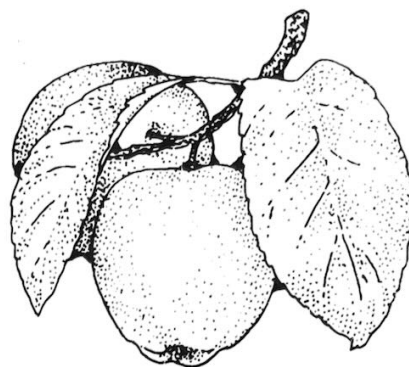


APPLE SOFTWARE PROTECTION DIGEST

\$3.00



Vol. 1, No. 2

January/February 1986



A LITTLE LATE, BUT BETTER THAN EVER

Well, we finally made it. For a while, I suspect that quite a few of you were wondering what happened to us. A couple of you even wrote that you feared your hard-earned bucks went into a black hole, never to be heard from again. Not true! As all of you early subscribers are aware, we sent out notices to each of you explaining that there would be a delay and offered you a complete refund with no questions asked. I'm happy to announce that not one of you requested a refund. I appreciate your confidence.

I think you're all entitled to an explanation of what happened and what you can expect in the future. The first issue of the Digest went together so easily and so quickly, that we really didn't anticipate any problems in getting the second one out. Well, nothing could be further from the truth. Just about everything that could go wrong did. Authors who promised material on cracking some of the more desirable programs, such as Sensible Speller, Newsroom and Wizardry, never came through. Add to that a few hardware problems and some problems with the printer and you wind up with a long delay.

We've finally got everything straightened out however, and we have a big jump on the next issue, so we should be back on schedule. We've also added back-up equipment, so hardware problems shouldn't cause any major delays anymore.

February 1987 issue (that's the last one they'll receive) while the last issue for everyone else who has subscribed up to now, will be January 1987. Everyone who has subscribed to date will have gotten the first issue free. Subscriptions start with the second issue.

Although we've been quite late with this issue, we hope you'll agree that the wait was well worth it.

Program diskettes available

With this issue of the Digest, we are adding a new service. Many of you have asked us to consider coming out with a moderately priced diskette with all the programs on it, so we have. Every month we will make available a diskette with all of the programs (with both source and object code when appropriate) on it. Assembly source files will be saved in text file format to facilitate use with a variety of assemblers. The diskette will be an unprotected DOS 3.3 diskette and will cost only \$15.

Cracks wanted list

We've added a new feature to the Digest with this issue, its our Cracks Wanted list. This will list all the programs that readers have expressed an interest in have cracks or back-up procedures for. In addition, starting with the next issue, each issue of the **Apple Software Protection Digest** will contain a list of all the program cracks we've presented during the year so that you'll have a quick and handy reference available at your finger tips.

Another feature we've included with this issue is a Letters column. This is a forum where you, the reader, can ask for information, present short ideas or discuss appropriate topics. It's also the way we get feedback from you one how we're doing.

continued on page 15

Contents

Editorial	1
Using COPYA to Backup Protected Disks	2
How to BRUN Applesoft Programs	4
Parameter Files for COPYP	6
Change the BLOAD Address of Binary Files	9
Letters	10
Hiding Machine Language Programs in Strings	11
Two Solutions to the Hide-A-Line Problem	12
Hiding Program Lines from BASIC	14
A Handy Decimal/Hexel-Decimal Converter	15
Applesoft Line Finder and Vanisher	16
Wipeout: The Ultimate Weapon of Destruction	17
Review: Locksmith 5.0G	19
Protection Tutorial — Part II ..	20

Two bonuses to say thanks

To say thanks to you for bearing with us, we've added two bonuses. Anyone who subscribed to the Digest in 1985 will automatically get an EXTRA FREE ISSUE to compensate them for the long wait. And, all subscribers are getting this double January/February issue, but it will only count as a single issue. Thus, all 1985 subscribers will have their subscriptions expire with the

USING COPYA TO BACKUP PROTECTED DISKS

If you plan on doing a lot of copying of protected diskettes, it would probably pay for you to purchase one of the nibble copy programs. My personal preference is *Copy II Plus*, which we reviewed in the last issue of *ASPD*. By the way, since that review came out, Central Point Software, the producers of *Copy II Plus*, has come out with a new version of the program — Version 6. The program sells for \$39.95, but you can purchase it from us for only \$30. Anyway, if you'd rather not shell out the \$30 dollars, there's an awful lot of protected programs that you can copy with COPYA, as long as you make the appropriate modifications to it, and that's what this article is going to tell you how to do.

We used a slightly modified version of COPYA to duplicate *Print Shop* in our last issue. Comments by *ASPD* readers indicating they like that approach, convinced me to go ahead and develop a general modification to COPYA that could accommodate a wide variety of programs.

Why doesn't COPYA work?

Before we can modify COPYA to copy protected disks (not all of them, but a good number of them) we have to understand why it won't copy them to begin with. The first thing that causes problems for us is the DOS error checking routine. The second thing is COPYA's error checking routine. If either of these routines encounters a diskette with a slightly modified DOS on it, they'll generate an I/O ERROR message and stop. Both of these routines can be easily disabled by executing two POKES. These are shown in line 75 of the program listing. By the way, the listing consists only of those lines that are to be added to COPYA. I have found that the simplest way to add them is to use an editor or word processor to create a text file that contains these lines. Then you should load COPYA into the Apple's memory and finally, EXEC the text file, which I call COPYP MAKER. Lines 300 and 305 are blank lines and when they're EXECed in, they cause those two lines to be deleted from Apple's original COPYA listing. From now on, I'll be referring to this modified version of COPYA as COPYP, and in fact, this issue already contains some parameters that will allow you to copy some protected programs.

Anyway, let's get back to our program description. The two POKES that are added in line 75 prevent the I/O ERROR from being generated and will thus permit COPYA to go ahead and duplicate the protected diskette, even though there's something "funny" about it. Sometimes those two POKES will be all that you'll need to copy a protected diskette. At other times, you may have to change the address and data field bytes

to make it possible to copy the diskette. I have provided additional modifications to COPYA to permit you to do this. Lines 85 and 86 read in the address and data field information from data statements. There is a separate data statement for the source (original) diskette and target (copy) diskette. The data listed below is for normal DOS 3.3 diskettes, so you can use this program to copy unprotected software as well.

As you can now see, the tutorial we have in this issue on the data and address fields has not been wasted. At least now you have an idea of why these bytes are being changed. In case you're adventurous and are considering modifying COPYP further to enhance its unprotecting capabilities (and I encourage all of you to do so), a little further explanation of just what is going on here is in order. Line 198 makes the source diskette's address and data field bytes the active ones and inserts them into DOS so that the protected diskette can be read. Line 248 makes the target address and data field bytes the active ones and inserts them into DOS so that the copy diskette can be initialized. Line 258 also uses the bytes for the target diskette, but this time they are inserted into DOS so that the target diskette can be written to.

If you will examine the specific bytes that are changed in lines 395 and 396, you will notice that only the read locations of the address and data field bytes are being changed

and might wonder why no write locations are being changed. The answer to that question is simple, the diskette that is being copied, is being transferred from a diskette with a protected DOS on it to a diskette with a normal DOS on it. Since we're only reading the protected diskette, there's no need to change its write locations. And, since the write locations haven't been changed from those normally used, there's no need to adjust them when we want to write data out to the unprotected diskette. Why then do we change the read locations back to normal? The answer to that is easy too. We change them back because after writing to the unprotected diskette, DOS reads what it's written to make sure that it was properly recorded. Not too difficult is it?

