evaluated.
8. The integer −32768 is not allowed.
9. An error with a numeric GET statement causes an inappropriate message and no line number to be reported.
10. HIMEM is not set properly when DOS isn't booted.

In addition, two bugs that can't be remedied with patches within the Applesoft ROM area are discussed, including references to published methods for correcting them. These problems are:

1. The random number generation routine does not generate truly random numbers. At some point, it starts to repeat the same sequence.
2. The garbage collection routine is inefficient, causing long delays.

Applesoft Bandaids does not correct the most familiar Applesoft bug, the stack problem with ONERR-GOTO. Programmers have learned to deal with it routinely and a correction would cause much existing soft-

ware to malfunction. (It turns out to be a single missing TXS instruction.)

The following items, while not actually bugs in Applesoft, have been corrected for the sake of convenience:

1. MID$, LEFT$ and RIGHT$ expressions that evaluate to zero length strings cause an error. The enhancement causes a null string to be returned.
2. CALL without an argument causes an error. The enhancement causes a jump to the Monitor.
3. When an error message is printed to the screen, the line where the error occurred is also listed.

FIGURE 1: Illustration of Multiplication Bug Code

```
                                Normal              With Bug

                                A=ØØ  C=1           A=ØØ  C=Ø
$E8FØ   ADC #$Ø8                A=Ø9  C=Ø           A=Ø8  C=Ø
        BMI                     Not taken           Not taken
        BEQ                     Not taken           Not taken
        SBC #$Ø8                A=ØØ  C=1           A=FF  C=Ø
        TAY
        LDA $AC
        BCS   $E911             Taken               Not taken

        ...
$E911   CLC                     C=Ø                 C=Ø
        RTS                     Return to LDA
                                next mantissa
                                byte with
                                Carry Clear
```

4. Spaces are significant as delimiters when entering an Applesoft statement (i.e., IF F OR K... is parsed as it is typed, rather than IF FOR K).

HOW TO USE THE PROGRAM

Just BRUN BANDAIDS to copy Applesoft to the language card area of memory, install the patches, and leave the soft switches so that the patched copy is in effect. You will find that the first nine bugs described above have been fixed and the enhancements have been added. If you have access to an EPROM programmer, you can create a permanent copy of the patched Applesoft ROM. This process is discussed in the section Putting It All Into ROM.

ENTERING THE PROGRAM

If you have an assembler that can handle multiple ORGs, conditional assembly and 65C02 mnemonics, enter the source code from **Listing 1**, use the proper code for your machine in **line 9** and assemble it. If you don't have an appropriate assembler, enter the Monitor with CALL −151 and key in the hex code from **Listing 2**. If you have an Apple II Plus, save the program with the command:

BSAVE BANDAIDS,A$8000,L$183

If you have a IIc, keep the code from **Listing 2** in memory and also enter the code from **Listing 3**. Save the IIc version with the command:

BSAVE BANDAIDS,A$8000,L$1AA

If you have a IIe, keep the code from **Listing 2** in memory and also enter the code from **Listing 4**. Save the IIe version with the command:

BSAVE BANDAIDS,A$8000,L$19D

For help in entering *Nibble* listings, see "A Welcome to New *Nibble* Readers" at the beginning of this issue.

DESCRIPTION OF APPLESOFT BUGS

In this section, each of the bugs listed at the beginning of the article is described and analyzed in more detail.

The Multiplication Routine Bug

We begin by turning on the Apple and typing:

```
PRINT  1 + 1.ØØØØØØ11 :  REM HOW
MANY SIGNIFICANT FIGURES DO YOU
THINK WILL BE PRESERVED?
```

The bug responsible for the inaccuracy is in the multiplication routine and involves an error in the least significant eight bits. It only occurs for a small percentage of the possible multipliers. But since the multiplication routine is called by all of the transcendental functions — COS, SIN, ATN, EXP, and so on — these functions are also flawed.

This bug is particularly insidious, as it gives you incorrect results without giving you any clue that something is wrong. Thus, the unsuspected error spreads far and wide.

To investigate the bug further, enter the following lines:

```
10 FOR I=1 TO 999999 STEP
   .ØØØØØ15*I :  IF I*1 <> 1*I
   THEN  PRINT I*1; "<>"; 1*I;
   SPC(3); CHR$(7);
20 NEXT
```

This program checks the results of a multiplication against the results when the multiplier and multiplicand are exchanged. Normal Applesoft prints a product every few minutes. After I patched the bug, the program ran overnight (up to I=228) without printing anything. Note that the values printed tend to be slightly more than $N/8$ where n is an integer.

What's wrong here? Each floating-point number is stored as a one-byte exponent and a four-byte mantissa [3,4]. During a calculation, there is a fifth "guard" byte of the mantissa for each floating-point accumulator. Multiplication of the mantissas is performed by a shift-and-add process, which you may remember having learned in the third grade. However, you used decimal numbers and the computer uses binary numbers. In binary, the only single-digit multiplications involve a zero or a one. Multiplication by one or zero boils down to adding the multiplicand or adding zero (i.e., not adding). To demonstrate how the shift-and-add algorithm works, let's multiply two three-bit binary numbers:

Binary	Decimal
100	4 (multiplicand)
101	5 (multiplier)
100	1 × 4 = 4
000	0 × 4 = 0
100	1 × 4 = 4
10100	20

The multiplicand is shifted and added. The multiplier drives the shift/add process. The problem that Applesoft solves, though, is a little more complex: five eight-bit words are in each mantissa and the 6502 adds eight bits at once. Ultimately, the exponents are considered, but we need not consider them here, since that calculation is not involved in the bug.

Now that we have some idea about the overall algorithm, let's use the Apple Mon-

FIGURE 2: Illustration of Numeric String Evaluator Bug Code

```
                          First Decimal      Second Decimal
                              Point              Point

                          C=1   $9B=$ØØ      C=1   $9B=$8Ø
$EC98 ROR $9B             C=Ø   $9B=$8Ø      C=Ø   $9B=$CØ
      BIT $9B             M=1   V=Ø          M=1   V=1
      BVC                 Taken              Not taken
$EC9E (Adjust exponent)
```

itor to list the offending section of code. To get into the Monitor, type **CALL −151** and then **E994L**.

What you see is a series of LDA instructions (which we can translate as "load a byte of the mantissa of the multiplier") and JSR $E9B0 instructions. $E9B0 is a subroutine that does the multiplication for one byte of the multiplier. The routine that actually performs the shift/add process, byte by byte, starts at $E9B5. The code between $E9B0 and $E9B5 checks to see if the mantissa byte is zero; if it is, then a shortcut can be taken (JMP $E8DA) which will shift the multiplicand eight bits in one fell swoop. If you replace the JMP $E8DA with NOPs (No Operation, "do nothing" instructions), then the routine runs a little more slowly but the test program doesn't detect any discrepancies when the two numbers are transposed.

