Open-Apple

Releasing the power to everyone.

January 1985
Vol. 1, No. 0

Uncle DOS survives *Softalk* bankruptcy

Matthew Monitor and Dr. Basic also found alive

Uncle DOS, the beloved leading character of *Softalk* magazine's monthly **DOStalk** column, is in good condition after a close brush with death. He will appear monthly in ***Open-Apple***, a new newsletter for Apple users, according to Tom Weishaar, former author of **DOStalk**.

Matthew Monitor and Dr. Basic, as well as many other mythical figures, have also signed to appear in the new newsletter, which Weishaar will publish himself.

Uncle DOS, Matthew, and the good doctor were found buried in *Softalk* rubble after an extensive search by Weishaar. *Softalk* entered Chapter 7 bankruptcy in August. When asked what happened, Uncle DOS reported, "We don't know—the roof just caved in on us."

Weishaar, developer of the Beagle Bros programs **ProntoDOS** and **Frame-Up**, had written *Softalk's* **DOStalk** on a free-lance basis. He had no other connection with the magazine and could not explain the bankruptcy either.

"*Softalk* was the most popular and highly-regarded of the Apple-only magazines," Weishaar said. "Its rapid growth caused problems that its managers were unable to deal with. The magazine's readers were loyal—many still are. One reader told me that if he had known the magazine was in trouble he would have taken up a collection."

Weishaar said he decided to create ***Open-Apple*** to fill one of the information voids left by *Softalk's* demise.

"There are a large number of Apple-only publications suitable for beginners," Weishaar said. "There are also magazines available for full-time professionals. What's missing is something for *intermediate* Apple users—people who want to release the full power of the Apple II in their own area of expertise. *Softalk* used to provide this information in columns such as **DOStalk** and **IInd Grade Chats**, but few other publications seem willing—or able—to fulfill the needs of the intermediate user."

Open-Apple will be published monthly, Weishaar said, with the first issue scheduled for February. The subscription price will be $24 a year. Potential readers are encouraged to encourage Weishaar by signing up as charter subscribers.

If you would like more information about ***Open-Apple***, just keep reading. You are holding a sample issue in your hands. Complete ordering information is on the back page, or if you are really lucky, you got an order card with this sample letter. *Please fill it out now, even if you don't intend to subscribe.* Your feedback on this sample issue will be carefully scrutinized and heeded.

Matthew Monitor, Dr. Basic, and Uncle DOS as seen at the "Apple II Forever" brouhaha held in San Francisco last spring.

My Two Bits

by Tom Weishaar

My banker says I'm crazy to start a newsletter about an eight-year old computer. He says eight years is an eternity in computer-time. He says the Apple II should be dead and forgotten by now like the other machines of its generation—the KIM, the Sol, the TRS-80 Model I, the Compucolor, and the Video Brain.

For most of the last eight years even the people at Apple Computer, Inc. have privately called the Apple II a "dead machine". They have consistently underestimated—often by wide margins—how many Apple IIs they could sell (while overestimating the market for the Apple III, Lisa, and Macintosh).

The high priests of computer science think the machine is a joke. Meanwhile, buyers have managed to liberate more than two million of them from Apple's factories and several hundred thousand compatible machines from other places.

In his book *Enhancing Your Apple II, Volume 1*, Don Lancaster (who because of his many articles and books about electronics is called "the father of the personal computer" by *InfoWorld* columnist John Dvorak and others) says:

The Apple II is far and away the single most powerful tool ***ever*** *put in the hands of many individuals on an uncontrolled and unregulated basis. The new personal freedoms and the potential opportunities that result from this are almost beyond belief. It's now a whole new ball game, a jump into hyperspace from where we are. The Apple II is far more significant and it will have a vastly greater impact than such short term frivolities as the automobile and television—and, possibly, even more than the printed word itself.*

Like Lancaster, I believe in the Apple II. This is the computer of the student, of the entrepreneur, of the researcher, of the cottage industrialist, of the people.

The combination of Apple's fatalistic attitude about the II and the machine's massive popularity have caused a slow evolution. Most computer companies periodically trash everything their customers know about computers through radical redesigns. Even Apple has tried this several times, but because *we keep*

...ing the Apple II instead of more "advanced" computers, our knowledge base has been protected. We have benefitted from a slow progression of enhancements to Steve Wozniak's dream machine.

Much of the software written for Apples in the late 1970s runs just fine on Apples manufactured today. Today's Apples can do more, yet none of the hours early users spent learning about their machines has been wasted. Today's Apples can still do what yesterday's Apples did. This is very unusual in a field as dynamic as computers.

The limits of the original Apple II are now in sight. The full potential of the IIe, however, isn't known. And we know even less about how to harness the IIc.

On the horizon is a 16-bit Apple II, currently known as the IIx. This machine will have the memory and processor capacity to run large, powerful programs, such as the successors to Lotus 1-2-3. But *the real beauty of the machine* is that it will also be able to run standard Apple 6502 8-bit programs, such as Applesoft and DOS 3.3. On the one hand, it will run any Apple II program; on the other, it will have horsepower today's Apple II users have never dreamed of. Best of all, it will validate your experience with and knowledge about today's Apple II.

This newsletter is dedicated to the Apple II. It will follow the advances Apple makes to this family and it will follow the advances users around the world make in pushing these machines to their limits. I want ***Open-Apple*** to be the world-wide gathering place for Apple II people.

If machines are to be pushed to their limits, it's critical that users exchange information. No one can discover everything alone. ***Open-Apple*** will be a clearinghouse for elegant tips, tricks, and procedures.

The information published in ***Open-Apple*** will come from several sources. As a software developer, I am able to obtain some material directly from Apple that is generally unavailable to the public.

I subscribe to every Apple publication I know about; some of the things you'll see here are refinements of the best ideas from other publications.

One of the most fruitful ways of learning is to take apart high-quality programs and see how they work. Much of my time is spent searching for beauty in other people's software. You'll read about my discoveries here.

Finally, many good ideas come in over the transom—in conversations and correspondence with other users. *You* may know several things about the Apple most of us have yet to discover. Part of the fun of ***Open-Apple*** is your ability to *participate* in what's published here.

The way to convince my banker I'm not crazy is to show him a stack of order cards. For that I need your help. If you believe in the Apple II, fill out the coupon on the back page of this newsletter and send it in. We have years of adventures ahead of us.

The IIc Meets the Imagewriter

After several hours of the usual frustration trying to get my Apple Imagewriter printer to work with both a Macintosh and an Apple IIc, I discovered—with the help of another user who had a working IIc-Imagewriter system—that the problem was in the Imagewriter DIP switch settings.

