Software Debugging for Beginners

Sometimes it must seem as though every effort in programming is one big error message. Aside from reading the supplied documentation, here are some useful tips that you can follow for avoiding those messages.

John Leslie
10 Souhegan St.
Milford NH 03055

If it hasn't happened to you yet, it has surely happened to someone in your computer club: After having sent your check for a $15 program, you received the program, loaded it, and it didn't work. Then, of course, you called the people who sold you the program to tell them the sad story. In general, people selling software to computer hobbyists do try to be helpful.

Often, though, you can have serious difficulty finding someone familiar enough with the program to help you. And usually, you end up waiting several weeks for a replacement cassette. Most people find this frustrating. Why can't you do something in the meantime?

Well, I am about to tell you that you can. You might get scared if I told you to debug the program, so, instead, I shall suggest simply that you play around with it. But in truth, you will be doing much the same things that a professional programmer does when he debugs.

Study the Assembly Listing

To start, you should read through the assembly listing, with both source code and object code. Usually this listing is supplied with the program. Oops, I think I lost a few of you there.

An *assembly listing* is simply a listing (printout) produced by an assembler. *Source code* is what the programmer actually wrote, and is used as input to the assembler. *Object code* is a representation (usually in hexadecimal) of the actual bit patterns the computer will use as its instructions.

An *assembler* is a program or process that puts together a machine-language program, given a *source program* specifying each machine-language instruction in detail. Many of the programs supplied to hobbyists have been *hand assembled,* meaning simply that the programmer, rather than the computer, assembled the program. It's of no consequence. He will have produced the same sort of listing.

Don't worry about understanding the listing in detail; don't even worry about understanding it in general. Your task is just to gain some familiarity with it.

First, Read the Comments

What do you look for? Look at the pictures. Sad to say, there usually aren't any. Next, look at the comments. Assembly code has the following format:

Label / Op code / Operands / Comments

Comments are off to the right, and are the only things that remotely resemble English. If they are written well, comments will tell you *why* something is done, rather than *what* is done. And at this stage, you aren't much concerned with *what* is being done.

The sample assembly listing (Fig. 1) is an actual portion of a program of mine. It performs a generation change in the game of Life. Reading the comments, you should get the idea that the program tests something about eight neighbors, and sets a cell's new state on the basis of its previous state and the number of its neighbors. That is enough to grasp for now.

Then, the Jump Op Codes

Next, look at the op code column. You should have a manual explaining the various op codes for your computer. You may have tried reading it, and given up. But don't worry—it is not meant to be read; it is a reference book. So now, get it out and *refer* to it. You want to know which of those alphabet-soup concoctions called op codes are supposed to cause the computer to jump to a new location. Memorize them by name. Don't worry too much about what they do.

In Fig. 2, I have outlined the jump op codes for the 8080, 6800 and 6502 *micro*computers. This should enable you to get through the example of Fig. 1 if you don't have a 6502.

The point of this is that all the other instructions will execute without a break in sequence, so you can safely ignore them; whatever they may do, the computer will proceed to the next instruction in sequence.

The jump op codes, on the other hand, can bring untold

confusion. They can cause machine instructions to be executed many times, or not at all.

Outline the Program Flow

So your next step is to scan the op code column for these jump op codes. This longer exercise is going to give you a picture of the *program flow.*

In the example of Fig. 1, you will find eleven uses of JSR, three uses of BNE and one each of BVC, BVS and RTS. You should assume that all subroutines called will usually return to the next instruction. (This is not necessarily true, of course; sometimes it is spectacularly false. But you should assume it anyway.)

So ignore all those JSRs. The three BNEs turn out to simply skip the next instruction, so you can ignore the first two. The third is a conditional skip of the return from subroutine instruction, so you will have to watch it.

The BVC and BVS instructions are both conditional jumps, and if you check the op code table of Fig. 2, you will see that they are opposite conditions. Thus, these two instructions together amount to an unconditional jump back to near the beginning of the example. (Using two conditional jumps like this is common in 6502 code, since there is no relative-addressed unconditional jump instruction.)

You now have a picture of the program flow . . . straight through until near the bottom, where there is a conditional skip followed by a return from subroutine instruction (this amounts to a conditional return instruction). If the return is not done, then most of the code is repeated.

