DRAWING BOARD

Unlocking the secret software passage.

ROBERT GROSSBLATT

The shareware debuggers described here last month are posted on the *Electronics Now BBS* (516-293-2283, 1200/2400, 8N1). Although they are nice pieces of software, we've received many requests for information on commercial debuggers.

There's certainly no shortage of commercial debuggers on the shelves of your local software store, but Codeview and Periscope are the only ones I've really spent any time with. The current generation of debuggers are all what the companies who make them like to refer to as "feature rich." That's both good news and bad news.

The good news is that a commercial debugger will have the muscle and subtlety to do whatever you need it to. The bad news is that the more powerful the software, the steeper the learning curve. For that reason alone, it's a good idea to pick your debugger carefully. Because you'll have to make an enormous investment in time to become proficient in using the software, make sure the debugger you choose is the one that's going to do the job you have in mind.

I've been using Periscope for years, and I have never regretted the time and money I invested in it. The memory overhead it requires is minute and, at a basic price of about $300, it's a good deal. Some people prefer Codeview. It's a good product as well, but one of the major strengths of any software is the level of support the buyer can get from the manufacturer. In my experience it's a lot easier to get a person on the phone when you call the Periscope Company than it is when you call Microsoft to get help with Codeview.

If you call Periscope, you'll find that they have a wide range of products available ranging from a software-only package at about $300 to hardware/software combinations that are a lot more powerful—but for a lot more money. If you're in the business of writing software, you'll find that Periscope will earn its keep the first time you use it to figure out why your code keeps crashing. You can get in touch with the Periscope Company at 1475 Peachtree Street, Suite 100, Atlanta, GA 30309 (404) 888-5335. But back to the business at hand.

LISTING 1

```
2BC0:1900 55          PUSH BP          ;Save the register
2BC0:1901 50          PUSH AX          ;Save the register
2BC0:1902 9AB3194F55  CALL 554F:19B3   ;Show the company logo
2BC0:1907 9AC6304F55  CALL 554F:30C6   ;Play some music
2BC0:190C 9A965D4F55  CALL 554F:5D96   ;Do a doc check
2BC0:1911 9A3ADB4F55  CALL 554F:DB3A   ;Go to the game
```

If you've been doing your homework over the last month, you've gained some experience tracing through the kind of code you see when you do a dump in DEBUG. The pages and pages of disassembly seem to be meaningless at first but, if you stick to it, you'll find that there's an underlying rhythm to it all.

Most programs start off with a bunch of setup instructions followed by either a jump table or a sequence of CALLs for displaying the company logo on screen, starting the music, and so on. With any luck, the CALLs would be in the same order as the list of events you wrote down. If that's the case, you will sooner or later come across code that looks something like Listing 1, without the comments, of course. The addresses and details of each instruction would undoubtedly be different.

As discussed last month, you should break into the debugger after the company logo appears on the screen, and trace back until you find the CALL that executes that part of the code. If the CALL is followed by a sequence of other CALLs (as shown in Listing 1), just try each of the CALLs to find the one you want.

With your fingers crossed, call up the JUMP instruction of our debugger and execute the CALL at offset 1907. After a few tense milliseconds, the catchy jingle from the game might come blasting from the speaker. After treasuring a victorious moment of earned pleasure, get back into the debugger and go back to the table you found.

Executing the CALL at offset 190C will take you, just as expected, to the screen that's used for the document check—and there's the cursor just waiting for you to look up a number from a table in the manual and enter it. "Look this up!" you say with a smirk on your face, and with supreme confidence you Jump to the CALL at offset 1911. The screen clears, and the opening screen from the game appears with its usual message telling you to "Press Any Key to Continue."

You press the Return key and your jaw drops; instead of getting into the game, you see a familiar message saying that "That's Not the Code Number I Want" and you're unceremoniously dumped back into DOS. What's going on?

While the practical result of this message is that you're going to be up all night, the reason for its appearance is that there's more going

on in the document-check routine than you thought when you first found it. Obviously, the program is looking for something that was supposed to happen during the routine. Since your way of dealing with the copy protection was to bypass it altogether, what was supposed to happen there was bypassed by the routine as well.