Using the IIc I could only print successfully when the *recognize/ignore eighth data bit switch* (switch 1-5) was closed (ignore). The Macintosh doesn't seem to care what position this switch is in—even when sending eight-bit graphics. The Imagewriter Reference Card, as you can probably guess, recommends leaving this switch open.

In addition, the IIc uses *data terminal ready* protocol (switch 2-3 open), while the Macintosh uses *xon/xoff protocol* (switch 2-3 closed). Mighty handy, isn't it?

My Imagewriter now has all the dips on switch one, except for number five, open. All are closed on switch two when I am connected to the Macintosh; to use the IIc I have to turn off the printer, open switch 2-3, switch cables, and turn the printer back on. A switch box would solve the two-cable problem, but the DIP switch problem looks insurmountable. Anybody want to buy a Macintosh?

Digging Into DOS

Reading a text file with an Applesoft program has always seemed like a pretty straightforward affair. You *open* the file, tell DOS you want to *read* from it, and *input* it.

If you know how many lines are in the file, you issue that many *input* statements, then stop. Often, however, the number of lines in the file isn't known. In that situation the typical procedure is to read the file until DOS sends an *end of data* message and then stop.

***Onerr goto* errors.** To accomplish this you issue an *onerr goto* statement. When an error occurs, program control will go to your error-handling routine. There you check to see if the error was indeed *end of data* (error number five). If not, you will want to take some true error-handling action. But when the error is simply your signal that the end of the file has been reached, you want to continue processing.

At this point most programmers send control back to their main program. This works for awhile, but if you read enough files in sequence your program will die quite unexpectedly.

For example, try this little program. It tries to read an empty file called *empty* over and over. It reports its efforts as it proceeds. After it tries the 18th time it crashes into the Monitor—fairly unusual for an Applesoft program this simple.

```
100  D$=CHR$(4)
110  N=1
120  F$="EMPTY"

200  PRINT "PROCEEDING WITH READ ";N : N=N+1
500  L=1 : ONERR GOTO 550
510  PRINT D$;"OPEN";F$
520  PRINT D$;"READ";F$

530  INPUT T$(L) : L=L+1 : GOTO 530

550  PRINT : PRINT D$;"CLOSE";F$
560  IF PEEK (222) = 5 THEN GOTO 200

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

After the program crashes, return to Applesoft with a *3D0G* and issue a *goto 570* command on your keyboard. Our program will report that we were killed by error number 77, *out of memory*, in line 550.

That may seem pretty incredible to you, given the small size of this program. But people often forget there's more than one kind of memory to run out of. The usual way to run out of memory is to have a program so large and with so many variables that it won't all fit in your computer. For example, just type on your keyboard *dim a(1000,1000)*. An array of that size won't fit and you'll see *?out of memory error* appear on your screen.

All stacked up. The other kind of memory you can run out of is called *stack* memory. Applesoft saves all kinds of stuff on the stack, which is a 256-byte area that the 6502 microprocessor can access quite easily. Whenever you do a *gosub*, for example, Applesoft pushes the current line number on the stack before passing control to the subroutine. That way, when the subroutine's *return* is executed, Applesoft can find its way home by pulling the line number off the stack and jumping to it.

Modify line 500 of our program to *500 gosub 200* and run it again. This will cause a whole bunch of *gosubs* to be executed without any *returns*. You'll again see the *?out of memory error* message; this time during the 26th pass. But unlike when you typed *dim a(1000,1000)*, you've now run out of *stack* memory rather than program memory.

Just as *gosub* pushes a line number on the stack, so does the *onerr goto* command. And the *resume* statement is *onerr goto's return*. The reason our first program crashed was that we never used *resume*.

However, the solution isn't as easy as just beginning to use it. *Resume* sends control back to the line in which the error occurred. If we tried to *resume* after an *end of data* error, we would just get another one and our program would lock up in a tight cycle of *onerr gotos* and *resumes*.

The solution to all this is to use a call to a machine language subroutine within Applesoft that will clear the stack. Use this call in place of *resume*. The call is mentioned in Apple's *Applesoft BASIC Programmer's Reference Manual, Volume 2* on page 265. The subroutine lies at $F328 and can be executed with a *call -3288*.

Earlier Apple manuals included a short machine language program that could be poked into memory and called that would also clear the stack. *Call -3288* does the same thing, and is obviously much easier to execute.

Using this call, you can put the portion of your program that reads files into a subroutine. If you try this without the call, you get a *return without gosub error* when you try to return to your main program after the first *end of data* error.

So here's the standard way to read a text file from Applesoft. This program will run forever or until you press Control-C.

```
100  D$=CHR$(4)
110  N=1
120  F$="EMPTY"
```

```
200  PRINT "PROCEEDING WITH READ ";N : N=N+1
210  GOSUB 500
220  GOTO 200

500  L=1 : ONERR GOTO 550
510  PRINT D$;"OPEN";F$
520  PRINT D$;"READ";F$

530  INPUT T$(L) : L=L+1 : GOTO 530

550  PRINT : PRINT D$;"CLOSE";F$
560  IF PEEK (222) = 5 THEN CALL -3288 : RETURN

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

Seen In Print

Steve Wozniak told *InfoWorld* to look for the **16-Bit Apple IIx** in 1986. If you missed it, *InfoWorld* carried a report of an interview with Wozniak in its November 19 issue. Wozniak said the machine would have a built-in drive, slots, expandable memory, and the 16-bit 65816 microprocessor. This chip can directly address up to 16 megabytes of memory. Of course it can also emulate a 6502 and thus run today's Apple II software.

In the October issue of *Apple Assembly Line* Bob Sander-Cederlof has a report on the **newly available 65802 microprocessor**. This chip can do internal calculations with 16 bits but handles data eight bits at a time like the 6502. It is completely compatible with the 6502—Bob installed his in a IIe and everything worked normally. Right now the chip costs about $100. The phone number for *Apple Assembly Line* is 214-324-2050.

I've always said ProDOS would become the Apple II operating system of choice when the price of hard disk drives dropped to $500. We're still not there yet, but we're getting closer. A company named **First Class Peripherals** announced a new, $700, 10-megabyte hard disk for the IIe in the November issue of most Apple magazines. The specifications look pretty good, including DOS 3.3 and ProDOS compatibility. We'll be watching this one. Their phone number is 800-538-1307.