Why bother to have a special fast routine for one special case out of 256? In other words, there are 255 ways a mantissa byte can be nonzero, so why provide a shortcut that is only used for one of the 256 possibilities? Integers and other even numbers (e.g. $n/8$, where n is an integer) are common in most programs. These frequently used numbers will have several mantissa bytes equal to zero.

We now have strong evidence that something about the $E8DA branch causes the errors. Glen Bredon states that the problem is that the Carry bit needs to be set when we get to $E8DA and that it will not be set if the prior mantissa byte was (also) zero.

Before going to look at $E8DA, you might want to assure yourself that the Carry bit is always set after going through the usual $E9B5 route. Essentially, the bit set by the ORA #$80 at $E9B6 is shifted into the Carry bit on the eighth time through the LSR at $E9DF. The RTS at $E9E2 returns to the next LDA (load a mantissa byte) in sequence with the Carry bit set.

At $E8DA, the eight-bit shift is performed without disturbing either the zero value in the A-Register or the contents of the Carry. This routine is called from several places in Applesoft other than just the multiplication routine, however. It is capable of more than just eight-bit shifts; the routine shifts n places when n is in the Accumulator on entry.

To understand what happens at $E8F0, see **Figure 1**. An incorrect multiplication occurs when a multiplier has two consecutive zero mantissa bytes. The first zero byte proceeds through $E8DA as shown in the Normal column. The second one starts with Carry clear and proceeds as shown in the With Bug column. The multiplicand undergoes extra, erroneous shifting. A final, nonzero multiplier mantissa byte adds the erroneously shifted multiplicand to the result. The final nonzero byte is *not* present unless the multiplier is slightly more than $n/8$. If it is much more than $n/8$, then there won't be two zero mantissa bytes.

We have already seen one patch to eliminate the bug, at the cost of slower multiplication for certain commonly encountered multipliers. Another, neater way of accomplishing the same thing would be to replace each JSR $E9B0 instruction with JSR $E9B5. A third, better approach is simply to set the Carry flag each time before calling $E8DA. This is what the patch in Applesoft Bandaids does. Note, however, that the JSR $E9B5 approach actually saves a few bytes rather than requiring extra space; there may be situations where that is the preferred patch.

TABLE 1: Program Functions

Lines	Function
Making Free Space	
118-181	Shorten cold start
215-220	Old RUN on any input
226-233	Use unused constants
Bug Fixes	
202; 320-325	Use new 5-byte constant for −32768
203-206; 294-299	Fix line number bug (line 440010)
226-233; 250-254	Fix numeric string evaluator to produce error with multiple periods
263-267	Stack properly maintained when RETURN occurs before expected NEXT
221-224; 243-248; 301-305	Properly evaluate TO argument
281-285; 307-312	Add SEC to multiplication routine
314-318	Look for colon or end of line after ONERR-GOTO
287-292	Generate meaningful error message for numeric GET
164-166; 127-128	Fix no-boot bug by properly setting HIMEM
Amenities	
184-201; 269-273	Print erroneous line after error message
211-213; 275-279	Causes CALL without argument to assume −151
235-241	Add recognition of significant spaces
256-261	Causes zero-length string functions to return null string

The Line Number Bug

Open your disk drive doors before you try this one! It just might destroy your disk before it finally crashes. Turn on your Apple and type:

```
440010
```

What you did was to tell Applesoft to delete a line whose number is outside the allowed Applesoft range. Instead of generating a SYNTAX ERROR message, the machine crashed, so the bug is in many ways worse than the first bug we discussed. This bug affects any reference (e.g., GOTO, GOSUB) to a six-digit line number. (Actually, certain seven-digit, eight-digit, nine-digit, etc. line numbers will also cause Applesoft to crash; it is the first five digits and the presence of additional digits that are at the root of the problem.)

Now let's get into the Monitor and type **DA00L**. The first instruction is a JSR that gets the next character of the BASIC program into the A-Register and clears the Carry if it is numeric or sets the Carry if it is not numeric. JSR $DA0C executes the routine to convert text into line numbers. At $DA0C, a two-byte number at $50,$51 is set to zero; this is where the line number is built up as characters are read from left to right.

At $DA12, the Carry bit is tested; if the character is not a number, an RTS is issued, and the new character is saved in $0D. The routine then copies the contents of $51 and $5E and compares the contents of $51 (the high byte of the line number being formed) to the number $19. Since this is the high byte, multiplying the decimal equivalent (25) by 256 yields 6400. Since another digit (in $0D) will be appended to the number, this is actually a test for a line number over 64,000. This is where our six-digit line number should generate its error message. The code below this point multiplies the forming line number in $50,$51 by ten and then adds the new digit. Finally, at $DA40, the program JSRs to get the next character of the BASIC program and loops back to $DA12.

At $DA1F, our special six-digit line number left the loop. What the authors of Applesoft wanted to do here was to generate a SYNTAX ERROR message. They needed to get to $D981 where there is a JMP in order to print the message. But the 6502's branch instruction won't reach that far; it

can only go backwards 126 bytes. What they did was to find an intermediate stop at $D9F4, where another routine decides whether to generate a SYNTAX ERROR message (it is looking for a GOTO after an ON).

At $D9F4, there is a comparison that was intended to always result in a branch to $D981. This comparison compares the old $51 value (i.e., the high-order byte of the line number) to the constant $AB, which is the token for GOTO. Hexadecimal $AB is 171 decimal, and 171*256=43776; the line numbers that make Applesoft blow up are 437760 through 440319. I have a strong suspicion that the Applesoft authors knew this bug was present when they wrote this code, but that space limitations caused them to ignore it.

When the value $AB hits $D9F4, no error message is printed. Applesoft tries to treat our six-digit line number as though it were in the middle of an ON-GOTO. It pulls one byte ($06) of the return address (that should have gotten us back to $DA06) off the stack at $D9FC and shortly after does an RTS into limbo.

The Numeric String Evaluator Bug

Let's demonstrate the next bug by typing:

```
PRINT 1.2.2: REM TWO DECIMAL
POINTS IN ONE NUMBER MUST BE AN
ERROR
```

The bug is in the numeric string evaluator, although it only shows up in the case of PRINT statements. Again, the bug is a missing instruction — but this time it is a three-byte JMP to the error processing routine. A flag is tested to make sure that a number has just one decimal point, but no error message is actually printed.

The routine to evaluate a floating-point number begins at $EC4A. Between $EC4A and $EC52, a block of zero page locations is set to zero. Note that this includes $9B, which is used to flag the presence of a decimal point in the ASCII string being converted to a number. At $EC66, the next character is compared to $2E, which is the code for a period; if it is a period, program flow branches to $EC98. At $EC98, the plot thickens (see **Figure 2**). If Applesoft does not take the branch at the BVC instruction, there is a second decimal point in the number. When it is time to print an error message, Applesoft instead falls into a routine that adjusts the exponent of the number being formed.