When you run across situations like this, the usual reason is that a successful completion of the document check (reading the correct number from the correct table on the correct page in the manual and entering it correctly at the keyboard) sets a flag somewhere in memory that tells the program to go into the game. When the document check is bypassed, the flag isn't set, and instead of going on to save the earth from an invasion by politically correct aliens, you're staring at a DOS prompt.

To figure things out, you'll have to trace through the document-check routine whose starting address you found in the jump table. This time around, you'll have to look for different code than you did when you were searching for CALLs. As you trace into the document-check routine, keep an eye out for CALLs as you did earlier, but also watch out for code that looks like that shown in Listing 2.

Whenever you run across a CALL, write down the address, execute the CALL, and observe what happens. More than likely you'll run across routines that do things such as pausing for keyboard input, storing an entered string in a memory buffer, or calculating checksums. These are all part of the larger document-check routine that collects and checks the validity of what you type at the keyboard. But no matter how fancy the document check is, its only job is to give a go/no-go indication to the rest of the program. And in most cases, that indication is marked by having a particular bunch of bytes stored in a particular memory location.

The line of code in Listing 2 is important because it's where the successful completion of the document check is marked. Instructions such as the one shown in Listing 2 will stand out like a sore thumb if they're preceded by stuff like that shown in Listing 3.

Notice the series of instructions that come before the MOV instruction at offset 62BA. A lot of things have to happen successfully for that instruction to be executed at all. Storing a 1111 at location 1C32h is the key to the siccessful completion of the document check. All the code in the routine is aimed at the rest of the game seeing a 1111 there, (meaning that correct code was entered from the table in the manual) or something else at 1C32h (meaning that incorrect code was entered and the user failed the document check).

Being able to spot this kind of code is partially due to luck, partially due to experience, and often results from an educated guess. There will always be lots of false tries before you find the one (or more than one) solution that works. In general, keep an eye out for lines of code that put hard numbers in absolute locations low down in the memory map. Most games are written in high-level languages such as C, and it's standard programming practice to declare variables early on in the code. That's what was done in our example here.

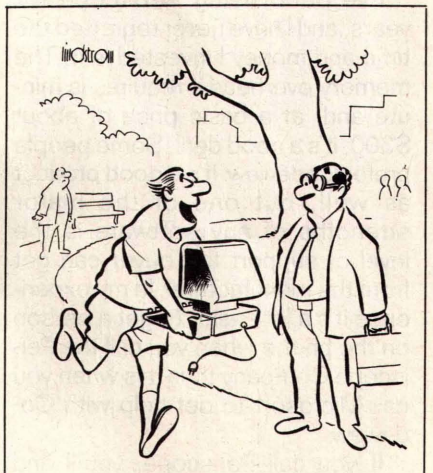
Once you find the key, it's a simple matter to check and see if it works. Go back into the debugger, manually stuff a 1111 into location 1C32h, and execute the CALL at offset 1911. If the screen clears and you see a shipload of aliens speeding to the earth, you can hit the sack and get some rest, secure in the knowledge that the hardest part of defeating the document check is behind you. Next time we'll see how to make it permanent. Ω

LISTING 2

```
4F1C:62BA C706321C1111          MOV WORD PTR [1C32],1111
```

LISTING 3

```
4F1C:6290 3B46E0          CMP AX,[BP-20]
4F1C:6293 7505            JNZ 629A
4F1C:6295 C746DEFFFF      MOV WORD PTR [BP-22],FFFF
4F1C:629A EB00            JMP 629C
4F1C:629C 837EDE00        CMP WORD PTR [BP-22],+00
4F1C:62A0 7402            JZ 62A4
4F1C:62A2 EB0C            JMP 62B0
4F1C:62A4 FF46D6          INC WORD PTR [BP-2A]
4F1C:62A7 837ED602        CMP WORD PTR [BP-2A],+02
4F1C:62AB 7D03            JGE 62B0
4F1C:62AD E937FD          JMP 5FE7
4F1C:62B0 837EDE00        CMP WORD PTR [BP-22],+00
4F1C:62B4 7504            JNZ 62BA
4F1C:62B6 0E              PUSH CS
4F1C:62B7 E8E8FC          CALL 62A2
4F1C:62BA C706321C1111    MOV WORD PTR [1C32],1111
```



"Pardon me. Just jogging my memory."