The Apple PugetSound Program Library Exchange—the big Apple user group in the Seattle area that publishes *Call-A.P.P.L.E.*—recently changed the legal form of its organization to a cooperative and is now known as **A.P.P.L.E. Co-op**. They've always had really good mail-order prices on hardware and software for Apples; as a co-op they may sell to non-members. Members will get patronage dividends at the end of the year. For people who don't need the hand-holding local dealers supposedly offer, A.P.P.L.E. is a good place to buy stuff. Their phone number is 206-872-2245.

Go, Logo, Go

After releasing the IIc, Apple came out with a new, 128K, ProDOS-based version of Logo called Apple Logo II. It comes with some good documentation—a tutorial and a reference manual—and a copy-protected master disk with the usual "as is" warranty. The package costs $100 retail. It comes in a IIc-like red and yellow box; don't let somebody sell you the older 48K Logo if you want this one.

You might wonder why anyone would *buy* Logo when Applesoft BASIC is included "free" with all Apples. Let me duck the superior/inferior language arguments for a moment and simply point out that the equivalent amount of documentation for BASIC—the *Applesoft Tutorial*, *Applesoft Programmers Kit*, and *BASIC with ProDOS* costs $110. Thus, if your goal is simply to learn a computer language, you can't use price as a deciding factor when comparing Logo and Applesoft.

Both languages can be learned from Apple's materials. They are "easy" in the sense that neither requires formal classroom training. They are also "easy" in the sense that both are *intrepreted* languages. You can type commands in on the keyboard and get immediate execution. You can write simple programs within five minutes of cracking open either tutorial. With other languages (take Pascal, *please*) you have to know half the language before you can add two numbers together.

Logo has a built-in full-screen editor for writing programs. This is nice—especially compared to what Applesoft provides. The editor isn't quite *AppleWriter*, but it's very easy to use. It can also create or edit any kind of ProDOS text file shorter than 6,145 characters.

Many people think Logo is a graphics language. While graphics programs are very easy to write with Logo, the language goes far beyond pictures. It is derived from LISP, a language developed by artifical intelligence researchers. BASIC is a better language for dealing with numbers. Logo shines at handling lists of words, that is, data bases.

Before you run out and decide to write the Great American Data Base Program in Logo, however, you should know that there's one critical problem. Apple Logo II is *slow*. Not only do programs execute slowly, they are loaded by "execing" them into memory. I haven't experienced such long waits since I disconnected my cassette recorder and fed it to the goats. One of the sample programs Apple provides takes well over two and a half minutes to load.

Nonetheless, Logo has its advantages. It's a nicely structured language. You program by writing *procedures*, which are kind of like subroutines. The name you give a procedure becomes a new Logo command. Thus you create your own language as you use it.

The main thing that attracts me to Logo, however, is a book called *Mindstorms* by Seymour Papert. Papert is a child psychologist and educator. He and his associates developed the language. In *Mindstorms* Papert tells why.

In Papert's vision, you don't learn Logo because you want to know about or program computers. Instead you learn Logo because you want to know, period. Papert presents Logo as a language for *learning*. Programming in Logo provides intellectual models, Papert says, that are otherwise rarely available. Using the computer, these models can be manipulated by people and readily absorbed. Learning Logo is supposed to be an adventre in *learning*, not an adventure in learning *about computers*.

In this column we're going to present tips and tricks for using Logo. They'll be based on the same kind of *peeks*, *pokes*, and *calls* our other stuff revolves around. Precious little work that I can find has been published in this area, so a lot of this column will be breaking new ground. Any *input* you might add to this procedure would be appreciated.

Reviewer's Corner

Fontrix. By Data Transforms, Inc. This program has received several favorable reviews and has some impressive advertising. The Apple version of the program has some severe limitations, however.

Fontrix bills itself as "extended screen graphics software." This is a good description of its most impressive feature—the ability to create and print graphics that are much larger than a single hi-res screen.

These graphics are stored in a special file *Fontrix* calls a *graffile*. Graffile size is defined in sectors and can range in size from a single hi-res screen, which is five sectors wide and six sectors high, up to 480 sectors in any combination of width of height that's larger than a hi-res screen.

You can create and print a graphic the size of a single sheet of paper, for example. The exact size depends on your printer; the *Fontrix* manual includes a table showing the possibilites. Most Epson printers can handle graffiles 17 sectors wide by 26 sectors high. Most other printers, including Apple's, can handle graffiles 11 by 24.

The program has three basic modules, a graphic writer, a graphic printer, and a font editor. The *Fontrix* disk comes with 11 fonts. You can create more yourself with the font editor, or you can buy additional *Fontpack* character set disks from Data Transforms for $20 each. The maximum character cell size is 32 by 32 pixels. Normal Apple text is 5 by 7. Characters can be either all the same width or proportionally spaced. The only input device recognized by the font editor is the keyboard.

The graphic printer is a limited program compared to many of the graphic printers available today. It allows inverse or normal printing on 8.5 or 14 inch paper widths. A magnification option accepts choices from 1 to 255, but choices over 3 result in some of the graphic being cut off on the right hand side. The only other options available relate to where on the paper a graphic will be printed—right side, center, and so on.

The graphic writer is the module you use to actually create displays. It allows you to type fonts on the screen. Some primitive graphics abilities are also available, including background colors and patterns, but *Fontrix* is clearly not a drawing program. Graphic input devices can be used, but only to position the cursor. This can be handy when non-character fonts, such as electrical, mathematical, or engineering symbols, are being placed on the screen, but is of little use with straight text.

In advertisments you will see stunning graphics created with *Fontrix*. These were no doubt created on the IBM version of the program, which has more advanced drawing capabilities.

Fontrix is presented in advertising as a typesetting program. The Apple version is virtually useless in this application, however, because *you can't send text files* you've created on a word processor to the graphic writer. This is possible with the IBM version, but not with the Apple version. In addition, the program has no provision for justifying text on both margins. You should also know that the examples of *Fontrix* printouts included in advertising and the manuals have been photographically reduced. The manual makes this clear and recommends a reduction of about 40 per cent for best results.

The day will come when software will be available that will turn an Apple II and upcoming laser printers into typesetting machines. However, we aren't there yet.

Fontrix 1.2, by Data Transforms, Inc. (616 Washington St., Suite 106, Denver, Colorado, 80203; 303-832-1501). Unprotected. "As is" warranty. $75

Picking Up Applesoft

Taking a poke at the Garbageman

Everybody has a nemesis. Hamlet has his step-father, Batman and Robin have the Riddler, Peter Rabbit has Mr. McGregor. Applesoft programmers working under DOS 3.3 have the Garbageman.

