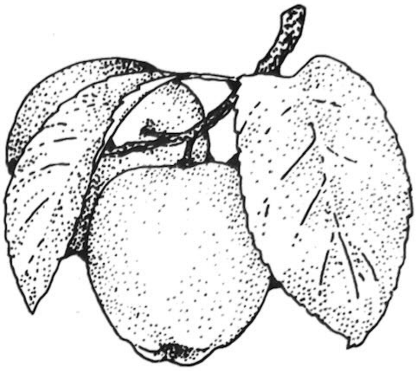
APPLE SOFTWARE PROTECTION DIGEST



$3.00

Vol. 1 No. 4 1986

Apple Software Protection Digest

Publisher & Editor, Jules H. Gilder; Contributing Editor, J. Scott Barrus. Published monthly at $24 per year by Redlig Systems, Inc. (718) 232-8429. Reprints of prior issues available at $3 each. Printed in the U.S.A.

HELP SPREAD THE WORD

As you can see we've gotten issue #4 of the digest out and cut down on the delay between issues. Things are starting to finally run a little smoother now and we've just started doing the typesetting for the newsletter in-house. This should significantly cut down on our costs and more importantly, make it easier for us to get the newsletter ready for publication. By going in-house we've eliminated a lot of delays that were caused by the fact that we had to travel some distance to get to the typesetter. It also precluded the possibility of turning articles into typeset material the same day. All that has now changed.

Now that we're back on track, we'd like to move ahead somewhat and try to build the subscription list, and we're asking you to help. Show the digest to your friends who have Apple // computers (or clones like the Laser 128). I think that once they see it, they'll like it and want to order it for themselves.

Get a free digital watch pen

We think you'll tell your friends about us because you like our publication and they will too. But, as a little extra incentive to you, we'll send you a beautiful, free digital watch pen for every new subscriber who subscribes to the digest and mentions your name. Obviously, a new subscriber can only mention the name of one already existing subscriber, but there's no limit on the number of new subscribers who can mention your name. Thus, if ten people subscribe because they heard about us from you, you'll get ten digital watch pens. This is a limited offer and will only be good for one month, so hurry up and get those subscriptions in quickly.

Special group discounts

In order to encourage group purchases of the digest, we have established a special discount of 16.7% whenever ten or more subscriptions are ordered at the same time. This means that the normal $24 subscription will cost only $20 a year. In order to qualify for this special rate, at least ten subscriptions must be ordered at once and they must be paid for with one group check (or charge). The free pen and discount offer cannot be combined.

Let's hear from you

One of the most useful features of any publication is the section that provides feedback from the readers. This is the Letters section. This is the place where you can air your opinions (even if we don't agree with them) and ask for help. It's also the place where we find out how we're doing. Certain things we know, like we're late and should try to improve our schedule. Other things we don't, like how do you find the material presented? Is it too simple or too complicated? Is there something you don't like? Is there something you'd like included? Let us know, we'll try to accomodate you.

Jules H. Gilder
Publisher & Editor

Crack Index

In order to make life just a little more convenient for you, each issue of **Apple Software Protection Digest** will contain a list of all the programs cracked so far, and what issues those cracks appeared in. This will save you from going through all past issues of the digest in order to find a particular program.

Applewriter //e - Vol. 1, No. 3, p. 3
Bookends - Vol. 1, No. 1, p. 7
Financial Cookbook - Vol. 1, No. 3, p. 6
Hayes Terminal Program - Vol. 1, No. 3, p. 11
Homeword - Vol. 1, No. 2, p. 9
Homeword Speller - Vol. 1, No. 2, p. 9
Microwave - Vol. 1, No. 3, p. 11
PFS Series - Vol. 1, No. 2, p. 7
Print Shop - Vol. 1, No. 1, p. 10
Print Shop Companion - Vol. 1, No.2, p. 6
Sensible Grammar - Vol. 1, No. 2, p. 7
Time Is Money - Vol. 1, No. 2, p. 9

BUGS

Although we spend a lot of time testing and rechecking all the information we present here, every once in a while a problem will still crop up. As soon as I find out about it, I'll let you know in this column. It is my sincere hope that this column will be missing from most issues, and very short in those issues in which it is included.

Print Shop Copier

Many of you have written in and complained of problems with the Print Shop Copy program from the first issue of **ASPD**. Interestingly enough, there were almost as many solutions to the problem as there were writers about it. All of you however treated the symptom, and thus while each of these fixes worked, none got down to the real cause of the problem, which lies in line 100 of Apple's original COPYA program.

This line sets up the start of the copy buffer. Since we've added some lines to the program the space between the end of the program and the beginning of the buffer — which is needed to store program variables — has been reduced considerably and thus causes the program to crash frequently. I had actually already discovered this when I wrote COPYP in the last issue, but didn't have time to get it into the issue. The problem can be eliminated altogether by simply adding the following line to the Print Shop Copier program:

```
100 POKE 715, PEEK (110) + 2
```

I'm sorry for any inconvenience I may have caused you.

Print Shop Companion

There was an omission in the listing of the parameters to be used to back up the program. Line 72 should have read: **POKE 863,34.** Thus line 72 should look like this:

```
72 POKE 863,34
```

This will tell the program not to copy the last track and eliminate I/O Error messages.

Tutorial — Part II

On page 8 of the last issue, where the modifications are listed from making a protected DOS, the second occurence of line 1 was incorrect. It should read:

```
1. B955:D5 N B95F:AA N BC7A:D5 N BC7F:AA
```

This will now restore normal DOS.

Letters

Dear Editor:

I continue to enjoy your magazine. I don't care if the issues are late, just keep them coming! I will certainly renew my subscription when it is due. I wonder if anyone has a crack for the program, "The Game Show" by Advanced Ideas. It involves reading half tracks and the disk is a combination of DOS 3.3 and DOS 3.2 and it is beyond my skills so far.

As far as your **Cracks Wanted** list is concerned, Sensible Speller was covered in Hardcore Computing, issues 9, 10, 11, 13, 15, 16 and 23. It was also covered in the 1985 Pirate's Harbor disk. Crush, Crumble and Chomp was in Hardcore's issue 7. Wizardry was covered in the Bootlegger's disk #2 and Hardcore in issue's 8, 20, 23, 26 and 29. Dazzle Draw was covered in the 1985 Pirate's Harbor disk and Hardcore 21 and 26. I have not seen cracks for Compres Software, Word Handler or Disk-O-Check anywhere.

Brian Symonds
Powell River, B.C.

I'm glad to here that you enjoy our publication so much. I appreciate your tolerance of our lateness. We are trying to overcome it. We have just purchased a laser printer to do the typesetting in-house and cutdown on costs and time. Parts of this issue were set on the laser printer and the entire next issue will be too. I don't know if anyone has crack "The Game Show" yet, but we'll add it to our list. Thanks for all the info on where the other programs on our list have already been cracked. That should be helpful to a lot of people. Nevertheless, we'd like to see other approaches to cracking these programs.