I've used the ILLEGAL QUANTITY error message for the second decimal point error in Applesoft Bandaids.

ONERR-GOTO Remainder of Line Ignored

This bug in ONERR-GOTO is demonstrated with:

```
10 ONERR GOTO 50 : PRINT "LINE 10"
20 END
30 STOP
```

FIGURE 3: Illustration of Improper Evaluation of TO Argument Code

```
        Location:   9D 9E 9F A0 A1 A2 AC   A-Register
                    A3 FF FF FF FF 00 E0
$D79C LDA $A2       A3 FF FF FF FF 00 E0   00
      ORA #$7F                             7F
      AND $9E                              7F
      STA $9E       A3 7F FF FF FF 00 E0   7F
                    Sets up return for call to $DE20
      JMP $DE20
$DE20 JSR $EB72     A3 7F FF FF FF 00 E0
                    A3 80 00 00 00 00 00
```

The PRINT statement is ignored. ONERR is processed at $F2CB. At $F2E3, a JSR $D9A6 skips over the rest of the line — this is useful for REM statements in which the colon (:) is *not* significant. For other statements, the routine is usually called by JSR $D9A3, so either the end of the line or a colon terminates the statement. I was unable to discover any ill effects from changing the instruction at $F2E3 to JSR $D9A3 in Applesoft Bandaids.

Successive GETs After ONERR-GOTO

The Applesoft manual documents a bug that occurs when there are successive errors in GET statements, without an intervening correct statement. The bug is demonstrated by the following code:

```
10 ONERR GOTO 60
20 GET N : REM TYPE A LETTER TO
   GET AN ERROR
30 END
60 PRINT "ERROR #"; PEEK(222);"
   IN LINE "; PEEK (218) + PEEK
   (219) * 256: RESUME : REM
   RETRY GET!
```

The fix for the GET bug (below) corrects this one as well; it will make a corresponding change in the error number reported.

The RETURN WITHOUT GOSUB Bug

The next bug is thoroughly discussed in at least two previous articles [1,5]. It can be demonstrated with the following short program:

```
5  LOMEM:  7677:GOSUB 10:END
10 FOR I = 1 TO 1: RETURN
```

When you run this program, a RETURN WITHOUT GOSUB error message appears. To eliminate the message, change the location of the variable I by changing LOMEM to 7678 (or deleting LOMEM).

The bug occurs when both of the following apply:

1. A FOR loop is still active at the occurrence of a RETURN statement.
2. The index variable of the FOR loop ('I' above) is stored at address $xxFF.

Of course, writing well-structured code is the best prevention against the first conditions.

The RETURN statement, whose code begins at $D96B, calls a subroutine at $D365, which clears the FOR loop information from the stack. This terminates any FOR loops that are still active. The clearing subroutine is also called by a NEXT statement. It supports the NEXT A syntax by comparing the address of the index variable(s) that the FOR statement put onto the stack against the address of the variable specified in the NEXT A statement (saved in $85,$86). If the variable addresses don't match, the information from that FOR is cleared off the stack and the index variable of the previous FOR is checked to see if it matches the NEXT A variable. If the stack is found to contain a non-FOR entry before a match is found, then a NEXT WITHOUT FOR error message is generated by the NEXT routine (not by the subroutine!).

The intention of the Applesoft authors was that any unterminated FOR loops be cleared from the stack before returning from the subroutine when a RETURN is encountered. To do this, it would be necessary to prevent a match between the FOR loop variables and the address where the NEXT syntax puts its variable ($85,$86). So the authors stored $FF in $85 at $D96F.

Unfortunately, $FF in $85 does not always prevent finding a match, i.e., if the index variable is located at $xxFF. In this instance, the routine at $D365 returns without clearing the FOR loop information from the stack. The return from subroutine portion of the RETURN statement finds this information on the stack instead of the expected GOSUB token (which GOSUB stores on the stack). At $D975, the stack token is compared to the GOSUB token and at $D979, the branch to print the error message is taken.

The fix developed by Cornelis Bongers is elegant: $FF is stored in $86 rather than in $85 before $D365 is called, so a match cannot occur. The index variable address would have to be $FFxx, and this is not possible within Applesoft's variable space.

The TO Value Bug

Once again, let's begin by demonstrating the bug with this line:

```
FOR I = 0 TO 2^35 - 1 : PRINT I
: NEXT : REM THIS LOOP WILL NOT
REALLY TAKE ALL NIGHT, BUT IT
SHOULD!
```

In this case, the TO value is misinterpreted during its processing.

An understanding of the floating-point number format is important to the following discussion. An ideal starting point is Eric Goez's "Real Variable Study" [6]. The error occurs as the number is compressed to the four-byte format in which it is saved on the stack. The text expression corresponding to the TO value is evaluated at $D799 by JSR $DD67. On return, the sign is in $A2 and the fifth byte of the evaluated expression's mantissa is in $AC. The rest of the floating-point value is in $9D.A1.

After evaluating 2^35 − 1, processing proceeds as shown in **Figure 3**. The floating-point value at the end of this process is negative. The purpose of the JSR $EB72 is to increment the mantissa by one if the fifth (guard) byte is over $80, i.e., to round up by one bit in the fourth mantissa byte. The instructions between $D79C and $D7A3 set the high mantissa bit based on the contents of $A2. Each part works well by itself, but the parts are executed in the wrong order, so the sign bit may be changed by the rounding operation. If we arrange things so that JSR $EB72 precedes packing the sign bit, then the calculation is done correctly.

The −32768 Bug

Bredon has concluded that the following feature is a bug. However, because it works in the same way as it is documented to work, it is properly called a "feature."

Integer variables are stored in two bytes. Those 16 bits can represent 65536 different values. Two's complement numbers can have the values 0 through 32767 and −1 through −32768. However, Applesoft's integer variables are only allowed to assume 65535 different values and −32768 is not allowed.

To demonstrate this feature/bug, enter:

```
A% = -32768
```

and you'll get an ILLEGAL QUANTITY error.

There are a variety of ways that this feature could become a significant problem. Most obvious to me is that it is impossible to represent an address in RAM with an integer variable. Therefore, this feature may need fixing, like the bugs we've discussed so far.

Let's see how the ROM code works. First, the expression is evaluated as a floating-point number. Then at $E10C, the floating-point number is tested to see if it is within the allowed integer range. The exponent is tested; if it is less than $90, then the absolute value is 32767 or less. If it passes this test, then it is converted to an integer with a jump to $EBF2.

If it is not less than 32767, then it is tested to see if it's equal to −32768. The constant −32768 is stored at $E0FE (90 80 00 00). The address of the constant is loaded into the A and Y Registers at $E112. However, the subroutine called to perform the comparison compares five rather than four bytes. The $20 from the subsequent JSR instruction is used as its low-order byte. As a result, this second test fails for −32768. The test does succeed for −32768.00049.