DRAWING BOARD

We're finally free to play our game without performing the document check!

ROBERT GROSSBLATT

If you've been following this series on software debugging (it's been discussed in the July, September, and November issues of **Electronics Now**), you now know the basic steps to getting rid of a document check. The job involves locating the relevant code and getting rid of it. Although that sounds simple enough, finding the code can be a difficult procedure—especially when there aren't any roadsigns to follow.

The pages and pages of code generated by raw debug dumps is, without question, some of the most stultifying and mind-numbing text you'll ever read. But after a bit of experience, the relevant portions will jump off the screen at you—even though the people who write document-check or copy-protection routines often go out of their way to make it as difficult to find and follow as possible. But more about that later. Let's first get back to finishing off our example.

Recall the steps taken so far. First, a list was made of exactly what happens when the game is played, and the order in which those things happen. We also noted what happens when a document check is answered correctly and what happens when an incorrect answer is given. Then the CALLs that initiated those routines were found and carefully examined for any clues as to the operation of the document check. In our example I've created a JUMP table that ran one part of the game's introduction after another. To refresh your memory, the disassembly listing for the code we're working on is shown in Listing 1. While things probably won't be quite as neat as Listing 1 in an actual program, the basic idea will be the same.

Simply eliminating the document check is usually a bad idea. Putting a bunch of NOPs (no operation) in place of the CALL to the document check will certainly bypass the routine but, as we saw last month, it can also cause the game to crash. That's because when the document check is bypassed, anything that it was supposed to do—such as passing data to the main program—is bypassed as well. If the data is found by the main program, the game gets started. If not, you're out of luck.

LISTING 1

```
2BC0:1902 9AB3194F55   CALL 554F:19B3    ;Show the company logo
2BC0:1907 9AC6304F55   CALL 554F:30C6    ;Play some music
2BC0:190C 9A965D4F55   CALL 554F:5D96    ;Do a document check
2BC0:1911 9A3ADB4F55   CALL 554F:DB3A    ;Go to the game
```

Tracing into the document check last month revealed that entering the correct code word caused the routine to set a flag, which allowed the game to continue normally. In the disassembled code shown in Listing 2, the key instruction is the last one before the return. If a 1111h is put in location 1C32h, the game can be started by executing the call that starts the game. What we have to do is figure out how to make that happen automatically.

As with most things, there are lots of ways to do the job. In our example, the most straightforward way to do it is to take the instruction at 554F:62BA and substitute it for the CALL found at 2BC0:190C—the one that calls the document check in the first place. By doing that, the program will set the flag correctly and then go on to execute the game. The only problem with this is that we don't have enough room in the jump table, because the MOV instruction is six bytes long and the CALL is only five bytes.

That's not much of a problem because we actually have a huge area of unnecessary code that we can overwrite—the document-check code itself. Once we finish patching the code so that it will bypass the document check completely, none of the code there will ever be executed. That is, after all, the reason behind this whole exercise. The document check starts at location 554F:5D96 (see the third line in Listing 1). That's the address where the instruction to set the flag and return to the jump table should be patched. The actual code we need

LISTING 2

```
554F:629C 837EDE00       CMP WORD PTR [BP-22],+00
554F:62A0 7402           JZ 62A4
554F:62A2 EB0C           JMP 62B0
554F:62A4 FF46D6         INC WORD PTR [BP-2A]
554F:62A7 837ED602       CMP WORD PTR [BP-2A],+02
554F:62AB 7D03           JGE 62B0
554F:62AD E937FD         JMP 5FE7
554F:62B0 837EDE00       CMP WORD PTR [BP-22],+00
554F:62B4 7504           JNZ 62BA
554F:62B6 0E             PUSH CS
554F:62B7 E8E8FC         CALL 62A2
554F:62BA C706321C1111   MOV WORD PTR [1C32],1111
554F:62C0 CB             RET
```

is the last two instructions found at the very end of the document check itself, that is, the last two lines in Listing 2. The program will execute normally, go to the document check, set the flag correctly, and execute the game.