PROTECT AND UNPROTECT PROGRAMS WITH MUFFIN PLUS AND DEMUFFIN PLUS

Whenever you're trying to protect or unprotect programs, its always important that you have the correct tools to do the job. One of the tools that I have found very useful over the years is a program called *DEMUFFIN PLUS*. If the name of this program sounds familiar to you, it should, because it is a modification of a program that Apple gives away on its DOS 3.3 System Master diskette. That program is called *MUFFIN*, and was originally designed to bridge the gap between Apple's old DOS 3.2 operating system, and the subsequent DOS 3.3 operating system. When *MUFFIN* was run, it allowed the user to take files (of all types) that were stored on a DOS 3.2 diskette and transfer them over to a DOS 3.3 diskette.

If you've ever tried using a DOS 3.2 diskette (are there still any around) on a DOS 3.3 system, you will notice that the Apple stubbornly refuses to recognize it and prints out an **I/O Error** message to the user. The reason for this is that the diskette is formatted differently than a DOS 3.3 diskette is. To get around this problem, the software wizards at Apple Computer designed a program that could use two different RWTS (Read and Write a Track and Sector) routines. One RWTS routine would be the standard one that is in DOS 3.3 and located in memory at $B800. The other one would be the 13-sector DOS 3.2 RWTS and it would be located in memory at $1900. The RWTS routine is the core of any DOS and is the routine that makes it possible for the computer to write information to, or read information from, a diskette.

Shortly after Apple came out with its DOS 3.3 and *MUFFIN* programs, some bright programmer decided that he was going to replace the resident copy of the DOS 3.2 RWTS with a copy of the DOS 3.3 RWTS. He apparently reasoned that if he could then interrupt a running protected program by pressing the RESET key (non-autostart ROMs that dropped you into the monitor when RESET was pressed were widely available) and somehow load in this modified *MUFFIN* program which was called *DEMUFFIN PLUS*, programs could then be transferred from a diskette with the protected DOS on it to a diskette with standard DOS 3.3 on it. An that's what was done. After *DEMUFFIN PLUS* was produced, it was stored out on a cassette tape and loaded in when needed.

While most people are primarily interested in transferring programs from a protected diskette to an unprotected one, if you've created your own protected DOS (see **ASPD** Vol. 1, No. 2, p. 22) you'll want some method of transferring programs the other way, from unprotected diskettes to protected ones. That's where a new program, which I call *MUFFIN PLUS*, comes in. In the rest of this article, I will show you how to make both of these programs and how to use them.

Here's what you'll need

To make the *DEMUFFIN PLUS* program, you will need a DOS 3.3 System Master with the *MUFFIN* program on it and an Apple computer that has Integer BASIC with the Programmer's Aid ROM available. If you have a 16K RAM card (or an Apple //c or //e) and you boot your system with the System Master diskette, Integer BASIC and the Programmer's Aid ROM will automatically be loaded for you into the RAM card memory. If your Apple does not have the extra 16K of memory and you don't have the Programmer's Aid ROM installed in your computer along with Integer BASIC, don't dispare, you'll still be able to make *DEMUFFIN PLUS*.

The reason we need to have the Programmer's Aid ROM available is because there is a very handy machine language program in it that let's you relocate machine language programs. By relocate, I mean move the program from its current location in memory to another one in such a manner so that it can run at the new location without any difficulty. Those of you somewhat familiar with the Apple monitor ROM might think that you can already do that by using the memory move command that is built into the F8 ROM. That will only work under special circumstances. Most of the time, Apple machine language programs are written in such a way, that they are tied to a specific location in memory. Thus if you move the program to a new memory location, it won't work because it is still going to search for some of the information it needs at the old location. However, if as you move the program, you replace the old addresses that were referenced, with the matching ones for the new location of the program, then everything will run fine. This process of moving and updating the program and its references to specific memory locations is called relocation.

As I mentioned earlier, this special relocation program is only available in those Apples that have 64K or more of memory or Integer BASIC with the extra Programmer's Aid ROM added to it. However, to accommodate those readers who do not meet either of these conditions, I have rewritten the Apple relocation program so that it does not require the Integer or Programmer's Aid ROMs. This program, called *6502 RELOCATOR*, is listed below. This version of the *6502 RELOCATOR* program has been specially modified so that it can run wherever it is loaded into memory. It is not tied to any specific memory locations. Unless you have a machine language program on page three that has to be worked on, you'll probably find it most convenient to load and run the relocator program at address $300. To make it easy to use, I have included a short BASIC program listing that will automatically enter the *6502 RELOCATOR* program for you. For those of you who do not like to type, the *6502 RELOCATOR* is available on **ASPD Diskette #3** along with all of the others programs in this issue. The diskette is available for $15.

In order to make *DEMUFFIN PLUS*, it is necessary to move the standard DOS 3.3 RWTS from its usual location of $B800 to $1900. Since the RWTS code references specific locations within itself, it is necessary to modify those specific references before we run the program, hence the need for the relocator.

Making DEMUFFIN PLUS

The first thing you have to do once you have a copy of *MUFFIN*, is to activate the relocator program. If you have an Apple with Integer BASIC in it and the Programmer's Aid ROM, activate Integer BASIC by typing INT. If you're using my modified version of the *6502 RELOCATOR*, just RUN the BASIC version, or BLOAD and CALL the program at 768. In either case, the next step is for you to BLOAD the *MUFFIN* program. *MUFFIN* loads in at location $803.

After *MUFFIN* has been loaded into memory, you must get into the monitor mode. You can do this by typing **CALL -151**. Once in the monitor, people not using my program must type **D4D5G** to activate the *6502 RELOCATOR* program. The program is designed to work with the monitor's Control-Y capabilities. To start the relocation process, type in the following:

1900<B800.BFFF Control-Y*

In the above instructions, you don't type the word "Control-Y", but rather, you hold down the **Control** key while at the same time you press the **Y** key. Then you let both of them go. Also, don't type a space after BFFF. I've put it in here so that it's a little easier to read. Don't forget to type the asterisk (*) immediately after the Control-Y. Now that you've told the relocator which block of code is being moved ($B800-$BFFF) and where it is going ($1900), you've got to tell the relocator which portions of this block contain instructions that have to be changed, and which portion contains data that must only be moved and not updated. Let's tackle the instruction segment first. Type in the following line and then press return:

1900<B800.BA10 Control-Y

Next, you want to move a section of data without modifying it. This is done by typing in the following line:

.BC57M

Don't forget the period preceding the BC57M, it's very important. Finally, you want to move another section of program and have all the internal references updated. This is done by typing:

.BFFF Control-Y

At this point, you might want to consider saving this file out to the diskette because we'll need it again later to make up the program *MUFFIN PLUS*. You can save it out by typing:

