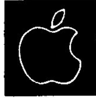
Open-Apple™



February 1985
Vol. 1, No. 1

Releasing the power to everyone.

My Two Bits

by Tom Weishaar

"Congratulations on avoiding the 'Apple Computer Co. = Apple II computer' trap," writes Brooke Boering (author of the freeware programs *Fire Organ* and *One-Key DOS*) in response to last month's *My Two Bits*. ***Open-Apple*** is primarily about the Apple II **computer**. But it's important to keep a watchful eye on our friends at Apple, Inc., as well. As noted here last month, there is far more respect for the Apple II **outside** the company than **inside** it. There is tremendous technological pressure inside Apple to "improve" the Apple II beyond recognition, even though those of us on the buying side are only beginning to understand what we can do with the unimproved model.

This pressure doesn't come entirely from within Apple. The February *Popular Computing* includes a review of the Apple IIc that includes statements like these:

"I found the IIc to be just a second facelift for the Apple II Plus. Cosmetic surgery is fine in some cases, but nothing short of major internal surgery will modernize the Apple II line."

"From a design perspective, compatibility is the IIc's Achilles' heel. To maintain that compatibility, Apple's engineers couldn't make the IIc very different from its predecessors."

"The IIc isn't a bad computer. It's just a computer based on aged technology."

The technonerd that wrote that review, like his many confederates inside Apple, has precious little understanding of why we users have purchased more than two million of these machines. We want them precisely because the Apple II has *evolved*.

Ask hackers who were weaned on Radio Shack Model Ones, Commodore Pets, or Texas Instruments 99/4s what they eat for dinner nowadays. All the time and energy they spent learning how to use those lizards has lost its worth. Meanwhile, those of us lucky (or wise, as the case may be) enough to start with an Apple have something of appreciating value inside our heads.

It is ***Open-Apple's*** editorial policy to protect that investment. We'll happily make mock of *Popular Computing* or any other publication that attempts to liquidate our investment. And we'll be watching developments at Apple, Inc. for internal attempts to bankrupt us, too.

It's important to realize that those of us who already own Apple IIs represent a significant proportion of Apple Computer, Inc's. future revenue. It's a fundamental rule of business that people who have bought a company's products in the past are more likely than anybody else to buy again.

According to press reports, Apple sold about 750,000 IIes and IIcs in 1984 (up from 580,000 in 1983). About a third of those went to homes, offices, or schools that already had an Apple in the family. That proportion will increase, not decrease, from here on out.

Looking at the data another way, 37 per cent of the people who already own a computer expect to buy another within a year, while only 13 per cent of non-owners expect to buy, according to a study done last summer by Dataquest and Software Access International.

Apple's president, John Sculley, is a marketing man. Marketing men live by the commandment that a company shalt produce goods that customers want to buy rather than goods that engineers want to produce.

Unfortunately, it isn't at all certain that Scully and Apple are taking the desires of current Apple II owners seriously—even though we represent a ton of future business potential. One gets the impression that their market research is totally based on people who have either never seen a computer before or who have used only competitive brands.

The latest example of this disregard for current owners is word from reliable sources that future models in the "Apple II family" won't support DOS 3.3. (If it won't support DOS 3.3 it's not an Apple II, is it?)

The reason the new machines can't use DOS 3.3, Apple's engineers say, is that Apple is converting all its computers to the 3 1/2-inch plastic-encased disks used on the Macintosh. These disks hold 409,600 bytes of data. DOS 3.3, we are constantly misinformed (most recently in the January 1985 *inCider*, page 82), can handle disks no larger than 143,360 bytes.

As the owners of 40-track drives know, this upper limit is all myth. DOS 3.3 will work, *without modification*, on disk drives having as many as 50 tracks and 32 sectors (the equivalent of exactly the Macintosh's 409,600 bytes). Apple's engineers, however, don't seem to know this. (Quick, name the only company that sells a hard disk for the Apple II that doesn't support DOS 3.3.)

I can't believe a marketing man like Sculley would let a decision like this be made if he knew both the true potential of DOS 3.3 and how his best customers feel about it. Since his marketing and engineering minions don't seem to be telling him, I guess you and I will have to. Fire up your word processor and let Sculley know what you think. In case you don't have it handy, his address is:

```
John Sculley, President
Apple Computer, Inc.
20525 Mariani Ave
Cupertino, Calif.  95014
```

Lest I be misunderstood, let me make it clear I have nothing against 3 1/2 inch disks. I think they're wonderful. I have nothing against ProDOS. I think it's wonderful too. But the common investment of time, knowledge, and productive software Apple users have in DOS 3.3 is simply too valuable to walk away from. If Apple's engineers can't figure out how to get DOS 3.3 to work with 400K disks, there are a dozen or more companies with names like Corvus, Rana, MicroSci, and Diversified Software Research who could show them how. It's time for Apple to spend a little more of its research and development money on its bread and butter product.

Enough of my blustering. The theme of this month's ***Open-Apple*** isn't monitoring Apple—our main task, actually, is to investigate the Apple Monitor. First, however, I want to thank all of you who have already signed on as ***Open-Apple*** subscribers. The response rate you've given our subscription campaign has been fantastic—and we didn't even have a sweepstakes.

A Song Called the System Monitor

One of the greatest death scenes in all movie history, according to a film critic friend of ours, occurs in *2001: A Space Odyssey*. HAL, an intelligent computer on board a spaceship bound for Jupiter, decides the biggest risk to his mission's success is human error and becomes very hostile toward the astronauts accompanying him.

Dave, a human spaceman, eventually works his way into HAL's memory bank and starts removing ROMs and RAMs (which look like small glass slabs). As each slab is removed, HAL loses intelligence. HAL pleads with Dave to stop. As Dave yanks out HAL's last bit of intelligence, HAL is singing *Daisy*, which was the first thing he ever learned.

Fortunately, Apples are meek and mild-mannered machines. But if the day ever comes that you find it necessary to climb inside and start removing intelligence, the last thing your Apple will sing is a song called the *System Monitor.* This is the first thing Apples ever learned. It includes software Steve Wozniak wrote in 1976 to impress his friends the Homebrew Computer Club.

Cornerstone treasures. The Monitor software is very elementary. But it's the cornerstone of the Apple's operating system.

When people talk about the Monitor, they usually mean one of two things (or three things, if you count display screens). In one sense, the Monitor consists of all the machine language subroutines found between byte 63488 (in hex that's $F800) and 65535 ($FFFF). These routines do things such as retrieve the messages you type on your keyboard, print text on your screen, and beep your speaker.

In addition, these routines include a program that allows you to interact with your Apple. Using this program, you can determine or change the contents of any memory byte, move data around in memory, verify that two ranges of bytes are the same, and do hexadecimal arithmetic and other stuff. This program, although it is really just a subset of the entire Apple system monitor, is also usually called the Monitor.

The Monitor program, like the rest of the system monitor, is very elementary. Most of the commands consist of a single character. Error messages consist of a single beep. Nonetheless, it provides a way for you to "get inside" your Apple and deal with it on a byte level—at any time. Very few other computers have a built-in program that opens the system up to you as much as this one does.