COPYP edits sectors too

So far the operation of COPYP has been explained up to line 397. But, just because we can copy a protected program onto an unprotected diskette, doesn't mean that the program will run. It may, but chances are that some routine somewhere in the program checks to make sure that the program is on the protected diskette and the protection scheme is intact. One way of doing this is to incorporate a nibble counting routine (we'll look into nibble counting in more detail in a future issue). To overcome these protection routines, it will be necessary to modify

```

0 REM COPYP MAKER
1 REM
2 TEXT: HOME
65 DIM B(10), S(10), T(10)
75 POKE 47426,24: POKE 929,24
80 PRINT " PROTECTED DISK DUPLICATION PROGRAM": PRINT: PRINT
85 FOR X = 1 TO 10: READ S(X): NEXT X
86 FOR X = 1 TO 10: READ T(X): NEXT X
100 POKE 715, PEEK (110) + 2: REM BUFSTART
198 FOR X = 1 TO 10: B(X) = S(X): NEXT X: GOSUB 395
225 VTAB 5: HTAB 24: PRINT " ": IF PEEK (713) = 1 THEN 295
248 FOR X = 1 TO 10: B(X) = T(X): NEXT X: GOSUB 395
258 FOR X = 1 TO 10: B(X) = T(X): NEXT X: GOSUB 395
290 GOTO 550
295 VTAB 19: GOTO 400
300
305
395 POKE 47445,B(1): POKE 47455,B(2): POKE 47466,B(3): POKE 47505,B(4):
POKE 47515,B(5)
396 POKE 47335,B(6): POKE 47345,B(7): POKE 47356,B(8): POKE 47413,B(9):
POKE 47423,B(10)
397 RETURN
400 IOB = 47080: TRK = IOB + 4: SCT = IOB + 5: CMD = IOB + 12: RWTS
768: RD=1: WR = 2
410 BUF = PEEK (IOB + 8) + 256 * PEEK (IOB + 9)
420 FOR X = 768 TO 774
430 READ Y: POKE X,Y
440 NEXT X
450 PRINT: PRINT "DOING SECTOR EDITS NOW": POKE IOB + 3,0
460 READ TK,SE,BYTE,OV,NV
470 IF (TK + SE + BYTE + OV + NV) * 1 = 0 THEN 550
480 POKE TRK,TK: POKE SCT,SE: POKE CMD,RD: CALL RWTS
490 IF PEEK (BUF + BYTE) <> OV THEN 530
500 POKE BUF + BYTE,NV
510 POKE CMD,WR: CALL RWTS
520 GOTO 460
530 PRINT: PRINT: PRINT "THIS VERSION OF THE PROTECTED PROGRAM"
540 PRINT "CANNOT BE UNPROTECTED WITH THE"
```

the code that is stored on the newly copied, unprotected diskette. This is done by reading in the appropriate sector, modifying the bytes of interest, and then writing the sector back out to the diskette. It's possible to do this manually, but it can be time consuming and annoying. It also means you have to have a little bit more of a hacker mentality. With the sector editing modifications I've added to COPYP, however, all you have to do is type in a few data statements, and the program will do the sector edits for you automatically. Here's how it works.

Lines 400 and 410 set up the variable names that will be used in this routine. Descriptive names are used so that it will be a little easier to see what is happening. Lines 420 through 440 read in a short machine language program that uses DOS's RWTS (Read and Write a Track and Sector) routines. Line 450 informs the user that COPYP is now implementing any sector edits that may have been requested. Next, any DATA statements that have been added between lines 999 and 9000 are read and used to specify what specific sector edits should be made (line 460). Line 998 tells you what data and in what order, it should be entered. Because software companies frequently change protection schemes, the COPYP sector editor requires that you enter both the value of the original byte and the value of the new byte that is to replace it. The program then checks to see if the value of the byte to be edited matches the old value of the byte that you entered (line 490). If it does, it proceeds to

perform the edit (lines 500 and 510). If it doesn't, it assumes that the protection scheme has changed somewhat and tells the user that the program cannot be unprotected with the parameters that were supplied (lines 530 to 545), and terminates the program.

If no sector edits are requested, the program will read the data in line 9000. It is very important that this line always be present because the zeros it contains signal the program that no more sector edits need to be done and the program should be terminated (line 470).

Determining the parameters

While the COPYP program presented here can be a very useful tool for unprotected diskettes, that's all it is. As with any other tool, you have to know how to set it up and use it. Setting it up is easy if you have the right information. But getting that information is a little more difficult. How do you know what values are used in the address and data fields? How do you know if any sector edits are required? Mostly it's a matter of digging into the protected program and figuring things out. ASPD will provide you with parameter files for use with COPYP and hopefully, all of you will contribute your time and expertise as well. Anyone who submits a parameter file or a modification to COPYP that makes it even more useful will get a one-month extension to his or her subscription when we print it. Contribute on a

regular basis and you'll wind up getting the Digest for free.

If you're interested in sorting things out by yourself, the first place to start is to either boot the program and then break out of it (you'll need a computer with an old F8 ROM in it or an NMI switch (more about that next issue) to do that. Once you get into the Apple monitor—that's where you get an asterisk (*) as a prompt instead of a square bracket (]), then you can examine the memory locations that hold the data for the data and address fields (see the chart in this month's tutorial). This of course assumes that only a slightly modified DOS is used. If that's not the case, then you'll have to use a program that can read in raw data directly from the diskette and display it. From this you'll be able to tell what the various address and data field bytes are.

COPYP is limited

While you'll find COPYP helpful in unprotected many programs, it can't be used for all of them because it currently lacks the ability to do some of the fancier things such as quarter and half track, nibble insertion, synchronous copying, etc. Over the next few months I hope to be able to add some of these capabilities to the program. Anyone who has ideas on how to improve COPYP is invited to send them in to me, and I'll see to it that the rest of our readers find out about it too. You'll also get a 3 to 6 month extension to your subscription for major, useful, modifications. In the meantime, you have the beginnings of a powerful tool. Let's see what you can do with it.

```

545 PRINT "PARAMETERS YOU'VE SUPPLIED."
550 POKE CMD,0
560 TEXT: HOME: PRINT CHR$(4);"FP"
890 REM
899 REM SOURCE ADDRESS FIELD PROLOGUE BYTES
900 DATA 213,170,150
909 REM SOURCE ADDRESS FIELD EPILOGUE BYTES
910 DATA 222,170
919 REM SOURCE DATA FIELD PROLOGUE BYTES
920 DATA 213,170,173
929 REM SOURCE DATA FIELD EPILOGUE BYTES
930 DATA 222,170
935 REM
939 REM TARGET ADDRESS FIELD PROLOGUE BYTES
940 DATA 213,170,150
949 REM TARGET ADDRESS FIELD EPILOGUE BYTES
950 DATA 222,170
959 REM TARGET DATA FIELD PROLOGUE BYTES
960 DATA 213,170,173
969 REM TARGET DATA FIELD EPILOGUE BYTES
970 DATA 222,170
980 REM
981 REM RWTS ROUTINE
982 REM
983 DATA 160,232,169,183,76,217,3
990 REM
995 REM SECTOR EDIT DATA
996 REM
998 REM TRACK,SECTOR,BYTE,OLD VALUE,NEW VALUE
999 REM
9000 DATA 0,0,0,0,0
9993 REM
9994 REM
9995 REM COPYRIGHT (C) 1986 BY
9996 REM
9997 REM JULES H. GILDER
9998 REM
9999 REM ALL RIGHTS RESERVED

```

**Don't miss a
single issue
SUBSCRIBE
TODAY!**

Apple Software Protection Digest
 Publisher & Editor, Jules H. Gilder; Contributing Editor, J. Scott Barrus. Copyright © 1986 by Redlig Systems, Inc., 2068-79th Street, Brooklyn, NY 11214. All rights reserved. No part of this publication may be reproduced, or electronically transmitted or stored without the publisher's written permission. 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.