BSAVE MUFTEMP,A$803,L$1900

Now that all of the DOS 3.3 code has been properly relocated and you've saved a copy for use again later, the next thing to do is to modify the *MUFFIN* program itself so that it will display the correct title and so it will work as we

```
                      1000 ***************************************
                      1010 ***                                   ***
                      1020 ***          6502 RELOCATOR           ***
                      1030 ***                                   ***
                      1040 ***           by S Wozniak            ***
                      1050 ***                                   ***
                      1060 ***       Modified to eliminate       ***
                      1070 ***    the need for the Sweet-16      ***
                      1080 ***       interpreter and to be       ***
                      1090 ***    completely relocatable by      ***
                      1100 ***                                   ***
                      1110 ***         JULES H. GILDER           ***
                      1120 ***                                   ***
                      1130 ***       Copyright  (C) 1982         ***
                      1140 ***       All Rights Reserved         ***
                      1150 ***                                   ***
                      1160 ***************************************
                      1170 *
                      1180 *
                      1190 * Equates
                      1200 *
0000-                 1210 BEGIN      .EQ $0
0002-                 1220 R1L        .EQ $2
0002-                 1230 FROMBEG    .EQ $2
0004-                 1240 FROMEND    .EQ $4
0008-                 1250 TOBEG      .EQ $8
000B-                 1260 INST       .EQ $B
002F-                 1270 LENGTH     .EQ $2F
0034-                 1280 YSAV       .EQ $34
003C-                 1290 A1L        .EQ $3C
0042-                 1300 A4L        .EQ $42
0100-                 1310 STACK      .EQ $100
0200-                 1320 IN         .EQ $200
03F8-                 1330 CNTRLY     .EQ $3F8
F88E-                 1340 INSDS2     .EQ $F88E
FCB4-                 1350 NXTA4      .EQ $FCB4
FF58-                 1360 RETURN     .EQ $FF58
                      1370 *
                      1380            .OR $300
                      1390 *
                      1400 * Do a subroutine jump to a known RTS
                      1410 * instruction to push the address of
                      1420 * beginning of the program +2 on the
                      1430 * stack.  Then add 36 ($24) bytes to
                      1440 * this address to get the beginning of
                      1450 * the program.
                      1460 *
0300- 20 58 FF        1470       JSR RETURN          Put program start
0303- BA              1480       TSX                 address on the
0304- BD 00 01        1490       LDA STACK,X         stack and then
0307- 85 01           1500       STA BEGIN+1         retrieve it.
0309- CA              1510       DEX
030A- BD 00 01        1520       LDA STACK,X
030D- 18              1530       CLC
030E- 69 24           1540       ADC #$24            Adjust starting
0310- 85 00           1550       STA BEGIN           address.
0312- 90 02           1560       BCC INITY
0314- E6 01           1570       INC BEGIN+1
                      1580 *
                      1590 * Set up the Control-Y vector with the
                      1600 * address of the start of the program.
                      1610 *
0316- A9 4C           1620 INITY LDA #$4C
0318- 8D F8 03        1630       STA CNTRLY
031B- A5 00           1640       LDA BEGIN
031D- 8D F9 03        1650       STA CNTRLY+1
0320- A5 01           1660       LDA BEGIN+1
0322- 8D FA 03        1670       STA CNTRLY+2
0325- 60              1680       RTS
                      1690 *
                      1700 * This is the beginning of the actual
                      1710 * relocation routine.
                      1720 *
0326- A4 34           1730 RELOC LDY YSAV            Initialize Y-register
0328- B9 00 02        1740       LDA IN,Y            Get next character.
032B- C9 AA           1750       CMP #'*+$80         Is it a *?
032D- D0 0C           1760       BNE RELOC2          No, relocate code.
032F- E6 34           1770       INC YSAV            Yes, incr. pointer,
0331- A2 07           1780       LDX #$7             Get move
0333- B5 3C           1790 INIT  LDA A1L,X           parameters of
0335- 95 02           1800       STA R1L,X           block.
0337- CA              1810       DEX
0338- 10 F9           1820       BPL INIT
033A- 60              1830       RTS
```

(c) B800-BA10 →1900
BA11-BC56 →1B11.1D56
(c) BC57-BFA7 →1D57
BFA8-BFC7 →20A8.20C7
(?) BFC8-BFFF →20C8.20FF
probably state data

1900<B800.BFFF ^Y *
.BC57 M
.BFA7 ^Y
.BFC7 M
.BFFF ^Y

want it to. This is done by typing in the following lines while still in the monitor mode.

```
1155: 00 1E
115B: D9 03
1197: A0 20
15A0: A0 D2 C5 D3 C9 C4 C5 CE
15A8: D4 A0 C4 AE CF AE D3 AE
15F7: A0 A0 A0 A0 C4 C5 CD D5
15FF: C6 C6 C9 CE A0 D0 CC D5
1607: D3 A0
161E: CD CF C4 D3 A0 C2 D9 A0
1626: CA D5 CC C5 D3 A0 C8 AE
162E: A0 C7 C9 CC C4 C5 D2
20A0: A9 1E 8D B9 B7 20 FD AA
20A8: 48 A9 BD 8D B9 B7 68 60
```

Once these lines have been typed in, you have completed the task of changing *MUFFIN* into *DEMUFFIN PLUS* and all that's let for you to do is to save it out to disk. You can do this by typing:

BSAVE DEMUFFIN PLUS,A$803, L$1900

You now have a useful utility that can be used to unprotect many programs.

Using DEMUFFIN PLUS

Now that you have this very handy tool, the next thing you have to learn is how and when to use it. *DEMUFFIN PLUS* cannot be used with all protected programs, only those that have a DOS that has not been too heavily modified. One good indication of such programs are those that display the Applesoft prompt ( ]) at some time during the boot-up process. If you see that prompt, even for only a short period of time, chances are pretty good that you'll be able to use *DEMUFFIN PLUS* with the program. If you don't see the prompt, there's no guarantee that *DEMUFFIN PLUS* won't work, but the likelihood of it working is considerably smaller.

Before you start to crack a diskette with *DEMUFFIN PLUS*, make sure you have all the tools you'll need. First, you'll need a blank, initialized diskette. Next, you'll need some way of breaking out of the program you want to copy once it has booted. This could be an Integer BASIC ROM card, an interrupter card such as the WildCard 2, or a modified F8 ROM that will drop you into the monitor when RESET (or Control-RESET) is pressed. If you have an Apple computer that will still let you use the cassette tape (those routines were eliminated from the new enhanced //e ROMs) *DEMUFFIN PLUS* can be saved out to tape by typing:

803.2103W

and reloaded from tape again when it is needed later by typing:

803.2103R

Both of these tape commands must be typed while you're in the monitor mode – that means the prompt character is an asterisk (*). If you're not going to use the cassette tape, most of the time you should be able to work around it by loading *DEMUFFIN PLUS* into an area of memory that does not get destroyed during the boot-up process.