In and out. The Monitor program starts at byte 65385 ($FF69). You execute the program by entering *call 65385*, or more typically, *call -151* (a negative 151 and a positive 65385 appear to Basic as the same number when you're using the call command). When you enter one of these commands, the Applesoft bracket prompt will change to an asterisk, like this:

```
]CALL -151

*
```

The asterisk prompt is your signal that you are "in the Monitor". Now that we're here, let's figure out if there's anything we can do that's useful.

First of all, try entering a familiar Basic command. How about:

```
*PRINT "HI"
                    (beep)
*
```

No luck. That beep, however, is the Monitor's signal that you made a syntax error. Notice that no error message is printed on the screen. How about a DOS command? Try this:

```
*CATALOG

DISK VOLUME 254

 A 002 HELLO
  ...and so on
```

DOS commands work just fine from within the Monitor. I've just shown a DOS 3.3 catalog, but ProDOS works too. With DOS 3.3 you'll see a string of numbers following the command's display. ProDOS eats these for breakfast. If the numbers intrigue you, look at the **If Then Maybe** column in the June 1984 *Softalk.* On page

Reviewer's Corner

The Sider. By First Class Peripherals. The Sider is a 10 megabyte hard disk drive for the Apple II-Plus and IIe that sells, by mail order only, for $695. At this eye-opening price I thought we'd better try one. So far I'm impressed.

About four weeks after I placed my order, UPS delivered a *massive*, mostly empty, box. Inside were the drive itself, a peripheral card for slot 7, and some cables and instructions. The thing takes about two hours to install and initialize.

After the Sider is installed, your system uses it for booting when you turn on your computer or press open-apple-reset. If you haven't turned the Sider on, your computer will "lock-up"; but you can press control-reset to enter Basic and PR#6 to boot from your floppy drive if you want to go that route.

When the Sider starts up, a menu appears that gives you your choice of booting CP/M, Pascal, ProDOS, or DOS 3.3. You can also boot a floppy disk in slot 6 (it usually won't be able to use the hard drive, however), run system utilities, or park heads (quit) from this menu.

One small problem with the way the Sider supports all four operating systems is that you have to set aside at least a few sectors on the disk for each of them. The actual share of space each gets is up to you. The system utilities and the main menu program run under DOS 3.3. It's the only operating system that actually comes with the drive—you must buy the master disks for the others from the appropriate source, as usual.

Under DOS 3.3, the Sider appears to be a bunch of floppy disks, each with a different volume number, in slot 7, drive 1. If you initialize the Sider with the maximum amount of DOS 3.3 disk space, you'll get 61 standard 140K volumes, numbered 1 through 61. You access these with the standard DOS 3.3 *V* parameter, for example, *CATALOG, V33.* You can switch back and forth between these 61 "disks", or *volumes*, simply by specifying different numbers. If you don't specify a number, you get whatever volume you used last.

Under DOS 3.3, the Sider also allows you to have large, 400K volumes. As mentioned in *My Two Bits*, DOS 3.3 can handle disks of this size without modification. You can have up to 22 of these larger volumes at one time.

As implemented on the Sider, ProDOS must always have two equal-sized volumes. They're called */hard1* and */hard2.* If you maximize the ProDOS space at initialization, each volume will have 9,076 blocks of space—a little over 4.5 megabytes each. The ProDOS system of *prefixes, subdirectories,* and *pathnames* makes a good deal more sense when you have large amounts of space like this to work with.

Besides their large capacity, hard disks are also famous for their speed. This is because the disk is always spinning—there's no coming-up-to-speed delay—and because it spins much faster than a floppy can. The following chart compares how long it takes to sequentially load ten high-resolution graphics from a floppy disk, the Sider, and a RAM (memory-based) disk. It's interesting to note that the operating system you use makes a lot of difference. If your primary interest is speed rather than capacity, a speed enhanced DOS—such as ProntoDOS, DiversiDOS or ProDOS—helps more than a hard disk or even a RAM disk.

Seconds required to bload ten hi-res graphics

	floppy	Sider	RAM disk
standard DOS 3.3	88	31	24
high-speed 3.3	20	9	3
ProDOS	18	6	2

An unusual feature of the Sider is that one periperhal card can handle a two-drive system. Most hard disks for the Apple don't have that capability. Those of you with uninterruptable power supplies (review upcoming in a future issue) will be happy to learn that the Sider draws only 40 watts of power.

If the Sider has a weakness, it's the documentation. Don't expect the manual to be as slick as the company's advertising. The manual does an adequate job of explaining how to install and initialize the drive and how to troubleshoot any problems. However, it's quite thin on actual operation. The "park heads" selection on the main menu, for example, isn't mentioned in the manual. I assume it's bad news if the lights go out with the heads engaged, but who knows—the manual doesn't say *anything* about how to turn the unit off.

The Sider comes with a one-year warranty. To receive service you must first follow a diagnostic prodecure outlined in the manual. If the unit is defective you then call to get a repair authorization and return the drive in its original packing to First Class Peripherals. They fix it free while it's in warranty; they charge $150 after that.

Are these folks for real? Apple charges $1,495 for its 5 megabyte Profile hard drive—half the capacity at twice the price. This is a new company and buyers take the risk they won't be around long. However, they appear to be doing things right. Their strategy is to sell by mail at below retail prices. Their market is Apple II users willing to forgo a dealer's handholding in exchange for those prices. (I suspect there's hundreds of thousands of people like that.)

The drive itself is manufactured by a company called Xebec. Members of the Apple users group in Miami bought a bunch of these drives a few months ago and our friends down there say they've been happy with them. I'm certainly happy with mine.

The Sider, by First Class Peripherals (P.O. Box 6187, Lehigh Valley, Penn. 18001; 800-538-1307). One-year warranty. $695

Downloaded from www.Apple2Online.com

141 of that issue, Roger Wagner throughly explores where these meaningless numbers come from and why. If these numbers don't intrigue you, just ignore them.

Now open the door on your disk drive and try the catalog command again. This is a good way to force a disk *i/o error.* Why do that? Try it and see:

```
*CATALOG

I/O ERROR (beep)

]
```

This time you do get an error message. This is because DOS is sending the message to you. But the important thing to notice is the prompt you get after the error. A DOS error from within the Monitor always lifts you back up into Applesoft. Enter *call -151* again, and try out what happens when you press control-reset. Once again the Applesoft prompt will reappear.

So we've established that *call -151* gets you into the Monitor and that *control-reset* gets you back out. Basic commands don't work in the Monitor; DOS commands do; however, they send you to Applesoft after an error.

Hexamining memory. Now let's try just pressing return a few times. Voila! You should get a display that looks something like this, but probably with different numbers:

```
* (press return)
B800- A2 00 A0 02 88 B1 3E 4A
* (press return)
B808- 3E 00 BC 4A 3E 00 BC 99
* (press return)
B810- 00 B8 E8 E0 56 90 ED A2
*
```

Not your normal, school-kid, decimal numbers, mind you; but *hexadecimal* numbers.