Apple is a resistered trademark of Apple Computer, Inc.

HOW TO BRUN APPLESOFT PROGRAMS

In the last issue we spoke about several different ways that lines in an Applesoft program can be hidden from view. One of the techniques used simply required that the next line pointer of the line before the one to be hidden, be changed so that it points to the line that follows the line or group of lines that are to be hidden. This is an effective technique, but it has some flaws, chief among them being a routine that automatically resets the line pointers to their correct value every time an Applesoft program is loaded.

The other problem with this approach is that if a line is added or deleted from the program, the line links again are automatically re-calculated and corrected. One way to overcome Apple DOS's apparent desire to prevent you from hiding certain program lines is to let DOS BRUN Applesoft programs instead of RUN them. This overcomes DOS's zealous attempts to maintain line link integrity by fooling DOS into thinking that our Applesoft program is actually a binary one. In addition, while we're loading our Applesoft program into memory, we can disable the RESET key and set Applesoft's auto-run flag to prevent anyone from pressing RESET to get out of the program and trying to list it. Finally, if you include an ONERR statement, that checks for someone pressing Control-C, as the first statement in your program, you can also prevent the user from using Control-C to stop the program. The statements you'd have to add to your program would look something like these:

```
0 ONERR GOTO 10000
10000 IF PEEK (222) = 255 THEN RUN
10010 Y = PEEK (222)
10020 PRINT "ERROR = ";Y
10030 RESUME
```

Of course, you can change the line numbers to anything that is convenient for you, but the first line of the program should contain the ONERR statement. These statements will cause the program to re-run itself if a Control-C is attempted and will print out the error number if any other error is encountered. It's best to add these lines only after the rest of the program has been completely debugged.

To make life easy for you, I have written a machine language program that automatically converts any standard Applesoft program into one that can be BRUN. The BRUN MAKER program asks you for the name of the Applesoft program that is to be converted and loads it into memory. At this point, it places a command, **CALL 37876** on the last line of the screen and reduces the active area of the screen by one line so that the information displayed there won't accidentally be erased. You may have noticed that there are several apparent "garbage" characters at the lower right-hand corner of the screen. These "garbage" characters represent

```

1000 *****
1010 *
1020 * BRUN MAKER FOR APPLESOFT PROGRAMS *
1030 *
1040 * Copyright (c) 1986 by *
1050 *
1060 * Jules H. Gilder *
1070 *
1080 * All Rights Reserved *
1090 *
1100 *****
1110 *
1120 *
0006- 1130 TXTPTR .EQ $06
0023- 1140 WNDWBOT .EQ $23
0025- 1150 CV .EQ $25
00AF- 1160 PROGEND .EQ $AF
00D6- 1170 RUNFLAG .EQ $D6
0200- 1180 INBUFF .EQ $200
0240- 1190 INBUFF2 .EQ $240
03F2- 1200 RESET .EQ $3F2
03F4- 1210 POWERUP .EQ $3F4
07E9- 1220 BRUNSTART .EQ $7E9
D566- 1230 RUN .EQ $D566
E000- 1240 BASIC .EQ $E000
FC58- 1250 HOME .EQ $FC58
FC62- 1260 CRTN .EQ $FC62
FD6F- 1270 GETLN1 .EQ $FD6F
FDDA- 1280 PRHEX .EQ $FDDA
FDED- 1290 COUT .EQ $FDED
1300 *
1310 *
1320 .OR $93A8
1330 .TA $0800
1340 *
1350 * This routine prints the opening screen and asks user
1360 * for the name of the file that will be converted to a
1370 * binary file so that it can be BRUN.
1380 *
93A8- 20 58 FC 1390 START JSR HOME
93AB- A9 83 1400 LDA #TEXT1
93AD- A0 94 1410 LDY /TEXT1
93AF- 20 52 94 1420 JSR MSGPRT
93B2- 20 6F FD 1430 JSR GETLN1
1440 *
1450 * Here the name entered name is transferred to a buffer
1460 * where it won't be destroyed and the terminating
1470 * carriage return is converted to a zero.
1480 *
93B5- A9 00 1490 LDA #$0
93B7- 9D 40 02 1500 STA INBUFF2,X
93BA- CA 1510 LOOP1 DEX
93BB- BD 00 02 1520 LDA INBUFF,X
93BE- 9D 40 02 1530 STA INBUFF2,X
93C1- E8 1540 INX
93C2- CA 1550 DEX
93C3- D0 F5 1560 BNE LOOP1
1570 *
1580 * A command that the user must copy with the cursor
1590 * is printed out on the last line of the display.
1600 *
93C5- A5 25 1610 LDA CV
93C7- 8D 80 94 1620 STA CV2
93CA- E6 25 1630 INC CV
93CC- A9 1D 1640 LDA #TEXT2
93CE- A0 95 1650 LDY /TEXT2
93D0- 20 52 94 1660 JSR MSGPRT
93D3- A9 17 1670 LDA #$17
93D5- 85 23 1680 STA WNDWBOT
93D7- AD 80 94 1690 LDA CV2
93DA- 85 25 1700 STA CV
1710 *
1720 * User is told to type VTAB 24 to continue.
1730 *
93DC- A9 35 1740 LDA #TEXT3
93DE- A0 95 1750 LDY /TEXT3
93E0- 20 52 94 1760 JSR MSGPRT
1770 *
1780 * Now the file is loaded into memory and the user
1790 * can do intermediate work or VTAB 24 and copy the
1800 * command that's on the screen.
1810 *
```



```

93E3- A9 9D 1820 LDA #TEXT4
93E5- A0 95 1830 LDY /TEXT4
93E7- 20 52 94 1840 JSR MSGPRT
93EA- A9 40 1850 LDA #INBUFF2
93EC- A0 02 1860 LDY /INBUFF2
93EE- 20 52 94 1870 JSR MSGPRT
93F1- 20 62 FC 1880 JSR CRTN
1890 *
1900 * This is where the BRUN code is
1910 * copied to its required location. The command
1920 * that is copied with the cursor re-starts program
1930 * execution.
1940 *
93F4- A2 00 1950 LDX #$00
93F6- BD 68 94 1960 LOOP3 LDA BRUNCODE,X
93F9- 9D E9 07 1970 STA BRUNSTART,X
93FC- F0 03 1980 BEQ FINISHUP
93FE- E8 1990 INX
93FF- D0 F5 2000 BNE LOOP3
2010 *
2020 * With the binary code in place, the length of the
2030 * file that is to be created is calculated and saved
2040 * for use with the BSAVE command.
2050 *
9401- 38 2060 FINISHUP SEC
9402- A9 E9 2070 LDA #BRUNSTART
9404- E9 01 2080 SBC #$1
9406- 8D 81 94 2090 STA LENGTH
9409- 38 2100 SEC
940A- A5 AF 2110 LDA PROGEND
940C- ED 81 94 2120 SBC LENGTH
940F- 8D 81 94 2130 STA LENGTH
9412- A5 B0 2140 LDA PROGEND+1
9414- E9 07 2150 SBC /BRUNSTART
9416- 8D 82 94 2160 STA LENGTH+1
2170 *
2180 * The BSAVE command is issued. The name used is the
2190 * same as the original with a B. appended to
2200 * it. The starting address and length are added
2210 * to the end of the name.
2220 *
9419- A9 A5 2230 LDA #TEXT5
941B- A0 95 2240 LDY /TEXT5
941D- 20 52 94 2250 JSR MSGPRT
9420- A9 40 2260 LDA #INBUFF2
9422- A0 02 2270 LDY /INBUFF2
9424- 20 52 94 2280 JSR MSGPRT
9427- A9 B0 2290 LDA #TEXT6
9429- A0 95 2300 LDY /TEXT6
942B- 20 52 94 2310 JSR MSGPRT
942E- AD 82 94 2320 LDA LENGTH+1
9431- 20 DA FD 2330 JSR PRHEX
9434- AD 81 94 2340 LDA LENGTH
9437- 20 DA FD 2350 JSR PRHEX
943A- 20 62 FC 2360 JSR CRTN
2370 *
2380 * Here the user is told the name of the binary file
2390 * and control is returned to BASIC via the warm
2395 * start vector.
2400 *
943D- A9 18 2410 LDA #$18
943F- 85 23 2420 STA WNDWBOT
9441- A9 BA 2430 LDA #TEXT7
9443- A0 95 2440 LDY /TEXT7
9445- 20 52 94 2450 JSR MSGPRT
9448- A9 40 2460 LDA #INBUFF2
944A- A0 02 2470 LDY /INBUFF2
944C- 20 52 94 2480 JSR MSGPRT
944F- 4C 03 E0 2490 JMP BASIC+3
2500 *
2510 * This routine prints out the message pointed to
2520 * by the A and Y registers.
2530 *
9452- 85 06 2540 MSGPRT STA TXTPTR
9454- 84 07 2550 STY TXTPTR+1
9456- A0 00 2560 LDY #$0
9458- B1 06 2570 LOOP2 LDA (TXTPTR),Y
945A- F0 0B 2580 BEQ ENDPRT
945C- 20 ED FD 2590 JSR COUT
945F- E6 06 2600 INC TXTPTR
9461- D0 F5 2610 BNE LOOP2
9463- E6 07 2620 INC TXTPTR+1
9465- D0 F1 2630 BNE LOOP2
9467- 60 2640 ENDPRT RTS
2650 *