A few inferences are in order. If we execute a return from subroutine instruction, this whole thing must have been called as a subroutine. (True.) And eventually the conditional skip must fail, and the return will be executed. (Also true.)

A Word about Operands

Now, I suppose I should confess to sneaking a few things by while you weren't looking. I promised that you would be looking at the op codes column, but I went right ahead and checked the next column (operands) whenever I felt like it. And since I haven't told you anything about that column, you feel justifiably confused.

I approach this section with trepidation, fearing that after I have told you about the operands column you will be even more confused. The plain fact is that this column is used for all the leftovers. The theory is that "obviously" some additional information is needed to complete the machine instruction, and it is "simply" placed in the operands column.

The experienced programmer never gives this a second thought. Depending on his background, he may think of the items in that column as addresses or data. The fact remains, however, that they are leftovers, and no consistent rule can be made to apply. If you attack this slowly, however, and memorize what is put in that column for each of several kinds of op codes, you will soon enough master it sufficiently.

For right now, you only need to know about the jump op codes. For these, obviously the needed information is where to jump. So, keeping in mind that whatever you see must be a description of where to jump, look over the entries in the operands column following the jump op codes. You will see BEGIN, TEST, SWAP, GLOOP and $+2. "What's this $+2 bit?" you may well ask.

Permit me to digress a moment. If somebody asks for directions, you probably will use your hand to point. "Over there," you say, but your pointing is what conveyed the information. Computers are not too good at reading hand motions. The computer requires exact latitude and longitude, so to speak. People, on the other hand, much prefer to say, "over there," and point. Clearly, some compromise needs to be reached. Several methods have been tried over the years, with two still being used frequently.

The first is to write notes to the computer saying, in effect, "This is the place I call home; this is the place I call school; and this is the place I call work." Later on, the computer, being suitably programmed, will know what to do when you tell it to go to school.

The second method saves time when you don't feel like writing a note giving someplace a name. It is the equivalent of saying, "second door on your right." $+2 is analogous to second door on your right. Be warned that this kind of notation is not the same for all assembly languages. Some use a dollar sign; some use an asterisk; some use a number sign; and goodness only knows how many different characters have been used for the function.

In every case, however, this special character refers to the current location. But, sad to say, not every assembler means the same thing by current location. Some mean the address of the machine instruction being assembled; some mean the contents of the internal program counter register when the instruction is executed (which usually points to the instruction after it).

And, as if that weren't bad enough, not all assemblers count the same thing when determining what +2 means. Some count instructions; some count bytes; assemblers for larger computers may even count words or half-words. But at least the plus vs minus directions are standard. Plus refers to higher addresses (instructions later in sequence) and minus refers to lower addresses (instructions earlier in sequence).

"How," you may ask, "do I tell what it means?" Well, if you're lucky, you may not have to. If you see $+2 and neither of the next two instructions is a jump, you can ignore the problem for a while. When you finally are forced to find out for sure, I recommend running the computer in single-step mode and finding out what *it* thinks. That, after all, is the only opinion that

```
Label                          Op Code  Operands    Comments
0328  20 1C 02  GENERATION     JSR      BEGIN
   B  A6 D3     GLOOP          LDX      CURRX       Preset to XMIN, YMIN
   D  A4 D4                    LDY      CURRY       X coordinate
   F  A9 00                    LDA      #0          Y coordinate
  31  20 14 02                JSR      TEST
   4  85 D2                   STA      CURR        1 iff occupied
   6  A9 00                   LDA      #0          Preset no neighbors
   8  E8                      INX                  1st neighbor
   9  20 14 02                JSR      TEST        Count if occupied
   C  C8                      INY                  2nd
   D  20 14 02                JSR      TEST
  40  CA                      DEX                  3rd
   1  20 14 02                JSR      TEST
   4  CA                      DEX                  4th
   5  20 14 02                JSR      TEST
   8  88                      DEY                  5th
   9  20 14 02                JSR      TEST
   C  88                      DEY                  6th
   D  20 14 02                JSR      TEST
  50  E8                      INX                  7th
   1  20 14 02                JSR      TEST
   4  E8                      INX                  8th
   5  20 14 02                JSR      TEST
   8  A0 00                   LDY      #0          Preset dead
   A  C9 02                   CMP      #2          If two neighbors
   C  D0 02                   BNE      $+2
   E  A4 D2                   LDY      CURR        Survives
  60  C9 03                   CMP      #3          If three
   2  D0 02                   BNE      $+2
   4  A0 01                   LDY      #1          Grows
   6  20 44 02                JSR      SWAP        Work on new board
   9  98                      TYA                  New state
   A  A6 D3                   LDX      CURRX
   C  A4 D4                   LDY      CURRY
   E  20 18 02                JSR      SET         Store new state
  71  20 20 02                JSR      STEP        Advance to next cell
   4  D0 01                   BNE      $+2
   6  60                      RTS                  Finished board
   7  20 44 02                JSR      SWAP        Back to old board
   A  50 AF                   BVC      GLOOP
   C  70 AD                   BVS      GLOOP
```