This is easy enough to try while still inside the debugger. All we have to do is edit the instructions at the beginning of the document check and replace whatever we find there with those two instructions. As soon as that's done, the debugger's JUMP command can execute the start of the entire game at the top of the original jump table.

Making this change permanent is a piece of cake. All you need is a hexadecimal editor, a bunch of hex bytes, and a search string. The first is a matter of preference and convenience, the second is the hex equivalent of the last two instructions in the document check, and the third is nothing more than the original hex code found at the beginning of the document check.

Use your hex editor's search command to find the beginning of the document check, patch in the seven replacement bytes, and write the file back to disk with those changes. Once you do that, you should be able to go back to the command prompt, enter the name of the EXE file, and run the game. Everything will be as it was except the document check will be gone.

The example presented here is much easier than what you're likely to encounter in a real program. Since the job of any copy-protection routine is, as the name would imply, to protect against copying, the code that does the work is often hidden and hard to find. You're undoubtedly going to run across techniques such as self-modifying code and hidden checks that can make a complete crack of the program more than you bargained for. In our example, the crack could have been accomplished by something as simple as changing the address in the CALL to the document check. Instead of adding a patch to the beginning of the document check, we could have simply CALLed the end of the check where the flag was set by the original code. In that case, the jump table would have been changed to look like Listing 3, and the result would have been the same as it was when we patched the beginning of the document-check code.

LISTING 3

```
2BC0:1902 9AB3194F55   CALL 554F:19B3  ;Show the company logo
2BC0:1907 9AC6304F55   CALL 554F:30C6  ;Play some music
2BC0:190C 9ABA624F55   CALL 554F:62BA  ;Do a document check (HA!)
2BC0:1911 9A3ADB4F55   CALL 554F:DB3A  ;Go to the game
```

LISTING 4

```
1BCE:01D2 8A0E9000      MOV   CL,[0090]
1BCE:01D6 E8BEFF        CALL  0197
1BCE:01D9 380E9000      CMP   [0090],CL
1BCE:01DD 74F3          JZ    01D2
1BCE:01DF A08000        MOV   AL,[0080]
1BCE:01E2 FEC0          INC   AL
1BCE:01E4 3C0A          CMP   AL,0A
1BCE:01E6 7405          JZ    01ED
1BCE:01E8 A28000        MOV   [0080],AL
1BCE:01EB EBE5          JMP   01D2
1BCE:01ED C606800000    MOV   BYTE PTR [0080],00
1BCE:01F2 A08100        MOV   AL,[0081]
1BCE:01F5 FEC0          INC   AL
1BCE:01F7 3C0A          CMP   AL,0A
1BCE:01F9 7405          JZ    0200
1BCE:01FB A28100        MOV   [0081],AL
1BCE:01FE EBD2          JMP   01D2
1BCE:0200 C606810000    MOV   BYTE PTR [0081],00
1BCE:0205 A08200        MOV   AL,[0082]
```

LISTING 5

```
2A23:011F 52            PUSH  DX
2A23:0120 BACE1B        MOV   DX,1BCE
2A23:0122 C706D201B39A  MOV   WORD PTR [01D2],9AB3
2A23:0128 C706D4014F19  MOV   WORD PTR [01D4],194F
2A23:012E C706D6019A55  MOV   WORD PTR [01D6],559A
2A23:0134 C706D80130C6  MOV   WORD PTR [01D8],C630
2A23:013A C706DA01554F  MOV   WORD PTR [01DA],4F55
2A23:0140 C706DC01BA9A  MOV   WORD PTR [01DC],9ABA
2A23:0146 C706DE014F62  MOV   WORD PTR [01DE],624F
2A23:014C C706E0019A55  MOV   WORD PTR [01E0],559A
2A23:0152 C706E201DB3A  MOV   WORD PTR [01E2],3ADB
2A23:0158 C706E401554F  MOV   WORD PTR [01E4],4F55
2A23:015E C706E6010E5A  MOV   WORD PTR [01E6],5AB8
2A23:0164 C706E8010700  MOV   WORD PTR [01E8],0007
2A23:016A B000          MOV   AL,00
```

The bottom line in defeating copy-protection schemes is that you have to work under the assumption that nothing should be taken at face value. If you've plodded through the code tracing every CALL and haven't found the one that brings up the document-check screen, there's a good chance that the actual CALL is hidden in some self-modifying code. That can be a real pain in the neck to find because the code being modified can appear to be anything at all. It can look perfectly legitimate and even the most astute crackist can be fooled.