```

the machine-language code that is being added to your Applesoft program so that it can be BRUN.

At this point, the program returns to Applesoft and allows you to set up your Applesoft program. This is particularly important if you intend to include some hidden lines. Once you've finished preparing your Applesoft program, all you have to do is type **VTAB 24**. This will place the cursor in the protected area of the screen and allow you to copy over the CALL command with the cursor (you do this by repeatedly pressing the right arrow key) and then pressing RETURN. Copy over the CALL 37876 only. Do not copy over the "garbage" characters at the end of the line. You don't have to worry about them being erased when you press RETURN. Since they're located in a protected portion of the screen, nothing will happen to them. After you trace over the CALL command and press return, the new binary file is then saved to the disk drive with a B. appended to the beginning of the program's original name.

The BRUN MAKER program is 570 bytes long and has been incorporated into an Applesoft program to make keying it in and using it simpler. Another advantage of this approach is that it eliminates the need to know assembly language or have an assembler available. For those of you who do know assembly language and want to see how the program was written and what is going on, the well-commented source code listing in should be just the thing you're looking for.

If you take the binary file that is produced by the BRUN MAKER program and convert it to an automatically running program as described on page 6 of the previous issue of **ASPD**, you'll have gone a long way in preventing a user from accessing your program. If you then store it on a 37 track diskette (**ASPD** Vol. 1, No. 1, p. 8) and use a customized DOS (see tutorial in this issue) you'll wind up with a diskette that will be quite difficult to backup.

GET A FREE SUBSCRIPTION

In our first issue, we offered a free six month subscription to anyone who wrote an article that we subsequently published. We had a nice response, but would like to have even more interaction with you, so we're extending that free offer another two months. So hurry up. This offer applies to full-blown articles only. Parameters for use with **COPY** or minor modifications of **COPY** will entitle you to a one-month extension for each one we use. However, a major revision of **COPY** will get you the 6-month prize. So let's see who the lucky winners will be.


```

952C- CC CC A0
952F- B3 B7 B8
9532- B7 B6      3000      .AS -" CALL 37876"
9534- 00          3010      .HS 00
9535- 8D 8D      3020 TEXT3 .HS 8D8D
9537- D7 C8 C5
953A- CE A0 D9
953D- CF D5 A7
9540- D2 C5 A0
9543- D2 C5 C1
9546- C4 D9 A0
9549- D4 CF A0
954C- C3 CF CE
954F- D4 C9 CE
9552- D5 C5 AC
9555- A0 D4 D9
9558- D0 C5      3030      .AS -"WHEN YOU'RE READY TO CONTINUE, TYPE"
955A- 8D          3040      .HS 8D
955B- A7 D6 D4
955E- C1 C2 A0
9561- B2 B4 A7
9564- A0 C1 CE
9567- C4 A0 D4
956A- C8 C5 CE
956D- A0 D4 D2
9570- C1 C3 C5
9573- A0 CF D6
9576- C5 D2 A0
9579- D4 C8 C5
957C- A0 C3 C1
957F- CC CC      3050      .AS -"'VTAB 24' AND THEN TRACE OVER THE
CALL"              .HS 8D
9581- 8D          3060
9582- D3 D4 C1
9585- D4 C5 CD
9588- C5 CE D4
958B- A0 D7 C9
958E- D4 C8 A0
9591- D4 C8 C5
9594- A0 C3 D5
9597- D2 D3 CF
959A- D2 AE      3070      .AS -"STATEMENT WITH THE CURSOR."
959C- 00          3080      .HS 00
959D- 8D 84      3090 TEXT4 .HS 8D84
959F- CC CF C1
95A2- C4 A0      3100      .AS -"LOAD "
95A4- 00          3110      .HS 00
95A5- 8D 84      3120 TEXT5 .HS 8D84
95A7- C2 D3 C1
95AA- D6 C5 A0
95AD- C2 AE      3130      .AS -"BSAVE B."
95AF- 00          3140      .HS 00
95B0- AC C1 A4
95B3- B7 C5 B9
95B6- AC CC A4   3150 TEXT6 .AS -",A$7E9,L$"
95B9- 00          3160      .HS 00
95BA- 8D 8D      3170 TEXT7 .HS 8D8D
95BC- D4 C8 C5
95BF- A0 C2 C9
95C2- CE C1 D2
95C5- D9 A0 C6
95C8- C9 CC C5
95CB- A0 C8 C1
95CE- D3 A0 C2
95D1- C5 C5 CE
95D4- A0 D3 C1
95D7- D6 C5 C4
95DA- A0 C1 D3
95DD- BA          3180      .AS -"THE BINARY FILE HAS BEEN SAVED AS:"
95DE- 8D          3190      .HS 8D
95DF- C2 AE      3200      .AS -"B."
95E1- 00          3210      .HS 00
95E2- 00          3220      .HS 00

```

continued from page 6

72 POKE 834,34
1000 DATA 3,12,224,169,96

Line 1000 puts a machine-language return code right at the beginning of the call to the nibble counting routine, totally bypassing it. By the way, if you're interested in examining the nibble counting code directly, break out of the program and get into the monitor. Then type **B619L**. This is the nibble counting routine. If you have an Apple //e or //c and can't break out of the program without it rebooting, then you can use a sector editor that disassembles code directly from the diskette (*Copy II Plus* has this feature). The nibble counting routine is located on track 3, sector 6 and starts with the 25th byte (in hex that's 19). That's all there is to it. See, that wasn't so hard.