I usually load *DEMUFFIN PLUS* into memory starting at $6000. This is a fairly safe area. Once the protected DOS has been loaded in, you should get into the monitor using any one of the techniques I discussed earlier. Once you're in the monitor, you'll have to move the *DEMUFFIN PLUS* program back down in memory where it belongs and then start the program running. You can do both of these things at once by typing:

803<6000.7900M N 803G

The spaces before and after the N are important, so don't leave them out. The command preceding the N performs the

```
                 1840 *
                 1850 * Here the next three bytes are copied
                 1860 * and examined and the length of the
                 1870 * instruction is calculated.
                 1880 *
033B- A0 02      1890 RELOC2 LDY #$2          Copy the next
033D- B1 3C      1900 GETINS LDA (A1L),Y      3 bytes.
033F- 99 0B 00   1910        STA INST,Y
0342- 88         1920        DEY
0343- 10 F8      1930        BPL GETINS
0345- 20 8E F8   1940        JSR INSDS2       Calculate the (operand)
0348- A6 2F      1950        LDX LENGTH       length.
034A- CA         1960        DEX              0=1-byte, 1=2-byte,
034B- D0 0C      1970        BNE XLATE        2=3-byte.
034D- A5 0B      1980        LDA INST         Is it Zero
034F- 29 0D      1990        AND #$D          page mode?
0351- F0 2A      2000        BEQ STINST       No, Immediate.
0353- 29 08      2010        AND #$8          Yes, clear the
0355- D0 26      2020        BNE STINST       high byte.
0357- 85 0D      2030        STA INST+2
                 2040 *
                 2050 * This section of code checks to see if
                 2060 * the address of the instruction is in
                 2070 * the range of the block of code being
                 2080 * relocated.  If it is, then the
                 2090 * address is adjusted.
                 2100 *
0359- A0 00      2110 XLATE  LDY #$0          Compare the
035B- A6 04      2120        LDX FROMEND      instruction address
035D- E4 0C      2130        CPX INST+1       with the end
035F- A5 05      2140        LDA FROMEND+1    address of the
0361- E5 0D      2150        SBC INST+2       source block.
0363- 90 18      2160        BCC STINST       It's larger.
0365- 38         2170        SEC
0366- A5 0C      2180        LDA INST+1       Compare it with
0368- E5 02      2190        SBC FROMBEG      start address of
036A- AA         2200        TAX              source block.
036B- A5 0D      2210        LDA INST+2
036D- E5 03      2220        SBC FROMBEG+1
036F- 90 0C      2230        BCC STINST       It's smaller
0371- 48         2240        PHA              It's in the range
0372- 8A         2250        TXA              so adjust it
0373- 18         2260        CLC              for its new
0374- 65 08      2270        ADC TOBEG        location.
0376- 85 0C      2280        STA INST+1
0378- 68         2290        PLA
0379- 65 09      2300        ADC TOBEG+1
037B- 85 0D      2310        STA INST+2
037D- A2 00      2320 STINST LDX #$0          Copy the fixed
037F- B5 0B      2330 STINS2 LDA INST,X       instruction to
0381- 91 42      2340        STA (A4L),Y      its new location.
0383- E8         2350        INX
0384- 20 B4 FC   2360        JSR NXTA4        Update pointers.
0387- C6 2F      2370        DEC LENGTH
0389- 10 F4      2380        BPL STINS2       End of instruc?
038B- 90 AE      2390        BCC RELOC2       End of block?
038D- 60         2400        RTS
```

memory move, while the command after it causes the computer to jump to location $803 and execute the program that is stored there. The N and the spaces are just used to separate multiple monitor commands on the same line.

Once the program is running, you'll be presented with an opening screen and a two-choice menu. The first choice is to convert the programs and the second is to quit. After selecting choice 1, you'll be asked for the source and destination drives and the name of the file you want transferred. Here you may answer with a name, an equals sign (=) or a partial name and equals sign, just as in FID. That's all there is to it.

How to make MUFFIN PLUS

Earlier we said that *DEMUFFIN PLUS* was a handy tool to use for transferring programs from a diskette with protected DOS to standard DOS 3.3. If you are developing your own protected programs however, you'll want a way of transferring your programs from a standard 3.3 DOS diskette to a diskette with your own protected DOS on it. To do this, we use another variation of the *MUFFIN* program which I call *MUFFIN PLUS*.

You start out making *MUFFIN PLUS* the same way you make *DEMUFFIN PLUS*. If you saved out the program MUFTEMP as suggested earlier, all you have to do is BLOAD it. Otherwise you have to load *MUFFIN* into memory and then you move and relocate the standard DOS 3.3 code done in memory to $1900, just as I told you how to do it for *DEMUFFIN PLUS*. Once this code has been relocated, or MUFTEMP has been loaded, you actually have a working copy of *MUFFIN PLUS*, however, to prevent you from confusing it with MUFFIN, you should key in the following lines, from the monitor mode, to change the title that is displayed when the program is run.

```
15F7: A0 A0 A0 A0 A0 CD D5 C6
15FF: C6 C9 CE A0 D0 CC D5 D3
1607: A0 A0
161E: CD CF C4 D3 A0 C2 D9 A0
1626: CA D5 CC C5 D3 A0 C8 AE
162E: A0 C7 C9 CC C4 C5 D2
```

This program should now be saved out to a diskette by typing:

BSAVE MUFFIN PLUS,A$803,L$1900

```
  1  REM BASIC PROGRAM TO INSTALL 6502 RELOCATOR
  2  REM
 10  TEXT : HOME
 20  PRINT : PRINT : PRINT : PRINT
 30  PRINT "INSTALLING 'RELOCATOR'..."
 40  FOR X = 768 TO 909
 50    READ Y
 60    POKE X,Y
 70  NEXT X
 80  PRINT : PRINT : PRINT : PRINT "INSTALLATION COMPLETE."
 90  PRINT : PRINT "TYPE 'CALL 768' TO RUN PROGRAM."
100  DATA 32,88,255,186,189,0,1,133
110  DATA 1,202,189,0,1,24,105,36
120  DATA 133,0,144,2,230,1,169,76
130  DATA 141,248,3,165,0,141,249,3
140  DATA 165,1,141,250,3,96,164,52
150  DATA 185,0,2,201,170,208,12,230
160  DATA 52,162,7,181,60,149,2,202
170  DATA 16,249,96,160,2,177,60,153
180  DATA 11,0,136,16,248,32,142,248
190  DATA 166,47,202,208,12,165,11,41
200  DATA 13,240,42,41,8,208,38,133
210  DATA 13,160,0,166,4,228,12,165
220  DATA 5,229,13,144,24,56,165,12
230  DATA 229,2,170,165,13,229,3,144
240  DATA 12,72,138,24,101,8,133,12
250  DATA 104,101,9,133,13,162,0,181
260  DATA 11,145,66,232,32,180,252,198
270  DATA 47,16,244,144,174,96
```

In operation, *MUFFIN PLUS* works the same way that *DEMUFFIN PLUS* does. The only difference is that you must have an initialized protected diskette ready before the program is run. In addition, since you are doing the protection it's an easy matter for you to first load *MUFFIN PLUS* into memory where it belongs and then modify your DOS so that it is a protected DOS. This makes it a little easier to get *MUFFIN PLUS* up and running.