The GET Bug

Even Bredon missed the next bug. Type:

```
10 GET K
```

When you RUN this, try typing a letter rather than a number. You get a SYNTAX ERROR message rather than something to do with your bad run-time input, and unlike most error messages, no line number is shown. You don't know where the error occurred and have few clues as to what went wrong. Fortunately, GET is rarely used with a numeric type variable.

The GET statement is handled starting at $DBA0. It turns out that GET, READ and INPUT are handled by a single main routine. At various points, that routine needs to know which of the three types of statements it is working on. A flag in location $15 is set to $40 for GET, to $80 for READ, and to $00 for INPUT. Input errors for all three types of statements are handled at $DB71; use the Monitor L command to list the code at $DB71.

The flag is tested to determine the statement type. If it is an INPUT statement, the program branches to $DB87; if it is a READ, the branch is to $DB7B. At $DB7B, the line number of the current DATA statement is copied into the current program line number. The error was in the input so a SYNTAX ERROR is indicated. If, at $DB71, the ROM routine is processing a GET, it loads $FF into Y and goes to $DB7F (in the middle of READ error processing). The $FF is stored in the high byte of the current line number at $76. This is a flag used by the BASIC interpreter to indicate direct mode. The program then jumps to $DEC9 to print the SYNTAX ERROR message.

Applesoft's authors intentionally destroyed the line number information, perhaps because of confusion about the ILLEGAL DIRECT ERROR (which is actually handled elsewhere). If we replace the lines at $DB77 with a JMP $DD76, the result is a TYPE MISMATCH error message and a correct line number. This seems more satisfactory.

The No Boot Bug

The next bug only occurs if DOS or ProDOS is not booted — so turn your Apple off, remove the disk, turn it on again and press Reset to get into Applesoft. Type in the following program:

```
5 GET A$ : PRINT VAL (A$)
```

When you run the program, type a number.

You can eliminate the bug by adding the following line:

```
2 HIMEM: 49151 : REM No Boot
```

The bug has been beautifully described by Bob Sander-Cederlof [7]. However, it seems to me that the blame lies not with the VAL function, as he asserts, but with the way a cold start sets HIMEM. The bug would certainly look different (and might not exist at all) on old Apple II's with 16K or 32K of RAM. This may well be how it escaped the notice of the Applesoft authors.

In a 48K or 64K Apple, a cold start sets HIMEM to 49152 ($C000 hex). In the example above, the first string created after startup is A$, and its value, the ASCII code of the number you typed, is stored at $BFFF. The next memory location ($C000) is not a RAM address; it is the keyboard input address — an I/O address.

The VAL function begins at $E707 with a JSR $E6DC. This sets $5E,$5F to point to the beginning of the string named in the call to VAL, and the length of the string is returned in A. A is preserved until $E71B, where it is added to the contents of $5E,$5F. The result is stored at $60,$61 to point to the byte after the string. The byte after the string is copied onto the stack and a zero is stored after the string. This zero flags the end of the string. Then the routine to evaluate a floating-point number is called ($EC4A, which we met in connection with the Numeric String Evaluator Bug). The expression is terminated by the zero and when the subroutine returns, the zero is replaced with the item saved on the stack.

However, since $C000 is not a RAM location, zero can't be stored there. $C000 continues to be read as containing the value of the last character typed. The expression evaluator reads characters until it finds a non-numeric character — usually, but not always, at $C010. [8].

The solution is to have the coldstart routine set HIMEM to 49151 ($BFFF) instead of $C000, or to always use DOS so that there is RAM after that first string. Using any DOS that is language-card resident will solve the problem, as well as having DOS in its regular location. The fix for this bug included in Applesoft Bandaids will only take effect if the code is placed in EPROM.

The Random Number Generator Bug

As detailed in two articles in the January 1983 issue of *Call-A.P.P.L.E.* [9,10], the random number generator is "fatally flawed." The authors of the first article present an interesting little program that demonstrates non-randomness while filling the Hi-Res screen with points chosen using the RND(1) function. A slight variation of that program is as follows:

```
 5 HGR2: HCOLOR=3
10 HPLOT RND(1)*280, RND(1)*192 :
GOTO 10
```

This program uses the random number generator to choose points to plot on the Hi-Res graphics screen. If the random number generator works well, the program eventually fills the screen with white dots. However, if the sequence of random numbers begins to repeat, then the screen does not fill and the same points are plotted over again. It is much easier to see the repetition develop this way than by searching long lists of random numbers, although this sort of program may easily underestimate the seriousness of the flaw in the random number generator.

The elegant substitute routine presented in the second article [10] is much longer than the built-in random number generator. Because the program is accessed through the USR function rather than RND, the fix doesn't preserve compatibility with existing programs.

The Garbage Collection Bug

Demonstrating this bug will take a program somewhat bigger than for previous bugs:

```
10 LOMEM:10000 : HIMEM:10070
20  FOR  I = ASC("A") TO ASC("Z")
 STEP 4:  A$= A$  +  CHR$(I)  +
    CHR$(I+1) + CHR$(I+2) + CHR$
    (I+3): NEXT
30 PRINT A$
```

Line 10 restricts the variable space, forcing Applesoft garbage collection. By eliminating line 10, you can see the correct program output. The bug occurs when a temporary string forces garbage collection to occur; line 20 forms temporary strings several times in the process of assigning a new value to the string variable A$.

Applesoft garbage collection is so slow, many programmers work around it whenever possible. Knowing that there's a bug in the built-in garbage collector just gives you one more reason to avoid it. For an excellent review of a technique for avoiding garbage collection, see [11]. On the other hand, there are a couple of published garbage collection routines you may want to look into. Cornelis Bongers' straightforward garbage collection routine [12] is a better choice than Randy Wiggington's fast garbage collection routine [13], but it is about $40 bytes longer than the built-in garbage collection routine.

ProDOS already includes a fast garbage collector. Those of you who are able to devote 20K to your operating system (instead of 10K for DOS 3.3) don't need to worry about slow garbage collection or the bug in it — unless you invoke the firmware garbage collector by accident. The latest version of Diversi-DOS also offers a fast garbage collector, without the large memory penalty of ProDOS [14].