Sensible Grammar and Bookends

Here's some tips on how to unprotect *Sensible Grammar* and *Bookends* (both of which are published by Sensible Software) from Allen L. Southmayd of San Antonio, TX. Allen says that this procedure is based on information published in *Computist* by Doni G. Grande, who used it to unprotect version 2.06 of *Bookends*. It can be used for other versions as well, he points out, but you'll have to locate the right track and sector. The same technique is used on *Sensible Grammar*, says Allen. I converted the information Allen sent in to DATA statements for use with *COPY*. Here they are:

1000 DATA 5,0,56,247,86 : REM
BOOKENDS VERSION 2.0

1000 DATA 1,13,56,247,86 : REM
BOOKENDS VERSION 2.06

1000 DATA 4,4,17,247,56 : REM SENSIBLE
GRAMMAR VERSION 1A

Allen notes that if you have the program *Essential Data Duplicator III (EDD)* that you can use that to make a backup copy of *Sensible Grammar*. To do that you copy tracks \$0 to \$22 (remember the \$ indicates hexadecimal numbers) using option 2. Then, he says, change parm 28 to \$02 or \$03 and re-copy track \$00 until the program boots properly. Thanks for the advice Allen. We're adding one month onto your **ASPD** subscription. Keep the contributions coming.

PFS Series - PRODOS Version

The new PRODOS versions of *PFS WRITE*, *GRAPH*, *REPORT* and *FILE* all use the same copy protection scheme. To disable it, all you have to do is copy the disk, and then perform a single sector edit.

```

10 TEXT : HOME
20 PRINT " BRUN MAKER FOR APPLESOFT PROGRAMS"
30 PRINT : PRINT : PRINT
40 PRINT "NOW LOADING MACHINE LANGUAGE PROGRAM"
50 PRINT "INTO MEMORY. PLEASE WAIT."
60 VTAB 9

```

continued on page 9

Tutorial

continued from page 22

line 1 or you can type in line 2 from Applesoft. Line 2 can be entered either from the direct execution mode (without a line number) or in the deferred execution mode (with a line number).

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

2. POKE 47445,170 : POKE 47455,213 :
POKE 48250,170 : POKE 48255,213

The first two locations that are modified by these lines are the READ locations and the last two are the WRITE locations. To switch back to normal DOS 3.3, simply enter either of the following lines:

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

2. POKE 47445,213 : POKE 47455,170 :
POKE 48250,213 : POKE 48255,170

You could just as easily change the bytes in the prologue of the data field instead of the address field. Or, you can change the first two epilogue bytes in either field. You can use a combination of techniques. The possibilities are almost endless, and each change you make results in a unique protected DOS.

Try your hand at making up some diskettes with a modified version of DOS. You'll be pleased to see how easy it really is. If you are going to use a modified DOS as described here, bear in mind that the level of protection afforded is limited since most nibble copy programs that are currently available can easily copy diskettes created with modified versions of DOS 3.3. However, it will prevent standard copy programs, such as COPYA, from duplicating your diskette.

continued on page 13

Table 3
Data Field DOS Locations

Location	Byte	Address	
		Hex	Decimal
Prologue Read	D5	B8E7	47335
	AA	B8F1	47345
	AD	B8FC	47356
Prologue Write	D5	B853	47187
	AA	B858	47192
	AD	B85D	47197
Epilogue Read	DE	B935	47413
	AA	B93F	47423
Epilogue Write	DE	B89E	47262
	AA	B8A3	47267

```

70 PRINT TAB( 14);: FLASH : PRINT "LOADING...": NORMAL
80 FOR X = 1 TO 570
90   READ Y
100  POKE 37799 + X,Y
110 NEXT X
120 VTAB 7
130 PRINT : PRINT
140 PRINT "THE PROGRAM IS NOW READY TO USE."
150 PRINT "TYPE 'CALL 37800' TO ACTIVATE IT."
160 DATA 32,88,252,169,131,160,148,32
170 DATA 82,148,32,111,253,169,0,157
180 DATA 64,2,202,189,0,2,157,64
190 DATA 2,232,202,208,245,165,37,141
200 DATA 128,148,230,37,169,29,160,149
210 DATA 32,82,148,169,23,133,35,173
220 DATA 128,148,133,37,169,53,160,149
230 DATA 32,82,148,169,157,160,149,32
240 DATA 82,148,169,64,160,2,32,82
250 DATA 148,32,98,252,162,0,189,104
260 DATA 148,157,233,7,240,3,232,208
270 DATA 245,56,169,233,233,1,141,129
280 DATA 148,56,165,175,237,129,148,141
290 DATA 129,148,165,176,233,7,141,130
300 DATA 148,169,165,160,149,32,82,148
310 DATA 169,64,160,2,32,82,148,169
320 DATA 176,160,149,32,82,148,173,130
330 DATA 148,32,218,253,173,129,148,32
340 DATA 218,253,32,98,252,169,24,133
350 DATA 35,169,186,160,149,32,82,148
360 DATA 169,64,160,2,32,82,148,76
370 DATA 3,224,133,6,132,7,160,0
380 DATA 177,6,240,11,32,237,253,230
390 DATA 6,208,245,230,7,208,241,96
400 DATA 169,102,141,242,3,169,213,141
410 DATA 243,3,133,214,169,112,141,244
420 DATA 3,32,88,252,76,102,213,0
430 DATA 0,0,0,141,141,160,160,160
440 DATA 194,210,213,206,160,205,193,203
450 DATA 197,210,160,198,207,210,160,193
460 DATA 208,208,204,197,211,207,198,212
470 DATA 160,208,210,207,199,210,193,205
480 DATA 211,141,141,160,160,160,160,160
490 DATA 160,160,160,160,195,207,208,217
500 DATA 210,201,199,200,212,160,168,195
510 DATA 169,160,177,185,184,182,160,194
520 DATA 217,141,160,160,160,160,160,160
530 DATA 160,160,160,160,160,160,202,213
540 DATA 204,197,211,160,200,174,160,199
550 DATA 201,204,196,197,210,141,160,160
560 DATA 160,160,160,160,160,160,160,160
570 DATA 193,204,204,160,210,201,199,200
580 DATA 212,211,160,210,197,211,197,210
590 DATA 214,197,196,141,141,141,141,197
600 DATA 206,212,197,210,160,212,200,197
610 DATA 160,198,201,204,197,160,206,193
620 DATA 205,197,186,160,0,141,141,141
630 DATA 141,141,141,141,141,141,141,141
640 DATA 141,160,195,193,204,204,160,179
650 DATA 183,184,183,182,0,141,141,215
660 DATA 200,197,206,160,217,207,213,167
670 DATA 210,197,160,210,197,193,196,217
680 DATA 160,212,207,160,195,207,206,212
690 DATA 201,206,213,197,172,160,212,217
700 DATA 208,197,141,167,214,212,193,194
710 DATA 160,178,180,167,160,193,206,196
720 DATA 160,212,200,197,206,160,212,210
730 DATA 193,195,197,160,207,214,197,210
740 DATA 160,212,200,197,160,195,193,204
750 DATA 204,141,211,212,193,212,197,205
760 DATA 197,206,212,160,215,201,212,200
770 DATA 160,212,200,197,160,195,213,210
780 DATA 211,207,210,174,0,141,132,204
790 DATA 207,193,196,160,0,141,132,194
800 DATA 211,193,214,197,160,194,174,0
810 DATA 172,193,164,183,197,185,172,204
820 DATA 164,0,141,141,212,200,197,160
830 DATA 194,201,206,193,210,217,160,198
840 DATA 201,204,197,160,200,193,211,160
850 DATA 194,197,197,206,160,211,193,214
860 DATA 197,196,160,193,211,186,141,194
870 DATA 174,0
880 REM
890 REM COPYRIGHT (C) 1986 BY
900 REM JULES H. GILDER
910 REM ALL RIGHTS RESERVED

```

CHANGE THE BLOAD ADDRESS OF BINARY FILES

When binary files are saved out to diskette under DOS 3.3 it is necessary to specify not only the length of the file to be saved, but also the address of the start of the file. When the file is loaded back into memory at a later time, it is this starting address that is used to properly place the program where it belongs. The designer's of Apple's DOS however, realized that sometimes, you might like to have the ability to save a file out from one location in memory and load it back in to another. This is particularly important when you're putting together protection schemes or when you want to modify a program that it will automatically run when it's loaded in (see **ASPD** Vol. 1, No. 1, p. 6).