In our friendly, everyday decimal system, each digit has ten possible values, 0 through 9. The digit on the far right represents units or ones; the next one to the left, tens; the next hundreds (10 * 10); the next thousands (10 * 10 * 10) and so on.

The hexadecimal system works exactly the same way, except each digit has sixteen possible values, 0 through F. (Start at 0, go up to 9, use A for 10, B for 11; F for 15.) Just as in the decimal system, the digit on the far right represents units or ones; the next one to the left, sixteens; the next 256s (16 * 16); the next 4096s (16 * 16 * 16); and so on.

A two-digit hexadecimal number can range in value from $00 (0) to $FF (255). A dollar sign in front of a number indicates it is a hexadecimal number. A four-digit hexadecimal number can range in value from $0000 (0) to $FFFF (65535).

Look again at the display you get by simply pressing on the return key. The numbers on the left edge of the screen are four-digit hexadecimal numbers. They represent an *address* in your Apple's memory. The bytes that make up computer memory can be considered small electronic cells that can hold any value between 0 and 255. Each of these cells has an address—kind of like a Post Office Box number. From the Monitor, you can examine the contents of these cells by simply pressing return.

In the above example, the contents of cell $B800 (47104) is $A2 (162); $B801 holds $00; $B802 holds $A0; and so on. Each line of the display shows you the values in eight consecutive bytes. The second line begins with byte $B808, the third with byte $B810.

Fine, but what if you want to see the value in byte $03D0, not byte $B800? Just say so:

```
*03D0
03D0- 4C
```

You can determine the value currently in any of your Apple's memory bytes by simply giving the address of the byte you are interested in (in hexadecimal) and pressing return. If you specify an address with a number that has less than four digits, leading zeros will be added automatically. If you specify more than four digits, all but the final four will be ignored. Thus:

```
*2
0002- D4

*20
0020- 00

*20000
0000- 4C
*
```

If you are interested in the values in a *range* of memory bytes, you have two choices. The first is to give the address of the first byte you're interested in and then press return a bunch of times. This gives you a display like this one:

```
*3D0

03D0- 4C
*                        (press return)

 BF 9D 4C B4 9D 4C FD
*                        (press return)

03D8- AA 4C B5 B7 AD 0F 9D AC
*                        (press return)

03E0- 0E 9D 60 AD C2 AA AC C1
*
```

Alternatively, to see a range of bytes you can give the beginning address of the range followed by a period and the ending address, like this:

```
*3D0.3E3

03D0- 4C BF 9D 4C B4 9D 4C FD
03D8- AA 4C B5 B7 AD 0F 9D AC
03E0- 0E 9D 60 AD
```

If you give a long range, such as *2000.3000*, the values will quickly scroll by on your screen. You can stop the scrolling by pressing control-S; restart it by pressing control-S a second time (any other alphanumeric key will also restart scrolling).

You can send these *memory dumps* to your printer, if you like. Use the DOS command *pr#1* (assuming your printer is connected to slot 1) to pass output to your printer. Use *pr#0* to stop it again. (The Monitor also includes its own versions of the *pr#* and *in#* commands, but they interfere with DOS. Always use the DOS versions.)

Disassembling your Apple. If you have a silcon-based brain, you may find these rows of hexadecimal numbers very interesting and useful. If, on the other hand, you have a carbon-based brain like most humans, you probably find them difficult to work with. For you the Monitor has a command called List. Beginners won't find its display any more intelligible than the last, but experienced users find it quite useful. To use List, enter an address, put an L immediately after it (don't include a space), and press return:

```
*3D0L

03D0-   4C BF 9D    JMP $9DBF
03D3-   4C 84 9D    JMP $9D84
03D6-   4C FD AA    JMP $AAFD
03D9-   4C B5 B7    JMP $B7B5
03DC-   AD 0F 9D    LDA $9D0F
03DF-   AC 0E 9D    LDY $9D0E
03E2-   60          RTS
03E3-   AD C2 AA    LDA $AAC2
03E6-   AC C1 AA    LDY $AAC1
03E9-   60          RTS
03EA-   4C 51 A8    JMP $A851
03ED-   EA          NOP
03EE-   EA          NOP
03EF-   4C 59 FA    JMP $FA59
03F2-   BF          ???
03F3-   9D 38 4C    STA $4C38,X
03F6-   58          CLI
03F7-   FF          ???
03F8-   4C 65 FF    JMP $FF65
03FB-   4C 65 FF    JMP $FF65
```

This range of memory (which you'll only get if you booted with a DOS 3.3 disk) begins at the same byte as our earlier row-by-row display. Again, the numbers on the left indicate the addresses of the bytes revealed on the right. The numbers in the three columns in the middle of the display are the actual values in these memory cells. If you compare them to the row-by-row display you'll find they are the same.

The difference is that the List command assumes these numbers are a machine language program. List organizes the numbers into a meaningful display. On the right side of the display is an assembly language translation of the machine language instructions.

For example, the machine language code for *jump* (equivalent to Basic's *goto*) is $4C. The address of the machine language program that is to be jumped to always follows the jump command byte. In machine language, addresses are usually written backwards. (No, I don't know why.) Thus the *4C BF 9D* in the first line of our display translates into *JMP $9DBF.*

The List command isn't always able to give a true translation of a range of machine language instructions. For example, look at the question marks near the bottom of the listing. The stuff stored in bytes $3F2 through $3F4 is data, not a command. But the List command has no way to know that. It insists on translating the data as if it were a meaningful instruction and gets nowhere.

If you have a IIc, incidentally, the List command will correctly list the additional codes of the 65C02 instruction set, something the List command in other versions of the II can't do.

Now you know how to list the contents of your computer's memory as raw hexadecimal values and as a machine language program. There is one other useful format for examining memory—as ASCII values. Many bytes hold a number that actually represents a letter or other character. The Monitor can't show you those letters and numbers directly, but we'll show you how to get a peek at them in a minute.

Warning: when examining your Apple's memory with any of these techniques, avoid the area from $C000 to $C0FF. This range of memory addresses doesn't contain true memory bytes. Instead, it's full of soft switches that control your Apple and the peripherals connected to it. Throwing these switches accidentally can mess things up. Non-believers may enter *C055 (return)* and rejoin us after using control-reset to recover.

Making changes. We've looked at how you can enter the Monitor with the call -151 command. There are also other ways to enter the Monitor. You may be familiar with "crashing" into the Monitor, which happens when a machine language program executes a *break* instruction. The reset key can also be used get into the Monitor if you make some adjustments to a few special memory cells.

Scan back and look at that "data" we saw earlier at bytes $3F2 through $3F4 when we used the List command. What that data actually represents is an address the Monitor will jump to when you press the reset key, and a check byte that makes sure the reset address is current. If you look at bytes $3F2 and $3F3 in the above display, you'll find they say BF 9D. Turn them around, since they are in the backwards machine language notation, and you have $9DBF. This is the same address the first line of our display jumped to.

Under DOS 3.3, $9DBF is the beginning of the DOS "warmstart" routine. (Under ProDOS, those bytes point to $BE00, which is Basic.system's warmstart address.) Thus, pressing reset normally warmstarts DOS, which in turn warmstarts Basic, which in turns explains why we go from the Monitor to Applesoft when we press the reset key.