If you come up with any interesting uses for either *MUFFIN PLUS* or *DEMUFFIN PLUS*, let us know and we'll pass it along to everyone else.

continued from page 11

```
  1  REM BASIC PROGRAM TO INSTALL AMPER RESTORE
 10  TEXT : HOME
 20  PRINT : PRINT : PRINT : PRINT
 30  PRINT "INSTALLING 'AMPER RESTORE'..."
 40  FOR X = 768 TO 970
 50    READ Y
 60    POKE X,Y
 70  NEXT X
 80  PRINT : PRINT : PRINT : PRINT "INSTALLATION COMPLETE."
 90  PRINT : PRINT "TYPE 'CALL 768' TO RUN PROGRAM."
100  DATA 169,76,141,245,3,169,18,141
110  DATA 246,3,169,3,141,247,3,76
120  DATA 23,3,169,174,32,192,222,160
130  DATA 0,32,88,252,185,126,3,240
140  DATA 6,32,237,253,200,208,245,165
150  DATA 103,24,105,3,133,6,165,104
160  DATA 133,7,160,1,145,103,136,200
170  DATA 177,6,208,251,152,24,105,5
180  DATA 160,0,145,103,165,103,133,6
190  DATA 165,104,133,7,169,0,133,8
200  DATA 177,6,200,208,2,230,7,201
210  DATA 0,208,241,165,8,201,2,240
220  DATA 4,230,8,208,235,200,152,208
230  DATA 2,230,7,133,105,133,107,133
240  DATA 109,133,175,165,7,133,106,133
250  DATA 108,133,110,133,176,96,166,210
260  DATA 197,211,212,207,210,197,141,141
270  DATA 194,217,160,202,213,204,197,211
280  DATA 160,200,174,160,199,201,204,196
290  DATA 197,210,141,195,207,208,217,210
300  DATA 201,199,200,212,160,168,195,169
310  DATA 160,177,185,184,178,141,193,204
320  DATA 204,160,210,201,199,200,212,211
330  DATA 160,210,197,211,197,210,214,197
340  DATA 196,141,141,141,210,197,193,196
350  DATA 217,174,141
```

PROTECTION TUTORIAL — Part III
ALL ABOUT SELF-SYNC BYTES

Last time we discussed how a track on a diskette is formatted. You may recall that we said the Address and Data fields on the diskette are separated by gaps. What we did not say was just what these gaps were composed of. They are formed by writing a series of special bytes to the diskette that are called *self-sync* bytes. Self-sync bytes get their name from the property that they have of being able to automatically bring the disk drive hardware into synchronization with the data that is stored on the diskette.

The newcomer to understanding disk drive operation might wonder why this is necessary. The explanation is simple. Data is stored on the diskette as individual bits, but must be retrieved as 8-bit bytes. The problem, however, is that when the computer starts reading information from a diskette, it has no way of knowing where a particular byte begins. When the disk drive starts to read data, it starts from whatever position the read head is located at. Since we're reading 8-bit bytes from the diskette, there's only a one-in-eight chance that it has started reading data from the beginning of a byte as we'd like it to. Without some sort of special marker bytes, the computer has no way of knowing where the start of a series of bits begins.

To clarify the situation a little, let's make believe that the following series of bits were read from the disk drive:

```
11011111   10101011   1010
   DF         AB
```

If we begin interpreting our data with the first bit, we find that the first two bytes that we've read are DF and AB. If, on the otherhand, the read head was located at a position where it would start reading data from the diskette at the second bit of the above data stream, then the data would be interpreted as:

```
10111111   01010111   010
   BF         57
```

As you can see from this very simple example, the first two bytes have now become BF and 57, a far cry from what our original data was. Thus, it becomes clear why where we begin reading our data is such a critical matter.

Self-sync bytes have 10 bits

To overcome this problem of deciding where to start reading data, the designers of Apple's disk drive decided to define a special byte called a self-sync byte. The one thing that differentiates this byte from all other bytes is that it is composed of 10 bits, with the extra two bits always being zeroes. In normal DOS, the self-sync byte is an FF with two zero bits appended to it. In the modified DOS that's used in many protection schemes, this FF byte has been changed to other values. But, while the value may change, one thing doesn't. The two extra bits are always zeroes.

At this point, if you are the least bit curious, you're probably wondering how these special bytes can bring the hardware into sync with the data coming of the diskette. It's not difficult to understand, but first we must know that the Apple hardware will not start reading a byte from the diskette unless the first bit is a 1. So, if it encounters a zero bit first, it will skip over it and wait for the next "1"-bit to appear. Knowing this, it is now possible to figure out how many self-sync bytes will be necessary to guarantee that the hardware is synchronized with the data. To do this, we should first write out the bits for about half a dozen self-sync bytes. They would look like the first line in the diagram below. As you can see, I have included spaces between each sync byte, but when they are read off the diskette, they would simply appear as a continuous stream of bits. Now, if we start to read this stream of data (e.g. the first time we start with the first bit, the second time we start with the second bit, etc.) we can find out how many sync bytes we'll have to read before we can be sure that the data we are reading is accurate.

If we look at the chart above, we find that it takes a maximum of four self-sync bytes to insure synchronization. No matter where we start reading data, by the time we've reached the fifth byte, we know we must be reading an $FF. For this reason, the gaps on a diskette track must have a minimum of five self-sync bytes.

How Self-Sync Bytes Synchronize Data and Hardware

Start Bit	Data Read From the Diskette						Syncs on Byte
	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	
1	1111111100	1111111100	1111111100	1111111100	1111111100	1111111100	1
2	111111100	1111111100	1111111100	1111111100	1111111100	1111111100	2
3	11111100	1111111100	1111111100	1111111100	1111111100	1111111100	2
4	11111001	111111100	1111111100	1111111100	1111111100	1111111100	3
5	11110011	11111100	1111111100	1111111100	1111111100	1111111100	3
6	11100111	11111001	111111100	1111111100	1111111100	1111111100	4
7	11001111	11110011	11111100	111111100	1111111100	1111111100	5
8	10011111	11100111	11111001	11111100	1111111100	1111111100	5
9	0011111111	0011111111	0011111111	0011111111	0011111111	0011111111	1
10	0111111110	0111111110	0111111110	0111111110	0111111110	0111111110	1

Synchronizing Data and Hardware With 8-Bit Bytes

Start Bit	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Syncs on Byte
	E4	92	E4	92	E4	92	
1	11100100	10010010	11100100	10010010	11100100	10010010	1
2	11001001	0010010111	0010010010	010111001	00	10010010	6
3	10010010	010010111	0010010010	010111001	00	10010010	6
4	0010010010	010111001	00	10010010	11100100	10010010	4
5	01001001	0010111001	00	10010010	11100100	10010010	4
6	10010010	010111001	00	10010010	11100100	10010010	4
7	00	10010010	11100100	10010010	11100100	10010010	2
8	0	10010010	11100100	10010010	11100100	10010010	2
9		10010010	11100100	10010010	11100100	10010010	2
10		0010010111	0010010010	010111001	00	10010010	6
11		010010111	0010010010	010111001	00	10010010	6
12		10010111	0010010010	010111001	00	10010010	6
13		0010111001	00	10010010	11100100	10010010	4
14		0101110010	0	10010010	11100100	10010010	4
15		10111001	00	10010010	11100100	10010010	4
16		0	11100100	10010010	11100100	10010010	3

Self-sync bytes can be eliminated

While it is not generally known, it's not absolutely necessary to have the 10-bit self-sync bytes on a diskette. If you choose the right combination of bytes, and there are several, you can synchronize the disk hardware and data with ordinary 8-bit bytes. You do pay a price for this however. Generally speaking you will need at least one more 8-bit sync byte. So, instead of requiring a minimum of four 10-bit sync bytes to guarantee synchronization, you'd need a minimum of five 8-bit sync bytes. Interestingly enough, both situations result in the same number of bits (40) to insure synchronization. One bit pattern that will insure synchronization within 5 sync bytes is **E4 92.** Let's check this out as we did with the 10-bit sync bytes. As you can see from the table above, no matter which of the first 16 bits you start reading the diskette from, by the time you have reached the 6th byte, the hardware and the data are in sync.