Fig. 1. Sample assembly listing.

is of any significance.

One of those manuals you received with your computer will, no doubt, tell you what the standard is, according to the manufacturer. That is the standard you should follow; and for any code run through the assembler, you know it is the convention the assembler followed.

But, as I mentioned before, many programs supplied to computer hobbyists have been hand assembled, and you *don't* know what convention that programmer may have used. Programmers have a tendency to use whatever scheme they were brought up with, until forced to change. Until you are quite certain which scheme the programmer used, don't assume anything. And, in case of doubt, check what the computer thinks.

On my way back to the subject, let me explain the first scheme—writing notes. The label field, starting with the first character of each line, is reserved for notes assigning names to places. On the second line of Fig. 1, you see a note: GLOOP. If there is anything in the label field, that name is assigned to that program location. Thus, the name GLOOP is assigned to program location 032B. When, near the end, you see BVC GLOOP, that causes a conditional jump to 032B to be assembled.

Now we can get back to business. I have told you about the operands for jump op codes, and we can get back to analyzing program flow. "Why," you may well ask, "make such a fuss about program flow?" The answer is quite simple, really. It makes no difference whether code is correct if it is not being executed . . . and, I might add, being executed the right number of times.

Set Breakpoints

Consequently, before charging off to check what a piece of code does, you should set a breakpoint to make sure it is being called. Very often, it isn't.

Some computer systems have powerful and easy-to-use methods for setting breakpoints. But on your microcomputer, it is probably pretty cumbersome. If someone knowledgeable is nearby, by all means ask him what the easiest way is. Most likely, it will involve overwriting the location where you want a breakpoint with an instruction to cause a software interrupt (BRK for the 6502, SWI for the 6800, or RST for the 8080), and assigning a monitor routine to field the interrupt.

To resume after the breakpoint, you will probably have to restore the instruction you overwrote, reset the program counter there and possibly fix the stack. If this sounds like a lot of work, you now understand why you keep seeing articles about better monitor systems. Nonetheless, I assure you, it is worth the effort.

Having a picture of the program flow, and knowing how to set breakpoints, you can now start setting breakpoints all over the place to prove whether the program is actually being executed according to your picture of the program flow. Surprisingly often, something has gotten garbled along the way, and the computer turns out to be jumping into some strange area. If you find such a case, it is usually easy to fix.

Your first hint of this is usually that the computer never reaches a breakpoint that you have set. Then proceed to set breakpoints gradually earlier (restarting the program each time), until the computer does stop. By this method, you can pinpoint where the program goes astray. Comparing the machine-language instruction (in memory) to the assembly code will quickly show any case of garbling.

This Program Will Self-destruct . . .

You should be warned that sometimes when a program goes haywire, it overwrites itself. You should always reload the program from cassette (or whatever medium you use) before setting a new breakpoint and restarting. However, this is *so* much of a nuisance that experienced programmers seldom do it. Nonetheless, when you're correcting a garbled instruction, it is worth the effort to reload and check whether it was garbled as loaded. If not, you haven't found the problem yet.

If you find a case in which the program is clobbering itself, you should set breakpoints progressively earlier, checking each time to see whether it has clobbered itself yet. For this case, of course, it is necessary to fix the clobbered code, usually by reloading from cassette before each restart.