If instead of Applesoft you'd like control-reset to put you in the Monitor, all you have to do is change the address at $3F2 so it points to the beginning of the Monitor program. As we learned earlier, the Monitor begins at $FF69. To make it all work, you also have to put a $5A in byte $3F4. If you want to know why, read the June 1983 **DOStalk**, which examined all this in detail. You might also like to look at *Trapping Reset* on page 16 this letter.

So now we know we want to put 69 FF 5A in bytes $3F2 through $3F4. But how?

In addition to allowing you to examine memory, the Monitor makes it quite easy to *change* the values in memory cells. As before, you give the address you want to start at, follow it immediately with a colon, and then enter the values you want in the specified bytes. You must *separate the values with spaces*, like this:

```
*3F2:69 FF 5A
*
```

You can change up to 85 consecutive values in one command line—just make sure you put a space between all the values. You can also enter a few values on one line and continue where you left off in the next line by simply entering a colon with no leading address.

After changing some bytes, you can confirm the presence of your new values by pressing return as many times as necessary. The returns will produce a memory display that begins at the last complete address you entered. Watch:

```
*2000:11 22 33
*:44 55 66
*:77 88 99
*:AA BB CC
*                           (press return)
2000- 11 22 33 44 55 66 77 88
*                           (press return)
2008- 99 AA BB CC 00 00 00 00
*
```

Let's assume you've made the *3F2:69 FF 5A* change to fix your Apple so that control-reset will always take you to the Monitor (until the next time you re-boot, anyhow). Now how do you get back to Basic?

The first line of the disassembly display we looked at earlier also contained a jump to the DOS warmstart address. The Monitor has a command, called Go, that allows you to execute the instructions at any memory address. To get back to Basic now that we've changed how control-reset works, we can do this:

```
*3D0G
]
```

In this command, *3D0* (note that's a zero, not a big o) is the address we want to jump to, the G tells the Monitor to jump. This address, $3D0, is known as the *DOS warmstart vector*, and is a worthwhile address to remember if you use the Monitor much.

Let me see letters. Control-Y is known as the Monitor's *user* command. Assembly language programmers can write a special routine, place its address at $3F9-$3FA, and execute it directly from the Monitor by entering control-Y. Here's a special routine that you can use to change the Monitor's memory dumps from hexadecimal numbers to their equivalent ASCII characters:

```
*3F9:00 03                   (put address of routine at $3F8)

*300:A5 3C 29 07 D0 03 20 92  (put routine at $300)

*:FD 81 3C 48 29 7F C9 20

*:B0 08 68 09 C0 48 A9 DE

*:D0 02 A9 A0 20 ED FD 68

*:20 ED FD 20 BA FC 90 D8

*:60
```

After typing all this in, check it carefully. If you heard any beeps as you were entering the line, you made a mistake—probably you substituted an O for zero.

Now you can examine memory in ASCII by entering the examine memory commands just as before but also pressing control-Y before pressing return. Characters preceeded by an ^ in the display are control characters:

```
*A880.AB00(control-Y, return)

A880-^]^E 7  I N I T
A888- L O A D S A V E
   ...and so on.
```

This month you've learned how to examine the contents of your Apple's memory as raw hexadecimal values, as an assembly language program listing, and as ASCII characters. You've also learned how to enter new hexadecimal values into memory. Next month we'll examine some of the fancier things the Monitor can do, such as move and compare values. For now, however, let's look at some Monitor tricks involving Applesoft, DOS, and Logo.

Picking Up Applesoft

The Basic connection. Getting the Monitor to dump memory in ASCII as we've just done is a nice feature, but typing in all those hexadecimal numbers is a pain in the chips. If we were dealing with Basic, we could write a program and save it, but there's no way to do that from the Monitor. We could save those two ranges of memory we've typed in as binary files and write a Basic program that would reload them, but then you wouldn't learn anything new.

Several years ago a fellow named S.H. Lam wrote a letter to *Call -A.P.P.L.E.* in which he showed how to execute Monitor commands from Basic. This technique has since been used all over the place. I even used it in **DOStalk** in March (DOS 3.3 disk free space patch) and May (DOS 3.3 type command).

The technique provides for entering, editing, and saving machine language programs and requires relatively few characters to be typed, which cuts time, effort, and mistakes.

Unfortunately, people often have trouble getting all the Is dotted and Ts crossed when using this technique—especially when typos sneak into program listings, as happened in the May **DOStalk**.

The technique works like this. You write an Applesoft program that simply defines what you would normally type into the Monitor as a string. After defining the string, your program jumps to a subroutine that gets Applesoft to type the string into the Monitor for you. Here's the magic subroutine:

```
500 C$=C$ + " N D9C6G" : REM space required before and after N
510 FOR I=1 TO LEN (C$)
512 : POKE 511+I, ASC(MID$(C$,I,1))+128
514 NEXT
520 POKE 72,0 : CALL -144
530 RETURN
```

The first line of this routine tacks *N D9C6G* onto the end of your command string. Note that there *must be a space* before and after the N. More than 95 per cent of all reported problems with this technique are caused by a missing space between the opening quote and the N.

If your command string (C$) is *300:0*, then after line 500 is executed it will be *300:0 N D9C6G*. Lines 510-514 poke this command string into your Apple's keyboard input buffer. Line 520 calls a machine language subroutine inside the Monitor that executes whatever's in that buffer.

In this case the Monitor would put a zero at byte $300, set display to normal, and jump to $D9C6. $D9C6 is a charmed spot inside Applesoft that continues execution of our Applesoft program. In this case we continue at line 530's *return* from the Lam subroutine.

One problem with the Lam technique is that the shortcut we used earlier of entering a colon without an address doesn't work. Here's how you would set up an ASCII memory dump loader using the Lam technique:

```
100 C$="3F9:00 03" : GOSUB 500
110 C$="300:A5 3C 29 07 D0 03 20 92" : GOSUB 500
120 C$="308:FD B1 3C 48 29 7F C9 20" : GOSUB 500
130 C$="310:B0 08 68 09 C0 48 A9 DE" : GOSUB 500
140 C$="318:D0 02 A9 A0 20 ED FD 68" : GOSUB 500
150 C$="320:20 ED FD 20 BA FC 90 D8" : GOSUB 500
160 C$="328:60" : GOSUB 500
170 END

500 C$=C$ + " N D9C6G" : REM space required before and after N
510 FOR I=1 TO LEN (C$)
512 : POKE 511+I, ASC(MID$(C$,I,1))+128
514 NEXT
520 POKE 72,0 : CALL -144
530 RETURN
```

To save even more typing time, the lines from 100 to 170 could be combined. But there's an even better way, and it's a DOS trick.

Digging Into DOS

Writing Monitor programs. Earlier in this letter, I pointed out that DOS commands work from within the Monitor. This flexibility leads to an interesting trick.