To accommodate such situations, Apple made it possible to use the A or A\$ option (e.g. **BLOAD FILE,A\$300**) to **BLOAD** a file into a different address than the one it was saved from. While this technique is better than not being able to do anything at all, it nevertheless is awkward. A much more reasonable solution would have been for Apple to include a DOS command that would automatically change the information on the diskette that tells DOS where to start loading the file in.

Since Apple did not have the foresight to include such a versatile command, we'll just have to do it ourselves. To make the job simple, I've decided to use the standard DOS command parser (command interpreter). In order to do this however, it will be necessary to get rid of one of DOS's already existing commands. In the eight years that I've owned my Apple, I've never used the **CHAIN** command, so I think it's fairly safe to convert that command into one that will change the **BLOAD** address of a binary file.

Change it to CHADR

The first thing we must do is change the actual command to something that is more mnemonic (connected with the operation that's going to be performed and therefore easy to remember). Since we're going to change the address, why not change the command to **CHADR**? To do that, all we have to do is change the the last two letters of the **CHAIN** command from **IN** to **DR**. DOS has a table of all the commands stored in RAM starting at 43140 (\$A884). By **POKE**ing new values into these locations, it is therefore possible to change the commands that DOS recognizes (we'll talk more about this in the next issue of **ASPD**). The changing of the **CHAIN** command name is actually done by two simple **POKE**s in line 30 of the program listing.

Next, to keep the program out of the way, we'll store it in high memory starting at 38320 (\$95B1 in hex). Once we know where the program is going to reside in memory, we have to notify DOS so that it will know how to access it. Just as DOS held a table of com-

```

10 HIMEM: 38320
20 POKE 40230,176: POKE 40231,149
30 POKE 43158,68: POKE 43159,210
40 POKE 43281,32: POKE 43282,113
50 FOR X = 1 TO 67
60   READ Y
70   POKE 38320 + X,Y
80 NEXT X
90 PRINT : PRINT : PRINT
100 PRINT "THE 'CHAIN' COMMAND HAS BEEN CHANGED TO"
110 PRINT "'CHADR'. TO USE IT SPECIFY THE FILENAME"
120 PRINT "AND THE NEW ADDRESS. THE ADDRESS CAN"
130 PRINT "BE IN DECIMAL, OR IF PRECEDED BY A"
140 PRINT "DOLLAR SIGN ($), HEXADECIMAL."
200 DATA 173,101,170,41,15,201,1,240
210 DATA 3,76,196,166,173,114,170,141
220 DATA 244,149,173,115,170,141,245,149
230 DATA 32,168,162,173,194,181,201,127
240 DATA 144,11,169,10,141,92,170,32
250 DATA 252,162,76,213,166,41,15,201
260 DATA 4,240,4,169,13,208,237,172
270 DATA 244,149,173,245,149,32,224,163
280 DATA 76,252,162

```

mand names in RAM, it also holds a table of addresses of where each routine begins. This table starts at 40222 (\$9D1E). The jump address for the **CHAIN** command is located at 40236 (\$9D26). If we **POKE** the address of our machine language program into 40236 and 40237, then whenever the **CHADR** command is issued, it will jump to our program instead of the **CHAIN** routine. That's exactly what is done in line 20 of the program.

Before actually loading the program that implements the **CHADR** command into memory, we have one more job to do. We must now tell DOS which of the optional features of DOS are valid with our command. For example, we want to require a file name and address and also let the user specify an optional volume number, slot number and drive number. Once again, we have a table in memory that holds this information (at 43281 for the **CHAIN** command) and once again we have two **POKE**s that will set the command up the way we want it to be. This is done in line 40 of the program.

Load the routine into memory

Now that we've set up the DOS command to do exactly what we want, the only thing left to do is load in a short machine language routine that will do the actual work for us. This is done in line 50 through 80 in the **BASIC** listing.

Just as with the **BSAVE** command, the new **CHADR** command that we've implemented will permit you to use either an A or an A\$. To change the **BLOAD** address of a file to 2048 (the text screen for example), you would issue the following command:

CHADR FILENAME,A2048 or CHADR FILENAME,A\$800

That's all there is to it. I'm sure you'll find, as I did, that this will be one of your most useful DOS modifications and a big help in implementing software protection schemes.

COPYP Parameters

continued from page 7

COPYP can do the job by simply adding the line below:

```
1000 DATA 4,1,170,25,3
```

PFS PLAN uses a similar protection scheme, but it is located on a different track than the others. To make a usable unprotected copy of it type in the following line:

```
1000 DATA 1,13,170,25,3
```

Time Is Money

To make an unprotected copy of *Time Is Money*, use the following data:

```
1000 DATA 5,15,25,189,96
```

Homeword

This word processing program from Sierra On-Line can be copied by typing in the following line into the **COPYP** program:

```
1000 DATA 16,10,0,206,96
```

Homeword Speller

This is a companion spelling checker that works with *Homeword*. Unprotecting it is almost as simple as unprotecting *Homeword*, except instead of changing one byte, you'll have to change two. The following lines, when added to **COPYP** will make the unprotected copy for you.

```
1000 DATA 1,7,200,32,76
1010 DATA 1,7,201,18,220
```

Letters

Dear Editor:

Congratulations on your new publication. The quality of the material in the premier issue was excellent. If it remains that way you should have a winner. I'm rooting for you.

For the past year I've wished that I could deprotect a BASIC checkbook program (Disk-O-Check) which no longer is being sold. As a matter of fact, the company that produced it has gone out of business. My aim is to put the program onto my Sider disk where file access would be considerably more rapid. Is it your plan to consider such problems and to provide information that would enable me to accomplish this feat?

Regardless of your answer, please accept my check for a one year subscription. I am also purchasing your Assembly Language book for my library.

Sandy Mossberg
Rye Brook, NY

Your complaint is a common one Sandy. Many good programs would be much more useful if they could be put on a hard disk drive. Also, it is unfortunately too common that software companies that were here yesterday, are gone today. I am currently working on cracking Disk-O-Check for you and when it's done, I will describe how it was done in the digest so that others in a similar situation, even though the program may be different, can do it too.

Dear Editor:

In the past few months I have been searching wildly for cracking and copying information. I just read about your publication and would like you to send me a sample copy. I have been searching for something like this for months. I am interested in the challenge, fun and usefulness of hacking and cracking and I hope you can help me.

Matt St. Jean
New Milford, CT

Matt, your sample copy is in the mail. I hope you enjoy it and will become a regular reader of ours.

Dear Editor:

I have read the first issue of the Digest from cover to cover several times and find it unusually interesting and helpful. I look forward to many learning sessions and I anxiously await its delivery. Some things I would enjoy seeing in the Digest would be a letters or readers column, checksums for the program listings and available diskettes of the programs on a periodic basis at a nominal cost.