GAINING SPACE

Before we patch the problems documented above, we need to find space within Applesoft to make the patches. Most long patches

LISTING 1: BANDAIDS Source Code

```
1     ********************
2     *      BANDAIDS      *
3     * by John R. Raines  *
4     * Copyright (C) 1987 *
5     * by MicroSPARC, Inc *
6     * Concord, MA  01742 *
7     ********************
8     * Merlin Pro Assembler
9     MACHINE  =     $2C          ;$00--II Plus, $2C--//c, $2E--enh. //e
10    *
11    GOWARM   =     $0000
12    GOSTROUT =     $0003
13    USR      =     $000A        ;vector to USR
14    A1L      =     $3C          ;pointer used by Monitor MOVE
15    A1H      =     A1L+1
16    A2L      =     $3E          ;ditto
17    A2H      =     A2L+1
18    A4L      =     $42          ;ditto
19    A4H      =     A4L+1
20    LINNUM   =     $50
21    LASTPT   =     $53
22    TXTAB    =     $67          ;start of prgm text
23    FRETOP   =     $6F          ;start of string storage
24    MEMSIZ   =     $73          ;HIMEM
25    CURLIN   =     $75          ;current line
26    FORPNT   =     $85          ;see ? APPLESOFT ERR
27    DSCLEN   =     $8F
28    JMPADRS  =     $90
29    DECPNT   =     $98
30    FPGEN    =     $A4
31    FACSGN   =     $A2          ;sign of FAC when it's unpacked
32    TXTPTR   =     $B8
33    TRCFLG   =     $F2          ;trace flag
34    SPEEDZ   =     $F1          ;actual SPEED is 256 minus this
35    OP_APPL  =     $C061        ;open apple key on //e, //c
36    IN       =     $0200
37    REASON   =     $D3E3
38    RESTART  =     $D43C
39    PARSE    =     $D56C
40    FNDLIN   =     $D61A
41    SCRTCH   =     $D64B
42    NXLST    =     $D6DA
43    CRDO     =     $DAFB
44    STROUT   =     $DB3A
45    OUTSP    =     $DB57
46    FRMNUM   =     $DD67
47    TYPERR   =     $DD76
48    SYNERR   =     $DEC9
49    PUSHFAC  =     $DE20
50    IQERR    =     $E199
51    RNDB     =     $EB72
52    EVAL     =     $EC61
53    INPRT    =     $ED19
54    ZPSTUFF  =     $F10B
55    NORMAL   =     $F273
56    WAIT     =     $FCA8
57    MOVE     =     $FE2C
58    MON      =     $FF65
60
61          ORG    $8000
62    START BIT    $C081
63          BIT    $C081        ;Write enable bank-sw. RAM, read ROM
64          LDA    #$D0
65          STA    A1H
66          LDA    #0           ;Start copying at $D000
67          STA    A1L
68          TAY
69    C1    LDA    (A1L),Y      ;copy ROM into bank-switched RAM
70          STA    (A1L),Y
71          INY
72          BNE    C1
73          INC    A1H
74          BNE    C1           ;all the way to $FFFF?
75    *
76          LDA    #>Q1
77          STA    A1H
78          LDA    #<Q1
79          STA    A1L
80    *
81    C2    LDY    #0
82          LDA    (A1L),Y
83          BEQ    DONE         ;END OF PATCH PROCESS?
84          STA    A4L
```

LISTING 1: BANDAIDS Source Code *(continued)*

```
85            INY
86            LDA   (A1L),Y
87            STA   A4H         ;A4L,H is destination address
88            INY
89            LDA   (A1L),Y
90            STA   A2L
91            INY
92            LDA   (A1L),Y
93            STA   A2H         ;A2L,H is last byte of source block.
94            INY
95            TYA               ;A=4
96            CLC
97            ADC   A1L         ;Update A1 to point to start of next patch.
98            STA   A1L
99            BCC   NOCARRY
100           INC   A1H
101 NOCARRY   LDY   #0
102           JSR   MOVE        ;MOVE PATCH INTO PLACE.
103           BCS   C2          ;ALWAYS BRANCH
104 *
105 DONE      BIT   $C082       ;WRITE LOCK L.C.
106           BIT   $C080       ;ENABLE L.C. READ
107           RTS               ;end of program to install patches
      ********************************************************
109 *
110 *DATA STRUCTURE IS:
111 *     2 BYTES "WHERE IT'S GOING"
112 *     2 BYTES "LAST BYTE OF THIS PATCH"
113 *     CODE OF THE PATCH
114 * AND REPEAT.  FINALLY ENDS WITH A ZERO BYTE AFTER LAST PATCH
115 *
116 Q1        DA    P1
117           DA    Q2-1
118           ORG   $F128
119 * ABBREVIATED COLDSTART ROUTINE FROM APPLESOFT
120 * THE ORIGINAL CONTAINS SOME OF THE FEW EXAMPLES
121 * OF INEFFICIENT CODE IN APPLESOFT.  A BUNCH
122 * WAS ALSO SAVED BY ASSUMING THAT THERE IS 48K
123 * OF LOWER RAM (APPLESOFT WAS WRITTEN LONG BEFORE
124 * THAT WAS A SAFE ASSUMPTION).
125 P1        LDX   #$FF        ;flag direct mode by $FF in high byte
126           STX   CURLIN+1    ;of current line pointer
127           STX   MEMSIZ
128           STX   FRETOP
129           LDX   #$FB
130           TXS               ;init stack
131           JSR   NORMAL
132           LDA   #$4C        ;JMP = $4C
133           STA   GOWARM
134           STA   GOSTROUT
135           STA   JMPADRS
136           STA   USR
137           LDA   #<IQERR
138           LDY   #>IQERR     ;USR() ==> ILLEGAL QUANTITY error
139           STA   USR+1       ;unless user inits
140           STY   USR+2
141           LDX   #28         ;length of page zero stuff
142 ZPCOPY    LDA   ZPSTUFF-1,X
143           STA   $B0,X
144           DEX
145           BNE   ZPCOPY
146 *
147 * X=0 after loop
148           STX   TRCFLG
149           STX   FPGEN
150           STX   LASTPT+1
151           TXA               ;A=0
152           STA   $800
153           PHA
154           LDA   #3
155           STA   DSCLEN
156           JSR   CRDO
157           LDA   #$01
158           STA   SPEEDZ
159           STA   IN-3
160           STA   IN-4
161           STA   TXTAB
162           LDA   #$55
163           STA   $52
164           LDA   #$BF        ;top of contiguous memory
165           STA   MEMSIZ+1
166           STA   FRETOP+1
167           LDY   #$08
168           STY   TXTAB+1
169           LDA   TXTAB
170           JSR   REASON
171           JSR   SCRTCH
172           LDA   #<STROUT
173           LDY   #>STROUT
174           STA   GOSTROUT+1
175           STY   GOSTROUT+2
176           LDA   #<RESTART
177           LDY   #>RESTART
178           STA   GOWARM+1
179           STY   GOWARM+2
180           JMP   (GOWARM+1)
181 * End of my cold start--free to F1D4
182
183
184 * PRINT LINE ERROR WAS FOUND IN ALONG WITH ERR MESSAGE
185 * FOR //C.  QUITE DIFFERENT FROM BASIS ROUTINE.
186 PLNERR    JSR   INPRT       ;print "IN nnnn"
187           LDA   CURLIN+1
188           LDX   CURLIN
189           STA   LINNUM+1
190           STX   LINNUM
191           JSR   FNDLIN
192           LDX   #$FF
193           STX   CURLIN+1    ;DIRECT MODE NOW
194           INX               ;X=0
195           STX   TXTPTR
196           LDY   #2
197           STY   TXTPTR+1    ;TXTPTR POINTS TO $200
198           TXA               ;A=0
199           STA   (TXTPTR),Y  ;$202=0  construct end of
200           STA   (TXTPTR,X)  ;$200=0  program flag.
201           JMP   NXLST       ;list the line & drop into direct mode.
202 M32768    HEX   9080000000  ;five byte -32768.

203 CHLINN    CMP   #$19        ;line # already >6400?
204           BCS   SYNRRR
205           RTS
206 SYNRRR    JMP   SYNERR      ;go directly to SYNTAX ERROR
208           HEX   8D          ;GET CHECKSUM CLOSE
208           HEX   8D          ;GET CHECKSUM CLOSE
208           HEX   8D          ;GET CHECKSUM CLOSE
208           HEX   8D          ;GET CHECKSUM CLOSE
208           HEX   8D          ;GET CHECKSUM CLOSE
208           HEX   8D          ;GET CHECKSUM CLOSE
208           HEX   8D          ;GET CHECKSUM CLOSE
208           HEX   8D          ;GET CHECKSUM CLOSE
210           HEX   99          ;GET CHECKSUM FOR APPLESOFT RIGHT!
211           ERR   $F1D0-*     ;following code must sit just above old CALL
212 GOMON     JMP   MON
213 CALL      BEQ   GOMON       ;CALL w/o argument goes to Monitor
      ********************************************
215           ORG   *-P1+Q1+4   ;ORG to where this LOADS rather than where
216 *                           ;it'll go
217 Q2        DA    P2
218           DA    Q3-1
219           ORG   $D560       ;old "RUN no matter what" routine
220 P2        BNE   PARSE
221 TOFIX     JSR   RNDB        ;round TO value before combining with sign
222           LDA   FACSGN
223           ORA   #$7F
224           RTS
      ********************************************
226           ORG   *-P2+Q2+4
227 Q3        DA    P3
228           DA    Q4-1
229           ORG   $F094
230 P3        BIT   DECPNT
231           BVS   J1          ;MULTIPLE "." IN NUMBER
232           JMP   EVAL
233 J1        JMP   IQERR       ;ILLEGAL QUANTITY error
      ********************************************
235           ORG   *-P3+Q3+4
236 Q4        DA    P4
237           DA    Q5-1
238           ORG   $D5AB       ;in Parsing routine
239 * Bxx *+2 is a NOP; used to keep checksum the same as original
240 P4        BCC   *+2         ;spaces in statements are significant
241           BCC   *+2         ;for parsing
      ********************************************
243           ORG   *-P4+Q4+4
244 Q5        DA    P5
245           DA    Q6-1
246           ORG   $D79C       ;in FOR statement
247 P5        JSR   TOFIX       ;fix TO statement
248           NOP
      ********************************************
250           ORG   *-P5+Q5+4
251 Q6        DA    P6
252           DA    Q7-1
253           ORG   $EC9A
254 P6        JMP   P3          ;Multiple "." in number
      ********************************************
256           ORG   *-P6+Q6+4
257 Q7        DA    P7
258           DA    Q8-1
259           ORG   $E6D3       ;MID$,LEFT$,RIGHT$
260 P7        BCC   *+2         ;with zero length now
261 *                           ;gives null string NOT error
      ********************************************
263           ORG   *-P7+Q7+4
264 Q8        DA    P8
265           DA    Q9-1
266           ORG   $D970
267 P8        HEX   86          ;See Bongers' Applesoft Error article
      ********************************************
269           ORG   *-P8+Q8+4
270 Q9        DA    P9          ;list error line (not a bug)
271           DA    Q10-1
272           ORG   $D439
```