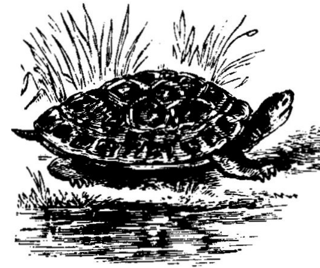
When considering DOS commands that you can use inside the Monitor, you will think right away to try *catalog*, but how about *fp* or *bload myfile* or *-/users.disk/convert*? Here's the real beauty, though, *exec ascii.dump*.

The exec command works just fine from inside the Monitor. What you put in the text file you'll exec is bare Monitor commands. Instead of typing in a whole Basic program, all you really have to do is sit down at your word processor and enter:

```
CALL -151                      (optional, not needed if you exec from
                                      within the Monitor.)
3F9:00 03
300:A5 3C 29 07 D0 03 20 92
308:FD B1 3C 48 29 7F C9 20    (as with the Lam technique, you must use
310:B0 08 68 09 C0 48 A9 DE          a loading address on each line.)
318:D0 02 A9 A0 20 ED FD 68
320:20 ED FD 20 BA FC 90 D8
328:60
```

Use a word processor to create an exec file filled with Monitor commands. This gives you the ability to edit your mistakes and to save your work. Now we can write "Monitor programs". It works with both DOS 3.3 and ProDOS.

If you get into this, you might want to use it to edit our control-Y program a little. Change byte $303 to $0F and you'll get a display 16 columns wide. A $1F at that spot will give you a 32-column display—a nice use for your 80-column screen (which, of course, you can turn on from within the Monitor with a *pr#3*).



Go, Logo, Go!

It's not the kind of thing they teach your children in school, but the Apple Monitor is readily available from Logo just as it is from Basic. With most Logos, the command *call -151* or its equivalent is probably the most convenient way to get into the Monitor.

With Apple's 128K version, however, getting into the Monitor requires a few tricks. You can't simply call the Monitor's starting address, because Logo turns on and uses all 128K of RAM memory. The Monitor ROM is turned off and is invisible when you're inside Logo.

On Apple's 128K computers, pressing Reset automatically turns on the ROM, making it momentarily accessible. Normally Logo traps a press of the Reset key, as we did earlier in this letter. It quickly turns the ROM back off, the RAM back on, and informs you that you've pressed the Reset key and should reboot. The reboot message seems a little heavy handed, however, since your program and data are still intact and ready to run.

To get into the Monitor, we can change the contents of the soft-entry vector. Then Reset will provide our passageway in. This is easy enough. The problem then becomes one of getting back out again. Apple's 128K Logo doesn't support the standard $3D0G warmstart address.

To solve this, let's move the contents of Logo's soft-entry vector into the control-Y vector. Then when we press control-Y from inside the Monitor, we'll go back to Logo via its Reset trap, and all necessary bank switching will be handled for us.

Logo commands will look like gibberish to those of you who have never encountered them before, so we'll explain them a little. In Logo you write programs by creating a series of subroutines, called *procedures*. Each procedure is given a name and you execute the procedure by simply saying its name. You might, for example, write a routine that draws a house on the high-res screen and call it *house*. Every time you enter the word another house will appear on the screen.

Let's write a short program, called *mon*, that will poke the changes we need into the soft-entry and control-Y vectors. Apple's 128K Logo does have all the *primitives* (built-in procedures) we need for this. If you are using a different version of Logo, your version may have different names for these primitives. *.DEPOSIT adr val* is like a Basic *poke*. *.EXAMINE adr* is like a Basic *peek*.

First, let's write the procedure that will do the actual poking.

```
TO DO.MON.POKES
 .DEPOSIT 1017 .EXAMINE 1010
 .DEPOSIT 1018 .EXAMINE 1011
 .DEPOSIT 1010 105
 .DEPOSIT 1011 255
 .DEPOSIT 1012  90
END
```

The *to* tells Logo we are writing a procedure. This causes what we type to be stored, rather than executed immediately; the same effect line numbers have in Basic. The effect lasts until we type *end*, which tells Logo we are done with that procedure.

After the *to* you put the name of your procedure. On the next and following lines come what you want to do. The first two lines of our procedure move the Reset-trap address that Logo keeps in the soft-entry vector into the control-Y vector. The final three lines poke the address of the Monitor program into the soft-entry vector and correct the power-up byte.

One problem with *do.mon.pokes* is that if you run it twice, by accident or otherwise, the control-Y vector ends up with the Monitor's address in it and you can't get back to Logo. Let's write another procedure that will call *do.mon.pokes*, but only if it hasn't been run already. We can tell by looking at byte 1018, which always equals 255 before running *do.mon.pokes* and some other value afterward.

```
TO MON
IF 255 = .EXAMINE 1018 [DO.MON.POKES]
END
```

That's all there is to it. To execute, type *mon*, then press Reset.

Ask (or tell) Uncle DOS

Detecting NOT SELECTED

Is there an easy way to detect that a printer is off before the *NOT SELECTED* message appears?

Harold D. Portnoy
Bloomfield Hills, Mich.

(For those of you who don't have the Grappler+ interface card, the Grappler places a beep and the words "NOT SELECTED" in the middle of the screen when you try to print something and the printer is turned off or deselected.)

The standard answer to this question is no. However, if you are writing software for your own system and thus know in advance what printer and interface card your program will use, it can be done quite easily.

Enter the Monitor with a call -151 and take a look at the softswitches associated with your printer. There are sixteen of them; they begin at $C080+s0, where s= your slot number. If your printer interface card is in slot 1, for example, look at the switches $C090.C09F. Start with your printer turned off.

```
   (printer off)
*C090.C09F

C090- 51 51 51 51 51 51 51 51
C098- 51 51 51 51 51 51 51 51
   (turn your printer on; leave unselected)
*C090.C09F

C090- 5D 5D 5D 5D 5D 5D 5D 5D
C098- 5D 5D 5D 5D 5D 5D 5D 5D
   (printer on; selected)
*C090.C09F

C090- 53 53 53 53 53 53 53 53
C098- 53 53 53 53 53 53 53 53
```

The values shown here are what you'll see if you have a Grappler+. Other cards will display other values. All you have to do now is peek at one of these locations and see if the expected value is present. For example:

```
10 SLOT=1
20 N$="NOT"
30 IF PEEK(49280+SLOT*16) = 83 THEN N$="":REM 83=$53
40 PRINT N$;" SELECTED" : GOTO 20
```

RUN this program while selecting and deselecting your printer. If you have a IIc, you'll find that it can't tell the difference between a printer that is selected and one that's turned off (though it can tell selected from deselected if the printer is turned on). I suspect this will be true of all serial printer interfaces.

ProDOS and compilers

Can I use a DOS 3.3 Applesoft compiler (Einstein, TASC, Expediter), transfer the object code to ProDOS using Convert, then patch the I/O hooks to get ProDOS to recognize DOS commands?

Michael Couvillon
Ft. Walton Beach, Fla.

I haven't tested it, but theory tells me it will never work. ProDOS uses a different scheme for recognizing DOS commands than DOS 3.3. Anybody out there know of an Applesoft compiler that can produce ProDOS-compatible object code?

FIND:/user group/

Do you know of any Apple II user groups in my area (northeast corner of Oklahoma)?

Claude E. Williams
Jay, Okla.