When the Garbageman strikes, your Applesoft program will come to a sudden and complete halt. Your keyboard will go dead. Your face will look confused and angry. You will probably hit control-reset, unless you've met the Garbageman before. In that case you will sit and wait. After a pause of something between a few seconds and a few minutes, the Garbageman, his can full of electronic detritus, suddenly flees. Your Apple will mysteriously start working perfectly again.

What strings. Consider the following famous program, which originally appeared in *Softalk's* **DOStalk** a couple of years ago:

```
10 INPUT "WHAT? ";A$
20 GOTO 10
```

When you run this program, the word "WHAT?" appears on your screen with a cursor beside it. You type "WITCHES" and press return. Applesoft puts the word "WITCHES" in an empty place inside your computer. It also stores the length and location of "WITCHES" in a table used to keep track of such stuff. It's called a *variable table.*

The empty place where strings are stored extends from where the variable tables end to where DOS starts. The first string is placed right up against DOS. The next one is tucked in next to the first. Figure 1 will help you figure out where this string storage place is inside your computer.

The left-most bar of Figure 1 represents the entire range of Apple II memory and shows its standard uses. The middle bar shows how Applesoft splits up the "free" memory area from byte $800 (in decimal that's 2048) to byte $9600 (38400). Applesoft uses this area for storing your actual program and your program's variable tables and strings. The right-most bar of Figure 1 gives you a peek inside the string storage area itself.

Usually memory maps are presented with the high addresses at the top. Some of you may think the ***Open-Apple*** map is upside down. We do it that way because it matches what you see when you examine memory using the Monitor. It takes only a few minutes of trying to match what you see in the Monitor's memory display with what you see in a standard memory map to realize that it's the standard maps that are upside down.

Since the first string is stored right next to DOS and the second next to the first, the order in which they appear in memory is backward from the order in which they are used. Note in the right-most bar of Figure 1 that our first string, "WITCHES", appears as the last word in the bar.

If line 20 of our famous little program said *print a$*, Applesoft would look in the variable table until it found the entry for a$, then would use the length and address stored there to retrieve "WITCHES".

Why the Garbageman takes so long. Ah, but the next command isn't *print*, it's *goto 10.* Our input statement is repeated. In response to the second "WHAT?", you type in "GOBLINS". Applesoft now stores "GOBLINS" in unused memory, right next to "WITCHES", and changes the length and address for a$ in the variable table.

Applesoft keeps track of where the edge between empty memory and the string storage area is by means of a pointer kept at bytes $6F-70 (111-112). This pointer is called *fretop* in the Applesoft literature. It, too, is shown in Figure 1, along with several other interesting pointers. Whenever you enter a new string, Applesoft refers to fretop to figure out where to store the string.

Since your program has changed the value of a$ from "WITCHES" to "GOBLINS", Applesoft correctly assumes you are no longer interested in "WITCHES". The word "WITCHES" is still in memory, but because its length and address have been overwritten by the length and address of "GOBLINS", Applesoft can no longer find it. "WITCHES" is now an *inactive* string.

"WHAT?" You type in "BROOMSTICK" and a couple thousand similar words. Eventually Applesoft will go to store a word you have typed in and find the strings have bumped into the variable tables — there is no more empty memory left. At that point Applesoft calls on the Garbageman to examine the variable table, find all the "active" strings, and consolidate them down at the DOS end of memory. After the consolidation, fretop will be changed to point to the new boundary between the active strings and the rest of memory. All the memory held by "inactive" strings will be freed and made available for reuse.

The Garbageman has a lot of work to do. He can't just go in, find the first active string in the variable table, and move it next to DOS. If he did, he might overwrite an *another* active string. So what he does is search through the *entire* variable table and find the active string that is already closest to DOS. Then he moves it into the first position in the string storage area.

This happens in a flash with the little program we've been using. Since there is only one string variable being used (a$), the Garbageman quickly finds its active value, moves it, and disappears. You probably wouldn't even notice he had stopped by.

But programs that have a lot of string variables — particularly programs with large string arrays — really slow the Garbageman down. For *each* variable in the table, he must look through the *entire* variable table once. The more variables in the table, the more times he must look *and* the longer it will take to look through it each time.

A demonstration. If you'd like to measure the effects of the Garbageman yourself, try this little program:

```
10  I=250
20  DIM G$(I)

30  FOR X=0 TO I
40  : G$(X)=STR$(X)
50  : PRINT G$(X)
60  NEXT

70  PRINT "COLLECTING GARBAGE"; CHR$(7)
80  IF PEEK(978)>=190 THEN X=FRE(0) : REM Applesoft/DOS 3.3
85  IF PEEK(978)=190 THEN PRINT CHR$(4);"FRE" : REM ProDOS
90  PRINT "GARBAGE COLLECTED"; CHR$(7)
```

Line 10 determines how many strings our program will have. Line 20 dimensions

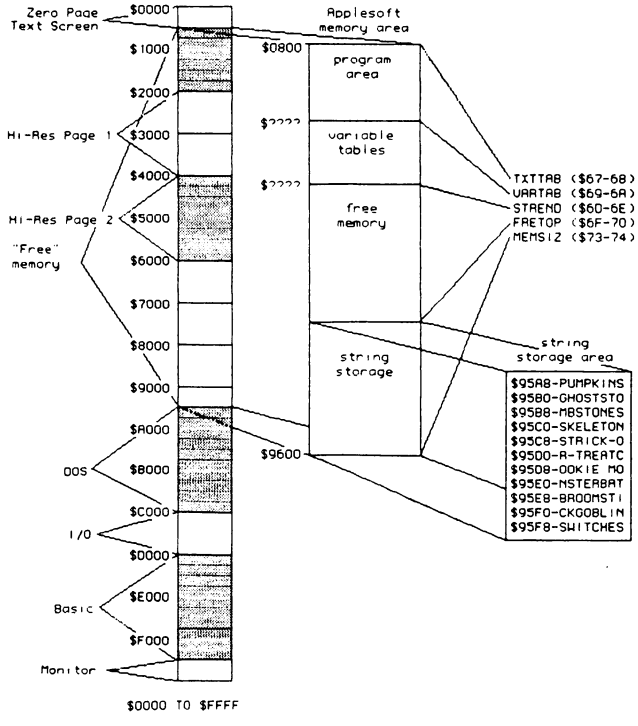


Figure 1. Applesoft string storage

80-Column Puzzle

Whenever you turn on the Apple //e or //c 80-column card with a *pr#3*, the screen is cleared. This is great when your program begins, but what if your program later turns on a printer with a *pr#1*? To return to the screen after printing do you have to destroy what's already there with another *pr#3* command?