I've seen suggest that you eliminate the various cassette-tape routines to get the needed space. This is not an option on the Apple IIc or Basis 108 computers, since the tape routines are already gone — the space has been used for lower-case, medium resolution graphics and a little 80-column support.

Another option is reassembling all of Applesoft, saving a few bytes here and there (perhaps by using 65C02 opcodes on the Apple IIc). However, this is unsafe, as Applesoft has been "frozen" for so long that many programs (such as some of the demonstration programs included with Merlin) assume that ROM Applesoft routines are at specific locations. We certainly should not move anything that looks like a useful entry point for a subroutine.

A lot of code that we can modify to save space is concentrated in the coldstart routine at $F128. This is not a useful subroutine, as it does not return to the calling program. The coldstart routine contains some examples of poorly written code, e.g. initializing $0001 twice. In addition, it contains a routine (at $F177-$F193) to determine RAM size dynamically (when Applesoft was written, the majority of Apples had less than 48K). It also initializes the old tape cassette software lock, which has become superfluous. In all, I saved 60 bytes by eliminating code from this routine.

At $F094, there are two unused floating-point constants, for a total of ten bytes of free space.

We need just a little more space for our patches. When Applesoft was written (back in the days of tape cassettes), a little-discussed feature was built into it to enable software writers to set a flag on page zero and prevent users from LISTing or otherwise modifying programs. Other forms of protection have superseded this one; thus it can be safely eliminated. By patching $D560 to BNE $D56C, we gain 11 bytes between $D562 and $D56B.

MY WISH LIST

In addition to fixing the bugs I've described, the program installs a few amenities that I wanted for my Apple IIc (I'd grown accustomed to them on my Basis 108). My "wish list" includes the following:

1. A CALL without any argument is interpreted as a CALL −151.
2. After printing an error message and "IN *nnnn*," the line that caused the error is listed.
3. String functions (RIGHT$, LEFT$, MID$) that evaluate to strings with length zero output the null string instead of an error message.
4. In parsing typed lines, spaces are considered significant as delimiters. For instance, "FOR I = S TO P" is no longer parsed as "FOR I = STOP". Spaces are *not* required, but if