You're surrounded by them—but they're all about 60 miles away. I don't have anything on these groups except their addresses, but you could write for more information:

```
Tulsa Computer Society's Apple Users
P.O. Box 1133
Tulsa, OK 74101

Joplin Apple Users Group
1903 E. 36th
Joplin, MO 64801

Razorback H.A.U.G.
3000 Rogers Ave.
Fort Smith, AR 72901
```

If you have a modem, the closest groups to you are probably the Micronet Apple User Group on CompuServe and the Source Apple User Group.

Incidentally, we're trying to develop a data base on Apple user groups here at **Open-Apple**. *We also exchange subscriptions with groups that have newsletters.*

Msg from a parallel universe

CEREAL INTERFACE

SOUPER CEREAL INTERFACE

I tried the programs dealing with the *onerr* routines in the January issue (page 2) and put the "empty" file on the ProDOS /*ram* volume. This speeds things up greatly and makes life easy on the old disk drives. It also executed the *onerr* routine 8,400 times without fail while I watched the late night news. What else are people doing with /*ram*?

T.S. Goodfellow
Raleigh, N.C.

Mousetrapped

There seems to be a lack of good information on the IIc. I hope your publication can supply that need as well as others. For instance, I feel suckered now for buying **MousePaint**, as I have been unable to find any other programs that use the IIc mouse. Help.

Tom Milford
St. Louis, Mo.

More mouse software should be upcoming soon. Apple has provided software developers with a Macintosh-like interface for the Apple II series that uses the 80-column text screen, the mouse, and the infamous "Mousetext" characters that occasionally appear on your screen where inverse capitals should be. Software using this interface will likely begin to appear by summer.

I'm not completely sold on the mouse—we have a lot of trouble around here with the ball sliding, rather than rolling, across the desktop; this will drive you nuts with great speed. I like the features of Apple's human-computer mouseface, however.

Stalking the I/O Error

I recently bought *Terrapin Logo* for the Apple and the disk wouldn't boot. Just for the heck of it, I went under the hood and reversed my drives. To my surprise, the disk booted ok on my former drive 2 (then the door hinge broke).

At the same time, along the same lines, I got an unprotected demo disk that wouldn't load. Using *Disk Recovery* I checked it out and got a report that both sides were full of bad sectors, but, strangely, the program couldn't seem to make up its mind about which sectors were bad—they changed every time I ran the program. After my second drive was fixed, the front side breezed through the test with no errors reported.

I also tested a couple of unprotected commercial disks that were perfectly functional. However, one contained many mis-assigned sectors and the other had an invalid track/sector list. In both cases I suspect that the producers resorted to subterfuge to conceal a very substantial part of their programming and the combination could cause the unwary to delete a large chunk of an otherwise valid program.

Gardyloo,

Ralph Moredock
Campbell, CA

I have several floppy drives; the oldest has been my most-used drive for almost four years. None of them have ever been adjusted, aligned, or (blush) even cleaned. I did take one apart once to see what was inside, but I didn't tweek anything.

Based on this experience, I suspect many disk drive problems start as software problems and become hardware problems only after the drive gets "adjusted" in an attempt to fix things. Those of you who have had real hardware problems will probably vehemently disagree with me.

The Number One cause of disk problems is bad centering. Disks must be **perfectly** *centered in the drive or they won't work. There is a mechanism inside the drive that is supposed to center the disk, but it doesn't work reliably.*

Hub rings on your disks help solve the centering problem. Most disks sold nowadays come with these; I recommend you refuse to buy any that don't have them. The hub ring stiffens the magnetic material, consequently the centering device does its job better. The original purpose of hub rings was to keep the center of the disk from wearing out, but this hasn't been a problem on 5-1/4 inch disks.

The best tip for centering disks, however, is to wai until the drive is spinning before closing the driv door. Yes, I know all the books say **not** *to do that. Jus never* **open** *the door while the drive is running—o more exactly,* **writing** *on the disk. That can destro disks. But* **closing** *the door with the drive on not onl doesn't hurt anything, it actually helps.*

A disk that is accidentally initialized while off-cente

is particularly useless. As soon as you remove it from the drive you've lost any data you've saved on it. You'll never get it back in the drive in just exactly the same off-center orientation you started with. Thus, it is very important to delay door-closing when initializing disks.

Whenever you have a commercial disk that won't boot, but just sits in the drive and spins, open and close the drive door. This will center the disk and allow it to boot about 90 per cent of the time. And while we're talking about doors, let's mention that it may not help, but it certainly doesn't hurt to open the doors on your drives before you turn off your computer. That little flicker of the in-use light could mean the drive head is flickering too.

If your disk is centered properly and still doesn't work, the next most common problem is electrical interference from your display screen. Monitors and television sets work by guiding an electron beam across the screen with electromagnetic radiation. If your disk drive is close enough to the screen, this radiation will overwhelm the tiny signals the disk head is picking up from the magnetic fields recorded on the disk.

To solve the problem, simply move the disk drive farther away from the display screen. Magnetic interference occurs quite often with the IIc because the drive is naturally close to the screen. And it occurs almost every time you try to use a IIc while it's sitting on top of a television set.

If the disk still won't boot, try changing to a different model of Apple. I was about to throw out two commercial, copy-protected disks that I thought had shot craps; however, I finally figured out that both boot just fine on my II-Plus. It's only on IIes and IIcs that they won't work. This is one of the many joys of copy protection.

Finally, if your disk still won't boot, get out your DOS 3.3 System Master and BRUN BOOT13. I really have purchased a commercial program within the last year that used DOS 3.2. These disks will not boot on today's Apples unless you boot via BOOT13.

If that doesn't work either, you've covered about 98 per cent of the probable problems. If we split up the 2 per cent that's left, there's a 1.9 per cent chance the disk is bad, a .09 per cent chance of some other problem (try cleaning the edge contacts on your disk drive card), and a .01 per cent chance your disk drive is bad.

The only clear sign that a drive needs to be adjusted is that disks written with one drive can't be read by another and vice-versa. One of those suckers needs fixing. All you have to do then is figure out which one.

As to the unusual things Disk Recovery *found on unprotected commercial disks, I suspect an attempt to enhance performance rather than subterfuge is the root cause. For example, GPLE uses a very strange file structure. You can't successfully copy it by BLOADING it, finding its loading address and length, and BSAVING it with those parameters. The reason for this isn't subterfuge, however, but to get it to load quickly.*

AppleWorks vs Oki

I have an Apple IIe, Grappler+, and Okidata 92. Any idea how I can get **AppleWorks** to work with my printer? As usual, my Apple dealer doesn't have the foggiest.

Arlene Samowich
Nashville, Tenn

There is an answer to your problem. I just don't know what it is. But here's a note I found in the Princeton (NJ) Apple Users Group newsletter, "Clancy-Paul (a dealer in Princeton) has 'found' the **AppleWorks** *configuration disk that Bart Thomas donated to them and spoke about at the last meeting. Anyone who has other than an Apple Parallel Interface Card, or who's using an Okidata 92 printer, will probably need this fix to print properly."*

From yet another note in the same newsletter, I know that Bart Thomas is an alternate system operator for the Micronet Apple User Group on CompuServe. Thus, I suspect Bart could put the fix you (and many others) need on MAUG (if it's not already there) and you could download it. If the fix is available from other sources I'm sure one of our readers will tell us about it.