Answer on page 6.

a string array of length I. Lines 30 through 60 fill the array with strings. Lines 70 through 90 allow us to measure how long the Garbageman takes to clean up this many strings.

You may be somewhat puzzled by lines 80 and 85. To understand these two lines, you first have to know that ProDOS includes its own Garbageman. He is quite a bit faster than the one built into Applesoft, as we'll soon see.

A program can tell if it is running under ProDOS or DOS 3.3 by taking a peek at byte 978 ($3D2). If this byte is equal to 190, then ProDOS is active. If it is anything else, then you are running under DOS 3.3.

If you are running with DOS 3.3, you can call the Applesoft Garbageman and ask him to clean up with the command *x=fre(0)*. To call the ProDOS Garbageman, however, you have to use a DOS command. Thus, in line 85, we say *print chr$(4);"fre"*.

After typing in the program and running it, increase the value of I in line 10. This will increase the number of strings in the program. Garbage collection will take longer and longer. Here are some sample timings:

number of strings	Garbage collection time in seconds DOS 3.3	ProDOS
250	5	0.1
500	19	0.2
1000	114	0.5

Speeding up the Garbageman. Back in January 1981, *Call -A.P.P.L.E.* published a program by Randy Wiggington (one of the very first Apple employees—currently famous as the author of *MacWrite*) that could make the Garbageman fly. The program worked by finding the *16* active strings closest to DOS on each scan of the variable table, rather than just the closest *one*. Rather than searching through the entire variable table once for each variable, the new technique scanned once for each 16 variables.

Apple's programmers used a similar technique in the ProDOS Basic.system software. That's why the ProDOS Garbageman is so much faster than the one built into Applesoft. Thus, the obvious way to escape the lulls caused by Applesoft garbage collection is to use ProDOS rather than DOS 3.3.

A second way is to incorporate Wiggington's 505 byte machine language program into your DOS 3.3 program and call it every once in awhile—just often enough to keep Applesoft's own Garbageman at bay.

Both of these solutions have some problems, however. Lots of people still seem to prefer DOS 3.3. ProDOS does give many new features but they come at the expense of memory. ProDOS requires users to have 64K in their machines but leaves less program space than 48K DOS 3.3 does. You could stick with DOS 3.3 and use Wigginton's program, but it is long, hard to find unless you know where to get back copies of *Call -A.P.P.L.E.*, and is copyrighted by Apple Computer, Inc.

Selective String Preservation. There is, however, another way to foil the Garbageman. This technique was developed by Clay Ruth and was described in the August 1982 *Call -A.P.P.L.E.*. The technique, which is relatively easy to implement, is called *selective string preservation*. Using this technique, you take out the garbage yourself by making a simple *poke*. Programs that use this technique avoid calls to the Garbageman completely and thus run even faster than ProDOS-based programs.

To see how the technique works, let's imagine an elementary order-entry program for a pumpkin factory. Each time the factory receives an order, the name and address of the purchaser is entered. Then the program asks for a pumpkin identification number (they come in many different sizes, shapes, and shades) and the quantity ordered. The program automatically looks up the identification number to figure out if that kind of pumpkin is available and its price.

Once the first item is found, the program asks for another pumpkin identification number (we hope it's a large order for many different kinds of pumpkins). The process continues until there are no more items, at which point the program saves the entire order in a disk file and prints an invoice and shipping papers. Then the program starts over and asks for the name and address on the next order.

This program, and most other database-type programs like it, involve two kinds of string variables. First there are *permanent* string varables. These are variables that are assigned a value only once. They may be referred to over and over again, but they don't actually vary. The DOS *d$* is a good example of a string variable that is assigned once, is never again changed, but is used throughout most programs.

In our order entry program, we would also have a large, permanent, string array holding pumpkin identification numbers. These would be strings like "RX-11-Pink" or "JC-0-Light". Whenever a pumpkin ID was entered, we would scan the array to make sure that the ID was valid. The ID's position in the array could also be used as a record number for a random-access inventory file.

In addition to permanent string variables, our pumpkin program would use a number of *temporary* string variables. These are variables that are assigned one value on the first order, a completely different value on the second order, and so on. After an order has been saved on disk and printed, the strings associated with these variables will not be used again. These temporary string variables hold information such as the name and address on the order and the ID numbers and prices of the pumpkins.

Any software similar to our order-entry program is exactly what the Garbageman loves to muck up. After entering an order or two, all available memory would be filled up.

Garbage collection would be slow because the string array holding pumpkin identification numbers alone would typically have 500 or more items in it. Not even counting all the other strings in the program, the Garbageman would get to scan the 500 array strings 500 times. Your computer will lock up for what seems like forever.

Beating the Garbageman to his own game. In the selective string preservation technique, the programmer takes care to make sure all the strings that will never be changed are assigned values first. This will pack them all together in the area of memory right next to DOS. Then the programmer takes a peek at fretop.

The first order is taken. Temporary strings are placed in memory, between the variable tables and the permanent strings. When we are finished processing the first order, we poke the value previously found in fretop back into fretop. This tricks Applesoft into storing the next group of temporary strings right over the top of the first group.

The second order is taken. Its temporary strings are placed in memory, again beginning right next to the permanent strings. At the end of the second order, we again poke fretop with its original value.

Using this technique, we constantly force new temporary strings to overwrite the old, no-longer needed ones. Consequently, the strings never fill up your Apple's memory and the Garbageman is never called. Here's what our order-entry program, which uses the technique, might look like:

```
100  GOSUB 1000 : REM initialize program
110  DIM IN$(500) : GOSUB 1100 : REM load index array
120  FT = PEEK(111) + PEEK(112)*256 : REM save value in fretop

130  GOSUB 2000 : REM get name and address
140  GOSUB 2100 : REM get items
150  GOSUB 3000 : REM save order on disk
160  GOSUB 3500 : REM print invoice, etc
170  POKE 112,FT/256 : POKE 111,FT - PEEK(112)*256 : REM fix freetop
180  GOTO 130 : REM do next order

     ...program subroutines start here
```

The important lines are 120 and 170. Line 120 saves the value in fretop after all permanent strings have been assigned. Line 170 pokes this value back into fretop when it's time to clear all the temporary strings.

Implementation tips. If you decide to use this technique, here are some tips. First, use great care in splitting the permanent strings from the temporary strings. In long, complex programs, you may have to use more than just two string groups. For example, you may have completely-permanent strings, semi-permanent strings, and temporary strings.