LISTING 1: BANDAIDS Source Code *(continued)*

```
273  P9       JSR   PLNERR      ;print "in line" and list that line
     ********************************************
275              ORG   *-P9+Q9+4
276  Q10      DA    P10         ;CALL w/o address to Monitor (not a bug)
277              DA    Q11-1
278              ORG   $D018       ;in keyword vector table
279  P10      DFB   CALL-1      ;NEW VECTOR FOR "CALL"
     ********************************************
281              ORG   *-P10+Q10+4
282  Q11      DA    P11
283              DA    Q12-1
284              ORG   $E9B2       ;in MULT
285  P11      JMP   PMULT       ;WHERE THE MISSING SEC GOES
     ********************************************
287              ORG   *-P11+Q11+4
288  Q12      DA    P12
289              DA    Q13-1
290              ORG   $DB77       ;fix GET error message
291  P12      JMP   TYPERR      ;goto TYPE MISMATCH error
292              BRK
     ********************************************
294              ORG   *-P12+Q12+4
295  Q13      DA    P13
296              DA    Q14-1
297              ORG   $DA1C
298  P13      JSR   CHLINN      ;chk line no. vs. 6400
299              NOP
     ********************************************
301              ORG   *-P13+Q13+4
302  Q14      DA    P14
303              DA    Q15-1
304              ORG   $D7AC       ;part of TO fix
305  P14      JMP   PUSHFAC+3   ;SKIP RNDB;ALREADY DID IT!
     ********************************************
307              ORG   *-P14+Q14+4
308  Q15      DA    PMULT
309              DA    Q15A-1
310              ORG   $E0FE       ;reuse old -32768
311  PMULT    SEC               ;MULTIPLY--MISSING SEC
312              JMP   $E8DA
     ********************************************
314              ORG   *-PMULT+Q15+4
315  Q15A     DA    ONRGO
316              DA    Q16-1
317              ORG   $F2E3       ;statements after ONERR-GOTO bug--
318  ONRGO    JSR   $D9A3       ;Ignore to : or end of line
     ********************************************
320              ORG   *-ONRGO+Q15A+4
321  Q16      DA    P16
322              DA    Q17-1
323              ORG   $E112       ;in integer conversion code
324  P16      LDA   #M32768     ;new (5 byte) -32768 address
325              LDY   #>M32768
     ********************************************
327              ORG   *-P16+Q16+4
328  TEMP     =     MACHINE-$2E
333  BLAST2C  =     $FCCA       ;hole blaster
334  COLD2C   =     $FCDE
335  RESET_X  =     $FABD
336  BUTN1    =     $C062
337  BUTN0    =     $C061
338  RTS2D    =     $FB2E
339  BEEPFIX  =     $FAA3
340  OLDRESET =     $FF59
341  VMODE    =     $4FB
342  ZRAMSWCH =     $C073
343  Q17      DA    RESET_X
344              DA    Q18-1
345              ORG   RESET_X
346              LDA   #$FF
347              JSR   RESET_ZRAM
348              ASL   BUTN1
349              BIT   BUTN0
350              BPL   RTS2D
351              BCC   BEEPFIX
352              JMP   OLDRESET    ;OLD STYLE RESET TO MONITOR IF BOTH APPLES
     ********************************************
354              ORG   *-RESET_X+Q17+4
355  Q18      DA    BLAST2C
356              DA    Q19-1
357              XC                ;65C02 opcodes available on //c
358              ORG   BLAST2C
359              DEC   $3F4        ;kill power up flag
360              BRA   $FCDE       ;BRANCH AROUND HOLE BLASTER.
361  RESET_ZRAM STA VMODE
362              STZ   ZRAMSWCH
363              RTS
364              HEX   2F          ;GET CHECKSUM RIGHT
     ********************************************
366              ORG   *-BLAST2C+Q18+4
367  Q19      HEX   00
368  *
END OF LISTING 1
```

LISTING 2: BANDAIDS II Plus Version

```
8000- 2C 81 C0 2C 81 C0 A9 D0
8008- 85 3D A9 00 85 3C A8 B1
8010- 3C 91 3C C8 D0 F9 E6 3D
8018- D0 F5 A9 80 85 3D A9 52
8020- 85 3C A0 00 B1 3C F0 23
8028- 85 42 C8 B1 3C 85 43 C8
8030- B1 3C 85 3E C8 B1 3C 85
8038- 3F C8 98 18 65 3C 85 3C
8040- 90 02 E6 3D A0 00 20 2C
8048- FE B0 D7 2C 82 C0 2C 80
8050- C0 60 28 F1 02 81 A2 FF
8058- 86 76 86 73 86 6F A2 FB
8060- 9A 20 73 F2 A9 4C 85 00
8068- 85 03 85 90 85 0A A9 99
8070- A0 E1 85 0B 84 0C A2 1C
8078- BD 0A F1 95 B0 CA D0 F8
8080- 86 F2 86 A4 86 54 8A 8D
8088- 00 08 48 A9 03 85 8F 20
8090- FB DA A9 01 85 F1 8D FD
8098- 01 8D FC 01 85 67 A9 55
80A0- 85 52 A9 BF 85 74 85 70
80A8- A0 08 84 68 A5 67 20 E3
80B0- D3 20 4B D6 A9 3A A0 DB
80B8- 85 04 84 05 A9 3C A0 D4
80C0- 85 01 84 02 6C 01 00 20
80C8- 19 ED A5 76 A6 75 85 51
80D0- 86 50 20 1A D6 A2 FF 86
80D8- 76 E8 86 B8 A0 02 84 B9
80E0- 8A 91 B8 81 B8 4C DA D6
80E8- 90 80 00 00 00 C9 19 B0
80F0- 01 60 4C C9 DE 8D 8D 8D
80F8- 8D 8D 8D 8D 8D 99 4C 65
8100- FF F0 FB 60 D5 10 81 D0
8108- 0A 20 72 EB A5 A2 09 7F
8110- 60 94 F0 1E 81 24 9B 70
8118- 03 4C 61 EC 4C 99 E1 AB
8120- D5 26 81 90 00 90 00 9C
8128- D7 2E 81 20 62 D5 EA 9A
8130- EC 35 81 4C 94 F0 D3 E6
8138- 3B 81 90 00 70 D9 40 81
8140- 86 39 D4 47 81 20 99 F1
8148- 18 D0 4C 81 D2 B2 E9 53
8150- 81 4C FE E0 77 DB 5B 81
8158- 4C 76 DD 00 1C DA 63 81
8160- 20 BF F1 EA AC D7 6A 81
8168- 4C 23 DE FE E0 72 81 38
8170- 4C DA E8 E3 F2 79 81 20
8178- A3 D9 12 E1 81 81 A9 BA
8180- A0 F1 00
```

END OF LISTING 2

```
KEY PERFECT 5.0
RUN ON
BANDAIDS (II PLUS Version)
```

CODE-5.0	ADDR# - ADDR#	CODE-4.0
6224400B	8000 - 804F	23D7
3187715E	8050 - 809F	298A
9E19D45D	80A0 - 80EF	2685
160D8EEF	80F0 - 813F	267E
FF3A097E	8140 - 8182	2114
172516C3 = PROGRAM TOTAL =		0183

LISTING 3: Additional Code for IIc Version

```
8182- BD FA 97 81 A9 FF
8188- 20 CF FC 0E 62 C0 2C 61
8190- C0 10 64 90 D7 4C 59 FF
8198- CA FC A8 81 CE F4 03 80
81A0- 0F 8D FB 04 9C 73 C0 60
81A8- 2F 00
```

END OF LISTING 3

LISTING 4: Additional Code for IIe Version

```
8182- C8 C2 9B 81 CE F4
8188- 03 A0 0F A9 C5 20 A8 FC
8190- 2C 61 C0 10 0B 88 D0 F3
8198- 4C 59 FF 24 00
```

END OF LISTING 4

you include them the BASIC interpreter will recognize them as valid separators.