Dump 80

Explain how to get an 80-column screen dump to printer and I'll sign up for 100 months.

Steve Madigan
Culver City, Calif.

To do a screen dump, you have to peek directly at screen memory to get each character. Screen memory has a Byzantine organization, so begin by setting up an array that holds the starting address of each of the 24 lines on the screen, like this:

```
100 DIM LADR(23)
110 FOR I=0 TO 23 : READ X : LADR(I)=X : NEXT
120 DATA 1024, 1152, 1280, 1408, 1536, 1664,
        1792, 1920, 1064, 1192, 1320, 1448,
        1576, 1704, 1832, 1960, 1104, 1232,
        1360, 1488, 1616, 1744, 1872, 2000
```

Our program will use one loop to advance from line to line. A second loop inside the first will peek at each of the 80 characters on a line. The trick is that there aren't really 80 different addresses for each line. There are only 40 different addresses—but half of them (odd columns if you consider the first column number 0) are in main memory and half of them are in auxiliary memory. When the 80-column mode is active, you can switch back and forth between the two memory banks by hitting softswitches at 49237 (even columns) and 49236 (odd columns).

Thus, the rest of our program looks like this (don't forget to include the lines 100 to 120 above!):

```
200 IF PEEK(78)=190 THEN 250 : REM 3.3 or ProDOS?
210 POKE 43602,0 : REM brute force <cr> for DOS 3.3
250 PRINT CHR$(4);"PR#1" : REM printer on
260 PRINT CHR$(9);"90N"  : REM don't mess up screen

300 FOR LINE = 0 TO 23
305 : L$=""
310 : FOR CHR=0 TO 39
320 : : ADR=LADR(LINE)+CHR
330 : : POKE 49237,0 : REM peek even columns
340 : : L$=L$+CHR$(PEEK(ADR))
350 : : POKE 49236,0 : REM peek odd columns
360 : : L$=L$+CHR$(PEEK(ADR))
370 : NEXT
380 : PRINT L$
390 NEXT

400 REM Rehook 80-col screen without erasing it.
410 REM See tip on page 6 of January Open-Apple.
420 IF PEEK(978)=190 THEN 450 : REM 3.3 or ProDOS?
430 POKE 54,7 : POKE 55,195 : CALL 1002 : REM DOS 3.3
440 GOTO 500
450 PRINT CHR$(4);"PR#A$C307" : REM ProDOS

500 END
```

To get this program to work I had to resort to a couple of tricks you may not have seen before. In line 200 we determine if our program is running under DOS 3.3 or ProDOS. If 3.3, we need to print a return so DOS will recognize the DOS command in line 250—but a return may mess up our screen display. Thus we resort to the brute force technique in line 210.

In lines 250 and 260 we turn on the printer. Line 260 has two functions. First, it tells your printer interface card that we don't want what we print sent to the screen. It would mess up the very display we are trying to dump if it did. Secondly, it tells the card to expect 90 characters per line. Since we will be printing 80 characters plus a carriage return for each line, the card will actually see 81 characters per line. If you use the standard CHR$(9);"80N" *in line 260, your dump will be double-spaced because both your program and your interface card will send carriage returns after 80 characters.*

Lines 300 through 390 scan the display memory and place the characters found in the string variable L$. After gathering a line, L$ is printed in line 380. The routine would be a lot faster if we could print the characters as we gather them in lines 340 and 360 rather than saving them in a string for printing later. However, for reasons I don't understand, the system I tested this program on (a IIc/Imagewriter combination) consistently garbled things when I tried to PRINT while the even columns were turned on. Can any of you wizards out there explain that?

Finally, lines 400 to 450 turn the printer off and return to the 80-column screen without erasing it. For the details on how this works look on page 6 of last month's newsletter.

That will be $200, please.

Problem too complex

Here's a little program you may find interesting (I find it perplexing).

```
]LIST

10 ONERR GOTO 100
20 PRINT "HELLO FROM LINE 20"
30 A="A"
40 END
100 I=I+1 : PRINT I
110 CALL -3288 : GOTO 20

]RUN
HELLO FROM LINE 20
1
HELLO FROM LINE 20
2
HELLO FROM LINE 20     (program hangs)
                       (you press Reset)
]LIST

?FORMULA TOO COMPLEX ERROR
]
```

The call -3288 that you recommended in the January issue (page 2) doesn't work if the error is *type mismatch* or *string too long*. The computer hangs and you have to hit Reset. Then, when you try to list the program, you get *?formula too complex*. A second attempt to list the program usually works. What's going on here? I'm *extremely* frustrated at trying to bullet proof my programs.

Brent Thorwall
Crystal Lake, Ill.

I haven't been able to figure this one out—maybe somebody else can help. The stack appears to be getting messed up, but in looking at the Applesoft code I can't see why it would. I have figured out that the error isn't absolutely related to type mismatch *—try replacing line 30 with A$=A, also a type mismatch error, and it runs just like it's supposed to.*

I'm sure there's an explanation for this, anybody out there got an easy answer?

Trapping Reset

I'll be happy to contribute $24 a year to keep you productive and off the streets.

How can I direct a program to line x or line y after Reset has been hit?

Dennis Mah
Federal Way, Wash.

My kids thank you.

Your question is short, to the point, and theoretically very easy to answer. So why has it taken me two days to figure it all out? In the process I've made an incredible, new, useless discovery about the 6502 microprocessor. Please, no more questions like this one!

Part of the reason your question was so hard was that I was sure I could answer it with a few PEEKS and POKES. I wanted to do this completely from Basic. I didn't want to have to stick any little assembly language routines anywhere.

The reason I had trouble is that pressing Reset, unbeknownst to me, scrambles the contents of the 6502's stack register. Applesoft uses a small section of the stack to parse command lines. If the stack register accidentally points at this area after a Reset, Dr. Basic trips over his own feet and tumbles to the ground.

In addition to scrambling the stack pointer, pressing Reset disables interrupts and causes the 6502 to fetch the address stored at $FFFC and jump there. On all but the earliest Apples $FFFC points to $FA62. The routine there is called, appropriately enough, reset.

That routine puts the 6502 in binary (rather than decimal) mode; turns on normal (rather than inverse) characters; turns off graphics and turns on full-window, 40-column text with the cursor at the bottom of the screen; connects the keyboard for input and the display screen for output (disconnecting DOS from the I/O hooks in the process); turns off annunciators 0 and 1 and turns on annunciators 2 and 3; clears the keyboard strobe; turns off any active peripheral card ROM; and rings the Apple's bell.

In the IIe and IIc, special hardware circuits also turn on the motherboard ROM and the main bank of RAM. Auxiliary RAM is disabled. On earlier Apples with 64K of memory, on the other hand, Reset does not change the memory configuration.