In this case you would save the fretop value twice, in different variables. This way you could delete only the temporary stings by poking fretop with one of the values and you could delete both the temporary and the semi-permanent strings by poking in the other value.

Don't try to poke new values into fretop until you have all other bugs out of your program. If you change fretop at the wrong place in your program, very strange things can happen. As Ruth pointed out in his original article, it is helpful to know that the pokes to fretop, and not some other bugs, are to blame.

When the selective string preservation technique is to blame for the bug, here's what happens. Your program tries to use a variable that was previously given a string value. But instead of the assigned string, the variable contains gibberish, or worse. This is because the original string assignment was placed in the temporary strings area and was later overwritten.

It is easy to get mixed up and think that since you assigned a value to a variable before peeking at fretop the first time, you can later safely change the value. It kind of seems like the new value should get stored in the permanent string area. But it won't. Remember that *the values in your permanent strings cannot be changed.* If they are, the new value will be stored in your temporary string area, and the old value will become wasted memory space.

One way you can get around this is by assigning values to string variables right inside your program. For example, if you have a line in your program such as *1000 f$="scary stuff"*, the pointer in the variable table will point at the string where it appears *in your program*. The string "SCARY STUFF" will not be moved into either the premanent or the temporary string storage areas.

This only works for program strings assigned with quotation marks. For example, *d$=chr$(4)* will put a control-D in the string storage area; *d$="" : rem invisible control-d between quotes* will not.

If you are using the selective string preservation technique, avoid the *fre(0)* command. This command calls the Garbageman and *asks* him to attack. What we are trying to do here is haul the garbage out ourselves, thereby completely avoiding the Garbageman's time-wasting visits.

Once you get the selective string preservation technique down, you'll find it quite helpful for those string- and disk-intensive data base programs you are writing. The Garbageman will no longer be *your* nemesis.

Ask (or tell) Uncle DOS

Life and times

I subscribe to a Time-Life newsletter about Apples that costs $48 a year. How will you and Weishaar stay in business with a price only half that?

Wiley Catt
Okefenokee Swamp

The October issue of Time-Life's letter listed seven executives and nineteen contributors. The committee that puts together ***Open-Apple*** *is smaller—just Weishaar and six or seven of us imaginary creatures—so our overhead is lower. Time-Life sells its letter to the U.S. business community, which is used to paying too much for computer-related services.* ***Open-Apple's*** *market, on the other hand, is the world-wide Apple user community, which demands getting its money's worth.*

Save the trees

Why is ***Open-Apple*** printed on such light-weight paper?

P.T. Bridgeport
Okefenokee Swamp

It saves postage (particularly to subscribers living outside North America), space in your file drawer, and gives us a distinctive feel (you'll never get ***Open-Apple*** mixed up with *Byte*, for example). The newsletters will be protected by envelopes in all subscription mailings.

From cotton to apples

Open-Apple looks interesting and I am considering a charter subscription. But could you please tell me Weishaar's qualifications as a newsletter publisher?

Howland Owl
Okefenokee Swamp

Weishaar (pronounced wise-ar) is a professional journalist who has been working with computers since he was in college. He received a masters degree in journalism from the University of Kansas in 1975. He then worked at Commodity News Services, an electronic news source for commodity traders. At CNS Weishaar started and managed a group of four weekly newsletters. Later, as the company's Managing Editor for operations, Weishaar was heavily involved in a changeover of the company's entire editorial department from typewriters and teletype machines to computers and terminals. He was also involved in the development of new electonic terminals for CNS customers. The terminals replaced teletype machines and early dot-matrix printers.

Weishaar bought an Apple II in 1980. A year later he left CNS to devote full time to the potential of the Apple. He is an accomplished Basic and assembly language programmer. He has developed two programs, **ProntoDOS** *and* **Frame-Up**, *that have been published by Beagle Bros. Both programs appear frequently on best-seller lists of Apple utility programs.* **ProntoDOS** *was chosen as one of the best new programs of 1983 in* Softalk's *annual reader poll (*Softalk, *April 1984, page 73).*

In addition to programming, Weishaar wrote Softalk's *monthly* **DOStalk** *column from April 1983 through the final issue of* Softalk *in August 1984. During that period he also wrote articles (***DOS Be Nimble, DOS Be Quick**, *March 1983;* **Breaking the Floppy Barrier: An Introduction to Apple's ProDOS**, *January 1984) and was one of the* Softalk Sages *who answered reader's questions in the* **If-Then-Maybe** *column.*

Weishaar is a member of the Apple Bits Users Group *in Kansas City. He is frequently invited to make presentations to groups of Apple users.*

80-column puzzle answer

(Question on page 6.)

To return to the 80-column card without clearing the screen do this:

```
DOS 3.3
   POKE 54,7: POKE 55,195 : CALL 1002

ProDOS
   PRINT D$;"PR#A$C307"
```

This trick reconnects the 80-column firmware at its *warmstart* address ($C307), which doesn't clear the screen. With DOS 3.3, you can safely do this by poking the warmstart address into the page-zero output hooks and calling the page-3 vector table routine at $3EA (1002), which forces DOS to use the current contents of those hooks for future output.

With ProDOS, this same trick can be accomplished much more easily because the *a(ddress)* parameter can be used with the ProDOS *pr#* and *in#* commands.

By the sea

Do I need to watch the SuperBowl this year? I missed Apple's *Big Brother* ad last year and my life has been a mess ever since.

Churchy LaFemme
Okefenokee Swamp

Things weren't real swell around here, either, though 1984 wasn't as bad as some predicted. Last year I was in the bathroom during Apple's ad and I, too, missed it. Do you really think that's the root cause of our troubles? I predict that Apple will buy another SuperBowl spot this year and that it will once again be more interesting than the game. You read it here first.

Where is the beginning?

I'm interested in doing some interfacing tricks with my Apple, such as hooking two Apples together and connecting stuff to the game port. However, I am a complete beginner at electronics. Where should I begin?

Grundoon Groundchuck
Okefenokee Swamp

First of all, if you don't know how already, you have to learn to solder (pronounced sodder). Soldering is a technique for connecting electronic components and wires together using heat and a tin-lead alloy. Soldering provides a good electrical connection that is also physically strong.

In addition to its use in electrical work, solder is also used in many other situations. The gutters on your house are probably soldered. However, if you know all about non-electrical soldering, you know just enough to be dangerous. Electrical soldering requires both smaller soldering irons (the device that provides the heat) and a different kind of solder (the tin-lead alloy).

Heathkit sells a self-instructional kit on soldering. It costs around $20. It includes an excellent book and some electronic parts you can safely destroy while learning. If you're interested in the hard side of the Apple II, this kit is a real good starting place.