Each time you find and fix a bug, you should feel free to remove all breakpoints to see if the program as a whole now works. If, on the other hand, you debug a newly written program, you should pause and spend a few minutes looking for similar mistakes. The human mind, once it has made a mistake, tends to make it again. It may even be worth your while to scan for similar mistakes when debugging a program that once worked.

Examine Variables

After you have proven that the program is being executed according to your understanding of program flow, it makes some sense to look at the program variables to see if they contain reasonable values at the strategic points during the computation.

As an example, it is often helpful to check the value of index variables at the beginning and end of iterative loops to see whether the loop is being done the right number of times. In the example of Fig. 1, CURRX and CURRY are index variables that represent the X and Y coordinates. Thus, you might reasonably expect them to range from 1 to 40 and from 1 to 24. Typically, at the end of iteration, one index variable will be

6800	6502	8080	Jump type
*BRA	JMP	JMP	Unconditional
JMP	---	---	Indexed
BNE	BNE	*JNZ	If not equal
BEQ	BEQ	*JZ	If equal
BCC	BCC	*JNC	If carry clear
BCS	BCS	*JC	If carry set
BPL	BPL	*JP	If plus
BMI	BMI	*JM	If minus
BGE	---	---	If greater than or equal
BGT	---	---	If greater than
BHI	---	---	If higher
BLE	---	---	If less than or equal
BLE	---	---	If lower or same
BLT	---	---	If less than
BVC	BVC	---	If overflow clear
BVS	BVS	---	If overflow set
---	---	JPO	If parity odd
---	---	JPE	If parity even
---	---	PCHL	Load PC from H,L pair
*JSR	JSR	CALL	Subroutine jump
---	---	CNZ	Conditional subr jumps
---	---	CZ	
---	---	CNC	
---	---	CC	
---	---	CP	
---	---	CM	
---	---	CPO	
---	---	CPE	
**SWI	BRK	**RST	Software interrupt
**WAI	---	**HLT	Wait for interrupt / Halt
RTI	---	---	Return from interrupt
RTS	RTS	RET	Return from subroutine
---	---	RNZ	Conditional returns
---	---	RZ	
---	---	RNC	
---	---	RC	
---	---	RP	
---	---	RM	
---	---	RPO	
---	---	RPE	

Fig. 2. Op codes on the same line are reasonably compatible. A single asterisk indicates a different addressing mode. A double asterisk indicates gross detail incompatibility (but the functions are similar).

at the final value or one past it. The rest should all be at final value or at initial value. If not, you have grounds for suspicion.

If you become suspicious, you can execute the loop exhaustively and count the number of times it is done. If, on the other hand, an index variable gets an obviously wrong value, you should suspect it is being clobbered and proceed to test where it is being clobbered.

It is also instructive to examine data areas during sections of code that are not supposed to change them, to ensure that they are, in fact, not being changed. If they are being changed, you can use the standard procedure to zero in on where the changes are occurring.

Data Structures

If you are particularly lucky and the program was well designed, there will be a subroutine you can call to display the status of data areas. When writing your own programs, you should be sure to include such a subroutine, preferably in a form that changes *nothing*, so that it may be called between any two instructions during the debugging phase.

If you are that lucky, you can now run through the section of code that is supposed to modify the data, setting breakpoints at convenient locations, and examine the data as changes are being made. Using this feature, you can often pinpoint the trouble area, still without having to know *what* the code is doing. But more likely, you won't be that lucky, and you will have to set out to learn about the data structure. And that, I fear, must wait for another article.

Summary

In debugging, you should always first establish which code is being executed. Then check to see that loops are being done the right number of times. After the program flow is proven correct, check that variables contain reasonable values. Only after you have localized a problem do you set out to understand what the code is doing.

Postscript—BASIC

To debug BASIC programs, you usually insert PRINT statements. In keeping with the debugging principles listed above, your first task is to establish program flow. So insert the simplest possible PRINT statements, using them like breakpoints. After program flow is established, then you should switch to PRINTing the values of the data. Printing your data before establishing program flow leads to much head-scratching and little progress. ■