Self-sync bytes can't be copied

Why have we spent so much time trying to understand self-sync bytes? The answer is simple, they can be used to great advantage in copy protection schemes. Many people have asked me, "Why can't you write a program that simply reads a series of bits off of one diskette and writes it out to another diskette?" That's a good question, and at first glance you would think that it would be a simple matter to do just that. There is a problem however, and that is, *self-sync bytes cannot be copied.* The hardware in the Apple computer is setup in such a way, that only 8-bit bytes can be read from the diskette and stored in memory. If a 10-bit sync byte is read, the two trailing zeroes are lost. You may be wondering why it's so important to identify self-sync bytes. It's important because with their help, it becomes possible to identify the beginning and the end of a track. This is necessary on soft-sectored systems like the one used for the Apple, because there is no physical marker on the diskette that can be used to tell the computer where a track begins. Hard sector diskettes have a hole punched at sector zero, so the disk drive hardware can be used to locate the beginning of a track.

From our discussion of the track format in the last issue (**ASPD** Vol. 1, No. 2, p. 20) we learned that there are three types of gaps on a track: one that marks the beginning of a track, one that separates the address field from the data field and one that separates the data filed from the next address field. As it turns out, the gap that marks the beginning of a track is easily identified because it turns out to be the longest gap on the track. Thus, when nibble copy programs go to copy a track, they read the track into RAM and analyze it, searching for a large group of $FFs. Once they find this group of $FFs, they assume that they have located the beginning of the track and mark its location. Next, the bit copier programs look for a second occurrence of this large group of $FFs. Once they find it, they know that they've located the spot where the track starts to repeat itself. They then backup to the spot just before these $FFs began and mark that as the end of the track. Now that the bit copier knows where the start and finish of the track is, all it has to do is write this information back out to the diskette. As it does that it makes assumptions as to which bytes are sync bytes. If the track overlaps itself when it is written back out to the diskette, the bit copier will then start removing either zeroes from sync bytes, or sync bytes themselves to shorten the length. If bytes are removed, a problem will result in those programs that contain a nibble counting scheme.

After software developers analyzed the bit copiers and realized that they were looking for large groups of $FFs, they decided to use different bytes for self-sync bytes, temporarily making the nibble copy programs ineffective. Some developers even went so far as to use many different self-sync bytes on the same diskette. The next generation of nibble copy programs learned to overcome this new protection ploy by incorporating sophisticated disk reading routines that could detect the presence of the two trailing zeroes that were tacked onto self-sync bytes. With these reading techniques, it was no longer important what the sync byte was, the nibble copiers could detect them because they could find the two zero bits that are characteristic of self-sync bytes.

Use your own sync bytes

If you want to create your own protection scheme by changing the byte that's used as a self-sync byte, it's very easy to do. All you have to do is change one location in memory — location **48224 ($BC60).** This location is near the beginning of the routine that's called by DOS's formatter and writes the address headers. It's entered with a number in the Y-register that tells the routine how many self-sync bytes to write. An LDA instruction at 48223 ($BC5F) gets the byte that will be stored as the self-sync byte on the diskette. You can change this value from an $FF to any other legal value (see chart of legal diskette bytes in **ASPD** Vol. 1, No. 2, p. 22) from BASIC. For example, if we wanted to change the normal self-sync byte to $FE, we would type in the following line:

```
POKE 48224,254
```

That's all there is to it. While it's easy to change the sync byte, your software must check for its presence to make it effective as a software protection scheme. In addition, it will not prevent current copy programs from backing up your diskette. In a future issue of **ASPD** we will talk about producing a useful protection scheme using modified sync bytes, bit insertion and nibble counting.

Cracks Wanted

Listed below are programs that our readers would like to unprotected. Anyone who comes up with a method of removing the protection from any of these programs will get a free three-month subscription, or extension to ASPD, so get those solutions in.

If you have a program that you'd like to see unprotected, please let us know, and we'll add it to our list so that some of our readers can try their hands at it.

1. Sensible Speller - DOS
2. Sensible Speller - ProDOS
3. Crush, Crumble & Chomp
4. Wizardry
5. Compress Software
6. Dazzle Draw
7. Newsroom
8. Word Handler
9. Disk-O-Check
10. The Game Show
11. Batter Up!
12. Certificate Maker

HOW TO RESTORE LOST APPLESOFT PROGRAMS

Many programs that are written in Applesoft appear to wipe themselves out when they're finished or when the program detects an attempt by the user to break out of the program by pressing Control-C or Control-RESET. Other programs contain special REM statements, which I call Wipeout REMs, that will automatically erase the program when you try to list it. The RAM disk formatter that was used with the Synetix 294K RAM card contained many statements of this type. These REM statements are easy to produce (I'll show you how to produce them in the next issue of **ASPD**) and provide protection from novice or casual users who want to examine, copy or otherwise manipulate your program code. They can be overcome however in many ways, and a program that can restore programs that are erased by the NEW or FP commands can be a powerful tool.

Another use for such a restoring capability comes up when you're trying to transfer Applesoft files from a protected disk to an unprotected one. This can be done by loading the program in from the protected disk. Once you get the Applesoft prompt back, you get into the monitor by typing **CALL -151**. Then type **AF.B0**. This command will print out two hexadecimal numbers for you which represent the low byte and the high byte of the end of the current BASIC program. Usually we know where an Applesoft program begins, at $801. However, if you don't want to take any chances, you can examine the start of program pointers which are located on page zero at locations $67 and $68.

Now that you know where the program starts and ends, you can move it up, out of the way of DOS's bootup process. I usually move the BASIC program up to $6000 and then boot a normal DOS 3.3 slave diskette. A slave diskette is one that is created by just using the INIT command and not using MASTER CREATE. Once DOS 3.3 has been loaded into the computer, **BLOAD &RESTORE**. Then move the BASIC program down from the $6000 memory range and restore the BASIC program by typing **CALL 768**. Once *&RESTORE* has been run, it can subsequently be invoked by typing **CALL 768** again or by typing **&RESTORE**.