After doing all of the above, the reset routine takes a look at byte 1012 ($3F4). If this byte, called the power-up byte, contains the proper value, the routine next jumps to the address stored in a place called the soft-entry vector, which is at 1010 ($3F2). If the power-up byte contains any other value, as it always does immediately after you turn on your computer, reset will attempt to boot a disk.

The address usually stored in the soft-entry vector is the DOS warmstart address—40383 ($9DBF) for DOS 3.3; 48640 ($BE00) for ProDOS.

Thus, the next thing Reset normally does is warmstart DOS. For DOS 3.3, this means the language card is switched back to whatever language had been in use when Reset was pressed; DOS is reconnected to the I/O hooks; a NOMONICO is executed; and a jump occurs to Basic's warmstart address, 54332 ($D43C). The Applesoft prompt and cursor appear on your screen. Your program and data are intact, but your program is no longer running.

ProDOS's Basic.system, on the other hand, handles warmstarts a little differently. It first jumps to the CLEAR command inside Applesoft at 54885 ($D665), which empties out all your variables and leaves them filled with zeros. This isn't real handy if you'd like to recover from a Reset without losing any data. The CLEAR routine also fixes the stack register, however—something DOS 3.3 neglects to do. Next ProDOS reconnects itself to the I/O links, pushes the cursor over to the left edge of the screen, and warmstarts Basic at 54335 ($D43F). Again the Applesoft prompt and cursor appear on your screen. Your program is intact but no longer running; your data is gone.

(Note the two Basic warmstart addresses are different by three bytes. ProDOS skips the first warmstart instruction, which sends a carriage return to the screen. A ProDOS reset doesn't make the screen scroll. A DOS 3.3 reset does. This is why. It's also why ProDOS has to push the cursor to the left edge of the screen by hand.)

Now you know everything there is to know (and much more than you probably wanted to know) about Reset. Your original question remains, however—how do you keep a Basic program running and send control to a special error routine when Reset is pressed?

In theory, all you have to do is change DOS so that instead of warmstarting Basic, it jumps to a Basic error handler. Applesoft doesn't have a "Reset" error, thus we'll have to use another one. There's a good one at 58123 ($E30B)—illegal direct. This error occurs when you try to do an INPUT, GET, or a few other commands from the keyboard. It never occurs in a running program. So let's modify DOS a little; a simple task.

DOS 3.3 keeps the Basic warmstart address in an internal vector table; it's at $9D5E (40286) when DOS is at its normal 48K location. ProDOS, on the other hand, doesn't use a vector table for this. In fact, the bytes you have to change move around from one ProDOS version to another. Luckily, they're easy to find; first look at the ProDOS warmstart vector at 48640 ($BE00) to determine where the actual warmstart code lies; the bytes that must be changed are the 11th and 12th bytes of the warmstart code (at least that's where they've been in all Basic.system versions to date).

After we make these modifications, Reset, in conjunction with an ONERR GOTO statement in your program, will pass control to ONERR with the code for an illegal direct error (149) stored where you can fish it out with the standard PEEK(222).

That's the theory anyhow. But Basic still occasionally crashes under DOS 3.3 because of stack problems and under ProDOS we lose all our data. Let's fix these problems, too. With ProDOS, we'll change the jump to the CLEAR subroutine into a jump to just the stack initialization subroutine, which lives at byte 54915 ($D683). With DOS 3.3, we'll put the jump to the stack initialization subroutine over the top of the automatic NOMONICO (good riddance!).

Whew. A little program that does all this stuff follows:

```
10 REM Reset trap

20 V1=58123 : REM address of FP illegal direct error
25 V2=54915 : REM address of FP stack initialization

30 IF PEEK(978)=157 THEN 50 : REM DOS 3.3 active
35 IF PEEK(978)=190 THEN 60 : REM ProDOS active
40 PRINT "ACTIVE DOS NOT RECOGNIZED." : END

50 A1=40286 : REM 48K DOS 3.3 Basic warmstart vector
52 A2=40401 : REM auto NOMONICO+1
53 IF PEEK(40409)=0 THEN 58     : REM shuffle some
54 FOR I=40409 TO 40400 STEP -1 : REM DOS 3.3 stuff
55 : POKE I,PEEK(I-3)           : REM around (first
56 NEXT             :REM   RUN only!) to make it work.
58 GOTO 70

60 A2=1+PEEK(48641)+PEEK(48642)*256 : REM ($BE00)+1
62 A1=A2+10 : REM fish out ProDOS adrs

70 REM 2-byte pokes adr,value
72 POKE A1+1,V1/256 : POKE A1,V1-(PEEK(A1+1)*256)
74 POKE A2+1,V2/256 : POKE A2,V2-(PEEK(A2+1)*256)

80 ONERR GOTO 200

100 GOTO 100 : REM lockup

200 PRINT "ERROR #"; PEEK(222); " IN LINE ";
      PEEK(218) + PEEK(219)*256
```

Run the program (nothing will seem to happen—it locks up in line 100) and press Reset. Line 200's message should appear on your screen. Since the illegal direct error won't ever occur naturally inside a program, you can always be sure its appearance means somebody pressed the Reset key.

Delete line 80's ONERR and RUN the program again. Now Applesoft's built-in message center will print ?illegal direct error in 100.

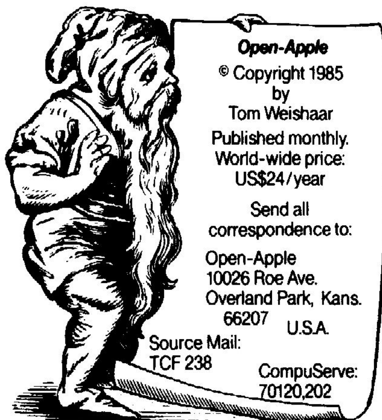
If you haven't saved the program to disk yet, **do it now**. *Put line 80 back in and add 210* RESUME. *RUN and press reset. You'll find yourself in an eternal cycle of errors. Too many things get changed when Reset is pressed to simply resume execution. Instead, warmstart your program with a GOTO. In this case, use* 210 GOTO 100.

RUN again and try pressing Control-C as well as Reset. That, Dennis, is how you direct a program to line x after Reset has been hit. The only way to get out of our sample program now is with a hard Reset (control-open-apple-reset or turn the computer off). Sure hope you saved it like you were supposed to.

Remember that Reset turns off graphics and 80-column displays, reconfigures memory, and makes other subtle system changes. Your ONERR routine (or your warmstart) may have to take special steps to set these things aright again.

The modifications we made here should never be permanently added to DOS. They cause a lot of trouble if you don't want Reset trapped. DOS 3.3's INT command could jam and 3DOG does strange stuff.

Since this seems to have turned into a definitive explanation of Reset, let's mention a couple more things. If you ignore my advice and try to change the soft-entry vector (instead of leaving it pointing to the DOS warmstart routines), CALL 64367 ($FB6F) after the change to correct the power-up byte. If you don't, the disk will boot when you press Reset. On the other hand, if that's what you **want** *to happen, just poke a zero into the power-up byte (POKE 1012,0).*



Open-Apple is a trademark of Tom Weishaar. Apple Computer and Open-Apple are two different, unrelated, independent companies that wish everyone in the world had an Apple II.