As an example, what's wrong with the code in Listing 4? The answer is that there's nothing wrong with it; it's part of a routine that I wrote some years ago for an EPROM in a microprocessor-controlled lighting system. If you came across this in a game you wouldn't give it a second glance.

Taking this a step further, suppose you ran across code that

LISTING 6

```
1BCE:01D2 9AB3194F55   CALL   554F:19B3   ;Show the company logo
1BCE:01D7 9AC6304F55   CALL   554F:30C6   ;Play some music
1BCE:01DC 9ABA624F55   CALL   554F:62BA   ;Do a document check
1BCE:01E1 9A3ADB4F55   CALL   554F:DB3A   ;Go to the game
1BCE:01E6 5A           POP    DX          ;More code
1BCE:01E7 OE           PUSH   CS          ;And more code
1BCE:01E8 A28000       MOV    [0080],AL
1BCE:01EB EBE5         JMP    01D2
1BCE:01ED C606800000   MOV    BYTE PTR [0080],00
1BCE:01F2 A08100       MOV    AL,[0081]
1BCE:01F5 FEC0         INC    AL
1BCE:01F7 3C0A         CMP    AL,0A
1BCE:01F9 7405         JZ     0200
1BCE:01FB A28100       MOV    [0081],AL
1BCE:01FE EBD2         JMP    01D2
1BCE:0200 C606810000   MOV    BYTE PTR [0081],00
1BCE:0205 A08200       MOV    AL,[0082]
```

looked like Listing 5. That is perfectly normal code as well. If you happened to come across it, especially if it was surrounded by a bunch of other similar MOV instructions, you might assume that something was being set up such as music, sound effects, graphics, or whatever. Actually though, it is some sneaky code whose execution results in overwriting the first piece of innocent code so that it will now look like Listing 6.

Listing 6 is, of course, the jump table we've seen before. It doesn't exist until after the self-modifying code has been executed. A trick that's even more difficult to decipher is using obscure routines to calculate the bytes. Anything can be done by the programmer, whose rule is to make the copy-protection technique as hidden and arcane as possible.

In general, if you can't find the jump table or the CALLs to the major routines in a piece of software, you're probably up against self-modifying code. The best approach is still to find the CALLs that run the routines that you can recognize in the game. Those include such things as playing music and putting messages on the screen.

Once you see a connection between sections of code and different events in the game, you can work backwards to locate other signposts. It's not easy—it's designed to be as difficult as possible. The only piece of encouragement I can offer to you is that the more you do this, the better you'll get.

You won't find any ready-to-use software that can do this job for you, and you won't find a shelf full of books to help you either. You've got to do it on your own and, as an extra side benefit, the better you get at deciphering copy-protection schemes, the better your own programming skills will become.

When we get together next time, we'll get back to circuits. Ω

AUDIO UPDATE

continued from page 80

But aside from those cavils, I found Davis's conclusions to be right on target. He found the differences between two-wire cables to be indistinguishable. Comparing two-wire cables with flat ribbon cables revealed a slight difference in the high treble averaging about 0.5 dB at 15 kHz. These differences are at the threshold of audibility and are a far cry from the "quantum leaps" of improvement extolled by the manufacturers and the audiophiles they've entranced.

Davis confirms my view that unless strange amplifier or speaker impedances are involved, 12-gauge "zip cord" will work fine. He concludes that the use of exotic materials for the conductors, or strange cable insulation, or oxygen-free copper multi-strand "hyperlitz" windings will have minimal positive effect on the signal, but maximal negative effect on your wallet. Ω