To clarify the situation, let's take a fictional BASIC program that is on a

```
                    1000 ***********************************
                    1010 ***                             ***
                    1020 ***          &RESTORE           ***
                    1030 ***                             ***
                    1040 ***   COPYRIGHT (C) 1982 BY     ***
                    1050 ***      JULES H. GILDER        ***
                    1060 ***    ALL RIGHTS RESERVED      ***
                    1070 ***                             ***
                    1080 ***********************************
                    1090 *
                    1100 *
                    1110 *
                    1120        .OR $300
                    1130 *
                    1140 *
                    1150 * CONSTANTS
                    1160 *
004C-               1170 JUMP    .EQ $4C         JMP op code
00AE-               1180 RESTORE .EQ $AE         RESTORE token
                    1190 *
                    1200 *
                    1210 * EQUATES
                    1220 *
0006-               1230 POINTER .EQ $6
0008-               1240 TESTBYT .EQ $8
0067-               1250 TXTTAB  .EQ $67
0069-               1260 VARTAB  .EQ $69
006B-               1270 ARYTAB  .EQ $6B
006D-               1280 STREND  .EQ $6D
00AF-               1290 PRGEND  .EQ $AF
03F5-               1300 AMPERSD .EQ $3F5
DEC0-               1310 SYNCHR  .EQ $DEC0
FC58-               1320 HOME    .EQ $FC58
FDED-               1330 COUT    .EQ $FDED
                    1340 *
                    1350 *
                    1360 * This is where the ampersand jump
                    1370 * vector is set up.  After set-up,
                    1380 * a relative jump is made to the
                    1390 * second entry point of the program.
                    1400 *
0300- A9 4C         1410         LDA #JUMP       Get the JMP
0302- 8D F5 03      1420         STA AMPERSD     op-code & store
0305- A9 12         1430         LDA #START      it and the
0307- 8D F6 03      1440         STA AMPERSD+1   address of the
030A- A9 03         1450         LDA /START      start of this
030C- 8D F7 03      1460         STA AMPERSD+2   program.
030F- 4C 17 03      1470         JMP START2      Go to START2.
                    1480 *
                    1490 *
                    1500 * There are two entry points to this
                    1510 * program.  One is via the &RESTORE
                    1520 * command (START) and one by a CALL 768
                    1530 * (START2).  At START, the program
                    1540 * looks at the information that follows
                    1550 * the & to see if it is the RESTORE
                    1560 * token.  This is done by SYNCHR. If
                    1570 * not RESTORE a  syntax error is
                    1580 * generated. Once  syntax has been
                    1590 * checked, the program title is
                    1600 * printed out.
                    1610 *
0312- A9 AE         1620 START   LDA #RESTORE    Does the RESTORE
0314- 20 C0 DE      1630         JSR SYNCHR      token follow the &?
0317- A0 00         1640 START2  LDY #$0         Yes, zero character pointer.
0319- 20 58 FC      1650         JSR HOME        Clear the screen.
031C- B9 7E 03      1660 LOOP1   LDA TEXT,Y      Get a character.
031F- F0 06         1670         BEQ NEXT        If done go to NEXT.
0321- 20 ED FD      1680         JSR COUT        Print a character.
0324- C8            1690         INY             Increment the pointer.
0325- D0 F5         1700         BNE LOOP1       Get more characters.
                    1710 *
                    1720 *
                    1730 * This section of program resets the
                    1740 * start of program pointers that are
                    1750 * wiped out when a NEW or Control-B
                    1760 * are entered.
                    1770 *
0327- A5 67         1780 NEXT    LDA TXTTAB      Get program start
0329- 18            1790         CLC             low byte. Calculate
032A- 69 03         1800         ADC #$3         and save the starting
032C- 85 06         1810         STA POINTER     line's low byte.
032E- A5 68         1820         LDA TXTTAB+1    Get program start
```

protected diskette. After booting the protected diskette, we reset into the monitor and find the beginning and end of the program by typing:

```
67.68 N AF.B0
```

For our fictitious program, the computer responds by printing:

```
0067 - 01
0068 - 08
00AF- 97
00B0- 21
*
```

Next, we move the program up in memory, out of harms way by typing:

```
6001<801.2197M
```

With the the program safe from destruction, we boot a DOS 3.3 diskette and **BLOAD &RESTORE**. Then we move our BASIC program back down where it belongs getting into the monitor mode with **CALL -151** and then typing:

```
801<6001.7997M
```

Finally, we restore the program by typing **CALL 768.**

What makes it possible for a program like *&RESTORE* to resurrect a dead Applesoft program is the fact that the designers of the Applesoft language wanted to have an efficient language and decided that it was not necessary to actually erase the contents of memory every time a NEW or FP command was issued. Instead, they just changed the information stored in the end of program pointer and erased only two bytes of data from the program. Thus, the program is still in memory, it's just that Applesoft doesn't know where to look for it. By restoring the two bytes that were erased (the pointer to the second line of the Applesoft program), and searching through memory until the end of the program is found and restoring the PRGEND pointer, the program can be brought back to life, as if it were always there.

While *&RESTORE* will bring back programs that were NEWed or FPed, it will not help a bit if the program has been wiped out by zeroing out all of memory with a program such as *Wipeout 1* or *Wipeout 2* (see **ASPD** Vol. 1, No. 2, p.17 for more details). Bear this in mind when setting up your own

```
0330- 85 07     1830          STA POINTER+1   high byte and save it.
0332- A0 01     1840          LDY #$1         Save 2nd line's
0334- 91 67     1850          STA (TXTTAB),Y  high byte.
0336- 88        1860          DEY             Zero the Y-register.
0337- C8        1870 FINDEOL  INY             Look for the end
0338- B1 06     1880          LDA (POINTER),Y of line marker
033A- D0 FB     1890          BNE FINDEOL     Keep looking.
033C- 98        1900          TYA             Found end of line,
033D- 18        1910          CLC             find value of
033E- 69 05     1920          ADC #$5         program start
0340- A0 00     1930          LDY #$0         low byte and
0342- 91 67     1940          STA (TXTTAB),Y  restore it.
                1950 *
                1960 *
                1970 * This part of the program resets the
                1980 * end of program pointers.
                1990 *
0344- A5 67     2000          LDA TXTTAB      Store start of
0346- 85 06     2010          STA POINTER     program pointers
0348- A5 68     2020          LDA TXTTAB+1    in POINTER for
034A- 85 07     2030          STA POINTER+1   future use.
034C- A9 00     2040 LOOP2    LDA #$0         Initialize end of
034E- 85 08     2050          STA TESTBYT     program test byte.
0350- B1 06     2060 LOOP3    LDA (POINTER),Y Start scanning
0352- C8        2070          INY             the program.
0353- D0 02     2080          BNE ZEROCHK     Page boundary?
0355- E6 07     2090          INC POINTER+1   Yes, increment the byte.
0357- C9 00     2100 ZEROCHK  CMP #$0         Does the accumulator=0?
0359- D0 F1     2110          BNE LOOP2       No, keep scanning.
035B- A5 08     2120          LDA TESTBYT     Yes, is it the
035D- C9 02     2130          CMP #2          end of the program?
035F- F0 04     2140          BEQ EXIT        Yes, finish up.
0361- E6 08     2150          INC TESTBYT     No, increment the test byte.
0363- D0 EB     2160          BNE LOOP3       Get the next byte.
0365- C8        2170 EXIT     INY             Adjust the byte
0366- 98        2180          TYA             count & see if
0367- D0 02     2190          BNE STORPTR     we have to increment
0369- E6 07     2200          INC POINTER+1   the high byte too.
036B- 85 69     2210 STORPTR  STA VARTAB      Store the low byte
036D- 85 6B     2220          STA ARYTAB      of the end of the program
036F- 85 6D     2230          STA STREND      in the appropriate
0371- 85 AF     2240          STA PRGEND      zero page locations.
0373- A5 07     2250          LDA POINTER+1   Store the high byte
0375- 85 6A     2260          STA VARTAB+1    of the end of the program
0377- 85 6C     2270          STA ARYTAB+1    in the appropriate
0379- 85 6E     2280          STA STREND+1    zero page
037B- 85 B0     2290          STA PRGEND+1    locations.
037D- 60        2300          RTS             Return.
                2310 *
                2320 *
                2330 * This is where text for program title
                2340 * and copyright notice are stored.
                2350 *
037E- A6 D2 C5
0381- D3 D4 CF
0384- D2 C5     2360 TEXT     .AS -"&RESTORE"
0386- 8D 8D     2370          .HS 8D8D
0388- C2 D9 A0
038B- CA D5 CC
038E- C5 D3 A0
0391- C8 AE A0
0394- C7 C9 CC
0397- C4 C5 D2  2380          .AS -"BY JULES H. GILDER"
039A- 8D        2390          .HS 8D
039B- C3 CF D0
039E- D9 D2 C9
03A1- C7 C8 D4
03A4- A0 A8 C3
03A7- A9 A0 B1
03AA- B9 B8 B2  2400          .AS -"COPYRIGHT (C) 1982"
03AD- 8D        2410          .HS 8D
03AE- C1 CC CC
03B1- A0 D2 C9
03B4- C7 C8 D4
03B7- D3 A0 D2
03BA- C5 D3 C5
03BD- D2 D6 C5
03C0- C4        2420          .AS -"ALL RIGHTS RESERVED"
03C1- 8D 8D 8D  2430          .HS 8D8D8D
03C4- D2 C5 C1
03C7- C4 D9 AE  2440          .AS -"READY."
03CA- 8D 00     2450          .HS 8D00
```