If you don't have a soldering iron, don't buy one until you need it. The first part of Heathkit's book describes the various kinds of irons that are available and what kind you need. Radio Shack sells adequate ones.

Here at ***Open-Apple*** *we'll help you out with a column called* Bus School. *It will be about Apple hardware and techniques for connecting your Apple to the world. Watch for it.*

Replacement reading material

Now that *Softalk* is gone, what Apple magazines do you read and recommend?

Mam'selle Hepzibah
Okefenokee Swamp

InfoWorld, *while not an Apple-only magazine, is my favorite. It's a weekly and concentrates on news in the microcomputer industry. Its editors have a healthy respect for Apples; you won't feel short-changed reading this one. In addition to news,* InfoWorld *publishes lots of reviews and a couple of very good columnists.*

Call -A.P.P.L.E. *and* Apple Assembly Line *are my favorite Apple-only publications. When you read these you feel like you're among friends—people who have Apples, use them, and are trying to share their discoveries with all of us.*

Hardcore Computist *is an Apple-only magazine with a single-minded devotion to one of today's most popular hobbies—defeating copy-protection schemes.*

Nibble *has always been a whole-heartedly Apple magazine. It is targeted toward new users, however. For some reason, I often have the feeling that many of its articles are written by minors.*

A+ *and* inCider *have always struck me as opportunist publications. Both are late-comers from publishers who first had other computer magazines. Most articles appear to be written by either CP/M holdouts or professional free-lance writers who do articles on Commodores one day and Apples the next.*

If your main interest is the advertising, A+ *now appears to be the thickest Apple magazine.*

Variable recovery

Why is it that the values of all variables are cleared whenever the most minor change is made to a line of an Applesoft program residing in memory?

I have worked with other machines that let you change a program line, issue a *goto* to resume processing, and have all variable values still intact. This is a big aid in debugging a program since you need not start back at the beginning with a *run*.

Is there a *poke* or some other technique that will restore pointers or values for variables in Applesoft after altering a program line?

Harold F. Williams
Hutchinson, Minn.

The reason Dr. Basic clears all the variables is that your program and its variable tables are adjacent to each other in memory. (see Figure 1 on page 4 of this newsletter.) When you edit a program, the memory space used by the program listing itself gets either bigger or smaller. To keep the variables intact, they would all have to be moved to keep them adjacent to the program.

The other Basics you've worked with probably use a different memory allocation scheme so that the problem doesn't occur.

If you are using ProDOS, however, there is a solution. Before editing your program, issue the store *command. This will save the current values of all*

variables in a file on your disk. After editing, issue a restore command to bring the variables back into memory. Goto (not run!) will then resume processing with all variables intact.

Putting RWTS in solitary

Is there any way to edit DOS 3.3 so only the RWTS subroutine and boot up procedure are left? Would this free up space on the disk? Also, on some disks that I boot up, the cursor flashes, but on others, it doesn't. I have searched the DOS manual, but have found no information on either subject. I would be grateful if you would answer these questions.

Albert Ting
Bellevue, Wash.

A flashing cursor indicates you are using Apple's 40-column firmware. A non-flashing cursor indicates you are using Apple's 80-column firmware (activated by programs on some of your disks via pr#3). Note that the 40-column firmware cannot display 80 columns, but the 80-column firmware can display either 40 columns (press esc 4) or 80 columns (press esc 8).

To edit DOS down to just RWTS, put a LDA $C088,X JMP $FF69 at $B70E and initialize a new disk (from the Monitor enter B70E:BD 88 C0 4C 69 FF). When you boot this new disk, RWTS will be loaded normally, but then our new instructions will turn off the drive and jump to the Monitor rather than continue loading the rest of DOS. This effectively frees up sectors 10 though 15 on track 0 and all of tracks 1 and 2, however, you will have to unmark the VTOC's free-space bit map on the newly-initialized disk yourself. If you don't know how to do this, see the March 1984 **DOStalk**.

Running without filenames

I need to create a DOS system that would allow me to simply boot up a disk and run a machine language program at a certain sector on a given track. I'd also like to be able to load other programs, given a track and sector.

I wish to do nothing more. No filenames, no file directory, just what I have said. I have tried this before, but soon found I just didn't have the necessary know-how to do it.

Jeff Biggus
Glen Ellyn, Ill.

The easy way to run a machine language program on start-up is to modify the DOS boot routine at $B700-$B749 so that it loads your program instead of the DOS command interpreter and file manager (see previous letter — but don't do what it says there*).*

To do this, start by storing your main program on a disk in ascending sectors. For example, if your program is $1200 bytes long, use a disk zap utility to store the first 256 bytes in track 1/sector 0; the second 256 bytes in track 1/sector 1; and so on. When you've filled track 1/sector 15, go to track 2/sector 0 and continue until your entire program is on the disk.

Now, to get this program off the disk and back into memory, use the disk zapper to change the number at $B7E0 (track 0/sector 1, byte $E0) to the number of sectors *used to store your program; the number at $B7E7 (track 0,/sector 1, byte $E7) to the high-byte address where your program's final sector should be loaded (low-byte of address must be zero); the numbers at $B715 and $B71A (track 0/sector 1, bytes $15 and $1A) to the numbers of the track and sector where that final sector is stored. At $B73B-$B73D (track 0/sector 1, bytes $3B-$3D) put a jump ($4C) to your program's starting address.*

For example, if your program starts and loads at $2000 and is $1200 bytes long, put the stuff from $2000-20FF at track 1/sector 0; $2100-$21FF at track 1/sector 1; $2F00-$2FFF at track 1/sector F; $3000-$30FF at track 2/sector 0; $3100-$31FF (the end of the program) at track 2/sector 1. Then put $12 (the number of sectors) at $B7E0; $31 (the high-byte address of the program's final sector) at $B7E7; $02 and $01 (the track and sector of the program's final sector) at $B715 and $B71A. At $B73B put 4C 00 20 (jmp $2000)

The program will be loaded and run automatically when you boot. Your program can load and run other programs, if they are stored on the disk using the same ascending-sector scheme, by making the above changes in memory, *loading the X register with what's at $B7E9 (the slot number of the disk drive times 16), and jumping to $B700.*

HeDaP heaven

All this talk in the computer magazines about the "sorry" state of computer documentation is disgusting. Users act as if they have a Constitutional *right* to information about how computers and software work. But as any student of the history and traditions of data processing knows, this information has always belonged solely to the Heros of Data Processing (HeDaPs).