I have a problem with the print shop program that maybe one of your readers can help me with. My system is an Apple //e, Rev. B, Apple Extended 80 Col. card, Enhanced except I replaced the new character generator ROM with the old one because I

didn't like those mouse characters. I replaced the CD ROM with a 2764 EPROM burned with Don Lancaster's patch to give absolute reset capability. I have an Apple DuoDisk drive in slot 6, a Microtek RV-611c parallel card in slot 1 with an Apple DMP and an Epson RX-100, and an Applied Engineering Z-80 card in slot 7. When I run the *Print Shop* printer test I get 2 or 3 garbage characters (not always the same) and then the printer is deselected. The same thing happens when I design something with the *Print Shop* and try to print it out. My daughter runs the same disk on her //c with an Apple Scribe printer with no problems. Help!

Virgil Flint
Poway, CA

Well Virgil, here's the Letters column. A diskette with all the programs from this issue and the previous one is now available from us for only \$15, so you don't have to key all the data in yourself. We'll have a diskette available with every issue. We haven't implemented a checksum technique yet, but we're working on it. It should be in the next issue or at the latest the one after that. As far as your problems with Print Shop are concerned, I don't know what to say. We've gotten lots of letters and so have other publications, concerning the problem of printing out with the program. From all the responses I've seen, Broderbund doesn't seem to be very responsive to its customers' needs. I've printed your problem here and hopefully one of our readers will be able to help you.

Decimal/Hexadecimal Converter

continued from page 15

```

10 TEXT : HOME
20 A$ = "HEX/DECIMAL/HEX CONVERTER": GOSUB 420
30 PRINT A$ = "COPYRIGHT (C) 1986 BY": GOSUB 420
40 A$ = "JULES H. GILDER": GOSUB 420
50 A$ = "ALL RIGHTS RESERVED": GOSUB 420
60 FOR X = 1 TO 84
70   READ NUM
80   POKE 767 + X, NUM
90 NEXT X
95 CALL 768
100 PRINT : PRINT : PRINT
110 PRINT "THE HEXADECIMAL TO DECIMAL CONVERTER IS"
120 PRINT "NOW ACTIVE. TO USE IT, TYPE AN"
130 PRINT "AMPERSAND (&) AND THE NUMBER TO BE"
140 PRINT "CONVERTED. TO CONVERT FROM HEXADECIMAL"
150 PRINT "TO DECIMAL, PRECEDE THE NUMBER WITH A"
160 PRINT "DOLLAR SIGN ($)."
```

165	PRINT :	PRINT "	&768	RETURNS	\$0300	WHILE"
170	PRINT "	&\$300	RETURNS	768"		
190	END					
200	DATA 162,76,169,16,160,3,142,245					
210	DATA 3,141,246,3,140,247,3,96					
220	DATA 201,36,240,23,32,103,221,32					
230	DATA 82,231,169,164,32,237,253,165					
240	DATA 81,240,3,32,218,253,165,80					
250	DATA 76,218,253,160,0,32,177,0					
260	DATA 240,8,73,128,153,0,2,200					
270	DATA 208,243,153,0,2,168,32,167					
280	DATA 255,166,62,165,63,192,6,144					
290	DATA 3,76,153,225,192,3,176,2					
300	DATA 169,0,76,36,237					
420	L = LEN (A\$)					
430	PRINT TAB((40 - L) / 2);A\$					
440	RETURN					

Dear Editor:

Thank you for the complementary copy of your premier issue of **Apple Software Protection Digest**. I tried to use your *Print Shop* copy program and got an UNABLE TO WRITE error. I checked the drive and the diskette and everything seemed fine. Then I tried eliminating line 277 in Apple's COPY program, which is the error-trapping routine for the UNABLE TO WRITE error, and the program worked perfectly. I would appreciate your entering a subscription for me. By the way, if you have any suggestions for backing up Word Handler or for eliminating the non-standard file formats, which have a tendency to garbage disks containing other programs, I'd be grateful to hear of them.

Paul Dunseath
Ottawa, Ontario

Glad you liked our first issue Paul. I think you'll like our second one even better. I'm adding your request to our Wanted list and hopefully we'll be able to accommodate you real soon.

HIDING MACHINE LANGUAGE PROGRAMS IN STRINGS

One of the key elements involved in producing protected software is making it difficult for the software cracker to not only access your program code, but to also make it difficult for him or her to understand what's going on. Finally, you want to make it easy for the would be cracker to overlook some important code.

A technique that I have found very handy in this last regard, is to hide short machine language programs in either strings or REM statements within your BASIC program. If, after doing that, you go one step further and hide the particular line that the REM or string appears in, using any of the line hiding techniques we've already discussed, then you'll make your program significantly more difficult to crack.

The basic technique of hiding a machine language program in REMs or strings merely requires that you set the string or REM statement up to have at least the same number of characters as you have bytes in your machine-language program. You could set aside more than you need, but obviously less just won't do. To show you how it's done, let's take a simple example. I'm going to take the WIPEOUT 1 program that's described elsewhere in this issue, and hide that in a string. If you go back and look at that program, you notice that the code is position dependent (that means it was designed and assembled to work in only one specific location in the computer and thus it is not relocatable). Because of that, the specific position dependent reference, which occurs on line 1150 of the program, will have to be changed once we know the new location where the program is going to reside in memory. To avoid this sort of problem, it is best to use relocatable, or position independent programs when possible. They take a little more time and effort to produce, but they're much more flexible.

Here's how to do it step by step

To begin the step-by-step description of how to hide machine language programs in strings, let's first produce a short Applesoft BASIC program, such as the one listed below.

```
10 PRINT "THIS IS A TEST"
20 GOTO 40
30 A$ = "12345678901234567
  89012345678901234567890"
40 CALL XXXX
50 PRINT "THAT'S ALL FOLKS!"
```

You'll notice that in line 30, I've defined A\$ as a series of digits from 1 through 0 (for 10). I've found this to be very convenient, because it makes it very easy to count the number of bytes used. Another advantage of this approach is that it stores an easily distinguishable pattern of bytes in RAM memory.

When this program is run, it should print out **THIS IS A TEST** execute a machine language program that is located at XXXX and then print **THAT'S ALL FOLKS!** If the machine language program that we have it call is WIPEOUT 1, however, memory will be cleared to all zeros and the last line of the program will never be executed.

Locate the string in memory

Now we have two things to do. We must first determine where our string begins in memory so we know where to store our machine language program, and second, we must change the Xs in the CALL statement to the appropriate number. The easiest way to locate the string in memory is to use the Applesoft Line Finder program that was published in the last issue of **ASPD** (you can also use the Applesoft Line Finder and Vanisher program from this issue) and simply enter the number of the line that the string is located on. By doing that and typing an &30, you'll get the following display on your computer screen.

]&30

```
081F: 51 08 1E 00 41 24 D0 22
0827: 31 32 33 34 35 36 37 38
082F: 39 30 31 32 33 34 35 36
0837: 37 38 39 30 31 32 33 34
083F: 35 36 37 38 39 30 31 32
0847: 33 34 35 36 37 38 39 30
084F: 22 00
```

As you can easily see from this display (even if you didn't know that the 1 was stored as a 31 in hexadecimal), the string itself begins at memory location \$0827 (remember that's a hexadecimal number). The decimal equivalent of \$827 is 2087. This is the location of the start of our machine-language program and line 40 can now be changed from CALL XXXX to CALL 2087.