protection schemes. Don't be lazy. Take the extra effort to include a wipeout feature in your protection schemes, if you really want to keep your programs from prying eyes.

For those of you who are interested in just how the *&RESTORE* program works, a detailed description of this assembly language program follows. For those of you who just want to use it, typing the accompanying BASIC program and it will install &RESTORE into memory for you and give you the option of saving the machine language program out to disk.

Redefining commands

In this program I'll also show you how you can use the existing set of key words and give them new functions to perform. In this case, as you've already guessed, we're going to use the **RESTORE** command. This command will still perform its usual function without any problems. But, when it is preceded by another command, the ampersand (&), it takes on an entirely new task.

The *&RESTORE* program begins, on line 1410, by setting up the ampersand jump vector to point to START and after that jumps to START2 (line 1470), skipping the code that checks for the presence of the word **RESTORE**. At line 1620, which is the ampersand entry point, the program loads the token for the word **RESTORE** (which is $AE) into the accumulator and then jumps to the syntax character checking routine (SYNCHR) to see if that token matches the information following the ampersand. If it doesn't, the subroutine prints out **SYNTAX ERROR** and stops execution of the main program. If it matches, the main program falls into the START2 routine.

It is not at all necessary to use the RESTORE command, but I thought you'd like to see how to do it. If you prefer to use just the & as the command, simply eliminate lines 1470, 1620 and 1630 and rename the label on line 1640 START. Once at line 1640, the program clears the screen and prints out the program's title, copyright notice and the word **READY**, indicating to the user that the program has already been restored. While the program has not yet been restored, the task is accomplished so quickly, that the user never realizes it.

The actual program restoration begins on line 1780 where the start of program pointer, TXTTAB, is used to produce another pointer (lines 1780 to 1830), called POINTER, which will skip the first four bytes of the line (these consist of the next line pointer and the line number). The reason we want to skip these bytes is that ultimately we want to find the end of the first line which is terminated with a zero. However, any of the first four bytes can legitimately contain a zero, which could result in premature termination of this program.

After POINTER has been calculated and stored, the high-byte of the start of program pointer is still in the accumulator and it is stored as the high byte of the pointer to the second line in the program (line 1850).

Now that the high-byte of the next line link to the second line has been restored, we have to find out where the first line ends in memory so that we can restore the low-byte. The routine that does this, FINDEOL, begins in line 1870. In lines 1870 and 1880, the Y-register is incremented and the contents of the location pointed to by both POINTER and the offset of the Y-register, are checked to see if they are zero. If not, the process is repeated until they are. If they are, the Y-register is transferred to the accumulator (line 1900), the carry bit is cleared in preparation for adding two numbers (line 1910) and 5 is added to the accumulator (line 1920). The five includes the four bytes that were skipped at the beginning, plus an additional byte so that the pointer will point not to the last character of the Applesoft line, but one past it, where the next line actually begins. This number is stored in the low byte of the next line pointer (line 1940).

If the program were to stop at this point, you would be able to list the program and it would appear as if it had all been restored. It hasn't, because if you saved it out to tape or disk and then loaded it back in, you'd find you had nothing, even though you were able to list it, and also run it. The program can be saved at this point only by listing it to an EXEC file. The reason the program will not save out properly is that we have not adjusted the end of program pointer, PRGEND, to point to the end of the program. This is what is done, starting at line 2000, where TXTTAB, the start of program pointer, is loaded into POINTER (lines 2000 to 2030).

In lines 2040 and 2050, a flag called TESTBYT is set to zero. This is going to be used to help us determine when the end of the program has been reached. A loop to scan the program is set up starting at line 2060, where POINTER and the Y-register are used to determine the next location from which a byte will be loaded and tested to see if it is equal to zero. After the byte is loaded, and before the test is performed, the Y-register is incremented (line 2070) and a check is made to see if a memory page boundary has been crossed (e.g. did we go from an address in the $800 range to an addresses in the $900 range). If no page boundary was crossed (line 2080) the program branches to ZEROCHK, otherwise, the high-byte of POINTER is incremented by one.

ZEROCHK is where the byte in the accumulator is tested to see if it is a zero (line 2100). If it's not, the program branches back to line 2040 where TESTBYT is reset to zero, and then checks the next byte. If it is a zero, we have to determine if this is the end of an Applesoft line or the end of the program. To do this we check TESTBYT and see if it is equal to two (lines 2120 and 2130). If it is (line 2140), this is the end of the program and the program branches to a routine that stores all of the pointers. If it's not equal to two, we increment TESTBYT by 1 and go back to check the next byte. As you see, TESTBYT is used to determine how many consecutive zeros we have encountered. The end of the program is indicated by three consecutive zeros; one for the end of line marker and two instead of the next line pointer.

The EXIT routine is where all of the Applesoft pointers are adjusted. These include the end of program pointer (PRGEND), the start of variable storage (VARTAB), the start of array storage (ARYTAB) and the end of string storage (STREND). In line 2170, the Y-register is incremented by one because we want to point to one past the three consecutive zero bytes. The Y-register is then transferred to the accumulator (line 2180) and the high-byte of POINTER is incremented if a page boundary is crossed (lines 2190 and 2200). All of the appropriate zero page pointers are updated in lines 2210 to 2290.

continued on page 6

In the Next Issue
THE ULTIMATE LINE HIDER

ASPD PROGRAM DISKETTES AVAILABLE FOR ONLY $15 EACH

Every month we will make a DOS 3.3 diskette available that contains all of the programs for the current issue of *Apple Software Protection Digest.*

Each diskette is available for only $15 each. Diskette 1 contains the programs from issues 1 and 2. There are currently 3 diskettes available.

To order send a check, money order or your credit card number and expiration date to:

REDLIG SYSTEMS, INC.
2068 - 79th Street
Brooklyn, New York 11214

REDLIG SYSTEMS, INC.
2068 79th Street
Brooklyn, New York 11214

BULK RATE
U.S. POSTAGE
PAID
BROOKLYN, NY
Permit No. 631

[redacted]