There is nothing new about the notion that an elite, closed group of people should control the technology of the time. That's how witch doctors did it and what's good enough for them is good enough for us.

Just imagine the sorry state the world would be in today if users had had access to witch doctor information 10,000 years ago. A careless user might have ridden a spirit hard, put it away wet, and contaminated all the spirits. If anyone could be a witch doctor, then everyone could be a witch doctor; and then no one would really be a witch doctor; and then where would we be?

I am not a HeDaP myself. But I do acknowledge that theirs is the power and the glory. Regretfully, I must admit I didn't always understand that. There was a time when it seemed to me that computers *couldn't be* any harder to start than lawnmowers. I argued constantly with one big HeDaP in particular about who should be allowed to use the power of our company's computer. He patiently explained to me many times how damaging us common bunglers can be to big computer systems.

Unconvinced and unrepentant, I snuck into an IBM seminar for HeDaPs. The seminar was about a machine called the DisplayWriter. Around the seminar room stood unique beings — half from the world of technology and half from the world of commerce — called Sales Engineers. The Sales Engineers sat each HeDaP down in front of a machine. A little stick man made of letters popped up on the screen and started to dance. Curious, I pressed a key on the machine's keyboard. The man stopped dancing. A Sales Engineer approached but neither he nor any of his colleagues could make the man dance again. My machine was dead. I had, with one uninformed keystroke, killed a little dancing man — to say nothing of a computer priced at 10 grand. I was embarrassed and sorry and knew from that moment on that computers must be protected from the masses.

Perhaps the greatest weapon we have in this fight is obscure documentation. The early materials that accompanyed CP/M, for example, were so good that one HeDaP told me CP/M stood for *Conspiracy to Protect the Ministry*. I believed him.

Oh, the users got a little uppity there for awhile when those two garage engineers developed the Apple II. I must admit it was frightening to see tens of thousands of people learning how to use and program a computer with the books that came with the machine. Fortunately, it was only Basic they were learning.

It is my pleasure to assure you that the Heros of Data Processing have now gained control even at Apple. Those books are no longer standard equipment. They cost $50 extra. That should keep the users from nosing around too much.

Even better, however, is Apple's policy with its MacIntosh. You have to be a certified HeDaP to get that computer's $150 programming manual. And the great joke is that even if users get a copy it won't help them; they need a Lisa to write programs — that kills me. The MacIntosh is a shoe-in for the HeDaP machine of the year.

Believe me, I know how serious the situation is when HeDaPs lose control. I've seen *War Games*; I've seen *Tron*; I know what happens when you let untrained individuals loose around a computer.

I sleep soundly at night knowing that the HeDaPs have the users on the run. However, I must admit I would sleep even better if documention writers would write more like scholars and a little less like newspaper hacks. Even politicans know more about proper English than most technical writers do.

For good examples of scholarly writing we need turn no further than the platform-writing committees of last summer's political conventions. The Democrats had a gem that went like this, "The Democratic party opposes quotas which are inconsistent with the principles of our country." Isn't that beautiful language? It is absolutely impossible to tell whether the Democrats think all quotas are un-American or if they think some are and some aren't and they oppose only the ones that are.

For their part, the Republicans wrote, "We...oppose any attempts to increase taxes which would harm the recovery..." Do they oppose all taxes or only those that are harmful? You can't tell. This is what scholarly writing is all about. Obscurity is the foundation on which elitism is built.

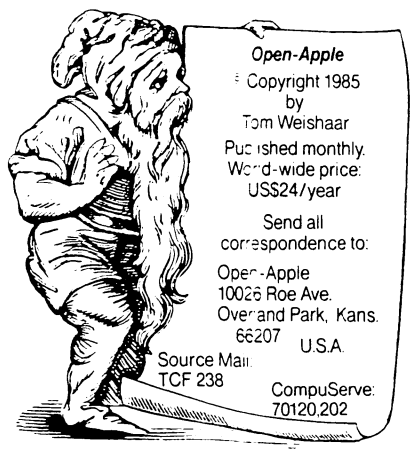
According to a report in the *Wall Street Journal*, a journalism professor from the University of Kansas told a group of newspaper editors this summer, "Outside this room, there are only three or four people left who know the difference between 'which' and 'that'." Don't you agree it's scary to think that one or two of them might be documentation writers? (I don't think we have to worry that any of them are politicians.)

This unbearably clear-minded professor, whose name is John Bremner, insists writers should always use "that" to introduce a clause that is essential to the meaning of a sentence. "Which", which should be preceeded by a comma, is reserved for clauses that add incidental information, according to Bremner.

I recommend that any technical writers who know the difference between which and that turn themselves in for reprogramming. Then we'll only have a few newspaper editors and this guy Bremner to worry about.

Deacon Muskrat
Okefenokee Swamp

Indeed, documentation that always has commas in front of whiches is a danger to HeDaPdom, which suits us just fine.



Become a Charter Subscriber Right Now!

Ordering Information

Open-Apple is published monthly. A one-year subscription costs $24. All subscriptions are fully guaranteed—upon cancellation, unsatisfied subscribers will receive the unused portion of their subscription payment in full.

Special Offer for New Subscribers: Do Not Send Money Now. We will bill you with your first issue. If you decide ***Open-Apple*** doesn't interest you, simply write "cancel" on our bill and return. You will owe us nothing.

International Subscribers: ***Open-Apple*** is sent air mail to all subscribers outside North America. *No additional subscription fees are required.* We offer a single world-wide price because we want to be the gathering place of the world-wide Apple community.

Here's How to Order: Simply fill out our form and mail it. You will receive our current issue and a bill for your first year's subscription by return mail.

Cut out or photocopy our coupon and mail today to:

Open-Apple
10026 Roe
Overland Park, Kans. 66207
U.S.A.

☐ Yes, sign me up for 12 monthly issues of ***Open-Apple*** and bill me $24 when you send my first issue.

☐ No, and I've written why below.

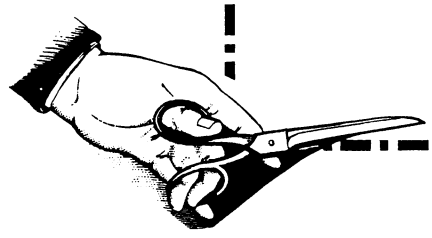
Name _____

Adr _____

_____

_____

Comments/Questions/Jokes:



Open-Apple
10026 Roe
Overland Park, Kans. 66207
U.S.A.

BULK RATE
U.S. POSTAGE
PAID BY
Open-Apple

Releasing the power to everyone.