Several of the amenities took no extra space, but just involved substituting NOPs for existing code.

HOW APPLESOFT BANDAIDS WORKS

Now it's time to look at the patch program, BANDAIDS (Listing 1). It fixes most of the bugs described above (the exceptions are the slow garbage collection, the RND bug and the stack problem with ONERR-GOTO). The program first copies ROM into bank-switched RAM (often called the language card or RAM card). Then it installs a series of patches in high RAM; the individual patches are located beyond **line 115** and each patch is part of the data structure described in **lines 109-114**. The ORG pseudo-op is used in a somewhat unusual way in order to assemble the patches as though they were in their final resting place, rather than where they are initially loaded. Finally, the program enables reading from high RAM and starts the new BASIC.

I recommend using the program just as listed. It deals with several issues that are not dealt with in detail in this article. These issues include the following:

1. Some of the patches are interdependent (e.g., the old four-byte −32768 constant becomes unused when the −32768 bug is fixed; those four bytes are recycled to fix the multiplication routine bug).
2. The checksum for Applesoft should be correct, especially if you are putting the patches into ROM. This will allow your ROM to pass the diagnostic routines and may be needed to use some copy-protected programs.
3. Several patches are in several pieces. I have included a table that serves as an index to the patches for those wishing to modify the program. I strongly urge anyone altering the program to obtain a commented disassembly of Applesoft [2].

When it is installed in bank-switched RAM, Applesoft Bandaids works on any 64K or 128K Apple. With the enhanced ROMs for the IIe, the checksum may not be correct when the program is put into EPROM. The program needs modification to work on the Basis 108, but with a little work, the fixes will all fit into existing holes in the Basis FP80 program. These holes are a long series of zeros at $F7A7 and shorter series at $F3C9 and $F1D8.

Several of the amenities took no extra space, but just involved substituting NOPs for existing code.

Because I was trying to write patches that would work on the IIc and Basis 108 — both of which have already eliminated tape commands and reused most of the space — I did not construct the patches to fit into the space occupied by Applesoft's cassette tape routines. For some readers, however, this may be the preferred way of implementing either fast garbage collection or a random number generator.

PUTTING IT ALL INTO ROM

The main reason these changes were put into ROM is that I know I won't actually use any pre-boot consistently. If I don't fix the ROM, then I won't always use the patches; this is particularly distressing when the multiplication routine bug invisibly undermines the accuracy of the calculations.

If you use ProDOS, then changing the firmware seems to be the only way to implement the patches described in this article since the bank-switched RAM is occupied by ProDOS. The older Apple II and II Plus computers used an unusual chip-enable configuration [15], so using 2716 EPROMs on the motherboard involves some extra effort [16]. Apple IIe and IIc computers accept standard 2764 and 27128 EPROMs, respectively, so changing the firmware is easier than on earlier Apples.

I believe that if you own a ROM copy of Applesoft, then making a firmware copy (for your own use) that fixes the bugs that Apple Computer, Inc. neglects is fair use, and the copyright law allows fair use copying.

continued on next page

The real issue preventing alteration of ROMs, I think, is that some programs may do a checksum or other verification making sure that ROM is just as it should be (perhaps as part of a copy-protection scheme). Personally, I have made it a rule not to buy anything that is copy protected.

MODIFICATIONS

Although Applesoft Bandaids does not fix the garbage collection bug or the random number generator bug, references 7, 8, 12 and 13 can help you fix them yourself. One way to find some space for making patches is to eliminate one or more of the cassette tape keywords from Applesoft (in the Apple II Plus or IIe).

A brief note about how to reuse the space for the tape routines is in order, since it should be done in the same way as on the Apple IIc. Leave the keyword in place, intact in the table at SD0D0. Replace the action address for the keyword with the location of the ampersand (&) vector. For example, to eliminate SHLOAD, the patch is D034:F4 03. F4 (not F5) is correct because the address is pushed onto the stack and control is actually transferred with an RTS.

The action addresses for the tape commands are stored as follows:

SHLOAD	D034	RECALL	D04E
LOAD	D06C	SAVE	D06E
STORE	D050		

If you are interested in implementing Bongers' garbage collection routine, consider whether it correctly handles garbage collection that is triggered by a temporary string — a string that doesn't have an associated variable name. The algorithm as published apparently makes no provision for this. Since Bongers put the garbage collector outside Applesoft, he did not need to handle this case; it can only arise in the middle of an Applesoft statement. However, it is essential to do so if a new garbage collection routine is implemented from within Applesoft, unless you are somehow able to guarantee that you always trigger garbage collection with FRE( ).

Though the IIc's Applesoft ROM has been upgraded, no one fixed the bugs. I hope this article helps stimulate interest in getting these bugs out of future ROMs.

REFERENCES

1. Bongers, Cornelis. "Applesoft Error," *All About Applesoft*, Apple PugetSound Program Library Exchange, 1981, p. 100.
2. Bredon, Glen. *Merlin*, Roger Wagner Publishing, 1983.
3. *Applesoft //*, Apple product no. A2L0006, Apple Computer, Inc., 1978, p. 137. (This has since been replaced by a new, more expensive manual.)
4. Ibid, pp. 30, 31.
5. Floeter, Alan, and Valerie Floeter. "Applesoft FOR-NEXT Problem," *Nibble*, Vol. 5/ No. 12, Dec 1984, p. 109.
6. Goez, Eric. "Real Variable Study," *Call-A.P.P.L.E.*, Jan. 1981, p. 8.
7. Sander-Cederlof, Bob. "A Problem With the VAL(A$) Function," *Call-A.P.P.L.E.*, Nov.-Dec. 1980, p. 65.
8. Sather, Jim. *Understanding the Apple //*, Quality Software, 1983, p. 5-36.
9. Sparks, David. "RND Is Fatally Flawed," *Call-A.P.P.L.E.*, Jan. 1983, p. 29.
10. Hare, Tom, John Russ and Gary Faulkner. "A New Pseudo-Random Number Generator," *Call-A.P.P.L.E.*, Jan. 1983, p. 33.
11. Ruth, Clay. "Garbagemen Strike: Selective String Preservation," *Call-A.P.P.L.E.*, Aug. 1982, p. 9.
12. Bongers, Cornelis. "Straightforward Garbage Collection," *Micro*, 51:90, August 1982.
13. Wiggington, Randy. "Fast Garbage Collection," *Call-A.P.P.L.E.*, Jan. 1981, p. 40.
14. Basham, Bill. *Diversi-DOS Version 4.1C*, Diversified Software Research, 1985.
15. Lancaster, Don. *Micro Cookbook Volume 1*, Howard W. Sams & Co., Inc., 1982, p. 351.
16. Sather, Jim. Op cit., p. 6-15.