If you counted the number of digits we placed in A\$, you'd find that it's 40, so that is the maximum size we should allow our machine language program to be. If we have a longer program, just make the string longer. You can go up to about 230 characters. Those of you who know that strings can be as long as 255 characters may question this discrepancy. The limit is due to the fact that Applesoft limits the length of a line that can be typed in to 239 characters. If you take away what you need for the line number and the A\$= you're left with about 230 (actually 233).

Load the program into the string

To load the machine language program into the string you can key it in directly as it's real short. The best thing, however is to BLOAD it in from a diskette, using the des-

tinuation (A\$) parameter. If for example, I had my program stored on a diskette as WIPEOUT 1, then to load it where it belongs in my BASIC program I would have to type BLOAD WIPEOUT 1,A\$827. This would load my machine language program right into the string in my Applesoft BASIC program. Alternatively, I could have typed in (from the monitor mode) the following to load the program in:

```
*827:A0 00 84 D6 B9 E2 02 99
00 02 C8 C9 82 D0 F5 4C 7G FF
B8<RETURN>
```

```
*:B0 B0 BA B0 A0 CE A0 B8 B0
B1 BC B8 B0 B0 AE B9 B5 C6 C6
CD 82<RETURN>
```

Once we have the machine code in its proper location, if it's not relocatable, we'll have to patch it so that it can run there. For WIPEOUT 1, that means we'll have to change the two bytes located at \$82C. We do this by typing CALL -151 to get into the monitor mode if we're not already in it, and then type:

```
* 82C:39 08 <RETURN>
```

Now type 3D0G to get back to Applesoft and then type LIST. Your program should look like this:

```
10 PRINT "THIS IS A TEST"
20 GOTO 40
30 A$ = " COLOR=
40 CALL 2087
50 PRINT "THAT'S ALL FOLKS!"
```

Notice that line 30 has changed and now has a strange looking statement in it. Don't worry about it. It may look strange, but it works fine. At this point, it's very important to tell you that absolutely no changes should be made to your Applesoft program from now on. Don't add any lines and don't delete any lines. If you do, you will mess up your program irretrievably. This, by the way, is another plus for hiding critical machine language routines in strings, because it makes changing a program virtually impossible without destroying it.

Ordinary saves won't work

Another thing you should be aware is that you shouldn't count on being able to save your program to a diskette this way, because it will be destroyed after it's loaded back into memory. The reason for this is that there are some zeros in the assembly language program, and Applesoft interprets them as end of line markers with faulty links to the next line of the Applesoft program. When Applesoft tries to correctly link the program (which occurs when a program is loaded or

continued on page 14

TWO SOLUTIONS TO THE HIDE-A-LINE PROBLEM

by Mark Landwehr

After seeing the challenge offered in the first issue regarding hidden program lines, I put together two machine-language programs that will hide any Applesoft program lines that are desired. The programs use two of the different techniques that were described in the first issue of **Apple Software Protection Digest**.

In the first program, Applesoft Line-Hide #1, all the lines that are to be hidden must begin with 5 colons, as was described in the original article. The program has two modes of operation. In one, you can specify individual lines that should be hidden, while in the second, all lines that have five colons in front of them will be hidden.

The program makes use of several Applesoft ROM routines and also uses the & to both jump to the machine language routine and pass parameters (in this case the line number) to it.

Set up the ampersand jump

The program is activated by typing **CALL 768**. It immediately returns you to Applesoft and it appears as if nothing has happened. In fact, however, the program set up the ampersand (&) jump parameters (this is done by the code marked INIT in the listing) so that the working part of the program (labelled START) will be jumped to every time the & key is entered.

The first thing that the program does once it's called via the &, is a subroutine jump to Applesoft's CHRGOT routine. This is a very short routine that is located on page zero and is used everytime an Applesoft command is entered. Applesoft used it to recognize that the ampersand was entered, and it left an internal pointer set to the next character after the &. If there was nothing else entered, the pointer will be pointing to a zero. If, however, a line number was entered too, the pointer will be pointing to it.

A jump to the CHRGOT routine will cause the accumulator to be loaded with whatever is being pointed to. If it turns out that the accumulator contains a value other than zero, we know a line number was entered too, and the program branches to the code that handles individual lines (the section marked ONE in the listing). If a zero was loaded into the accumulator, then the program assumes that the user wants all lines that have five colons in front of them to be hidden and falls into the subroutine, called ALL, that does this.

Whether the ONE or ALL subroutines are used, both of them make use of the CHECK subroutine, which inspects the line being processed to see if it has five colons in front of it. If it does, the first colon is replaced with a zero to implement the hiding, otherwise, the line is ignored.

```

1000 *****
1010 *
1020 *  A P P L E S O F T   L I N E - H I D E  # 1  *
1030 *
1040 *  Will make any or all program lines that begin *
1050 *  with 5 colons disappear. Line numbers will *
1060 *  remain visible, but with no program data... *
1070 *
1080 *                      COMMAND SYNTAX
1090 *
1100 *          & = all lines with 5 colons
1110 *          &<line#> = desired line only
1120 *
1130 *****
1140 *
1150 *
1160 *
0000- 1170 PTR          .EQ $00          storage pointer
0067- 1180 TXTTAB      .EQ $67          start of BASIC program
009B- 1190 LOWTR       .EQ $9B          pointer used by FNDLIN
00B7- 1200 CHRGOT      .EQ $B7          fetch data routine
03F6- 1210 AMPER       .EQ $3F6        ampersand vector
D61A- 1220 FNDLIN      .EQ $D61A       search for line number
DA0C- 1230 LINGET      .EQ $DA0C       put line number in zero page
1240 *
1250 *
1260 *                      .OR $300
1270 *
1280 * This routine initializes the ampersand jump vector.
1290 *
0300- A9 0B 1300 INIT      LDA #START
0302- A0 03 1310          LDY /START
0304- 8D F6 03 1320          STA AMPER
0307- 8C F7 03 1330          STY AMPER+1
030A- 60 1340          RTS
1350 *
1360 * Check entry after the "&". If nothing, do all lines.
1370 * If there's a number, then execute only on that line.
1380 *
030B- 20 B7 00 1390 START   JSR CHRGOT      Get entry.
030E- D0 1F 1400          BNE ONE          Handle a single line
1410 *
1420 * Hide all lines in program that begin with 5 colons.
1430 *
0310- A5 67 1440 ALL       LDA TXTTAB      Load start of program
0312- A4 68 1450          LDY TXTTAB+1    pointers.
0314- 85 00 1460          STA PTR          Save pointers to zero
0316- 84 01 1470          STY PTR+1      page locations.
0318- A0 00 1480 LOOP1     LDY #$0        Initialize index.
031A- B1 00 1490          LDA (PTR),Y     Get lo-byte link
031C- 48 1500          PHA              and save it.
031D- C8 1510          INY
031E- B1 00 1520          LDA (PTR),Y     Get hi-byte link
0320- 48 1530          PHA              and save it too.
0321- F0 35 1540          BEQ DONE        If it is 0, end prgrm
0323- 20 44 03 1550          JSR CHECK    Check for 5 colons.
0326- 68 1560          PLA              Retrieve hi-byte link.
0327- 85 01 1570          STA PTR+1      Store it.
0329- 68 1580          PLA              Retrieve lo-byte link.
032A- 85 00 1590          STA PTR          Store it.
032C- B8 1600          CLV              Always go and look at
032D- 50 E9 1610          BVC LOOP1      the next line.
1620 *
1630 * Hide only the line that was specified by the user.
1640 *
032F- 20 0C DA 1650 ONE     JSR LINGET    Put line# in LINNUM.
0332- 20 1A D6 1660          JSR FNDLIN   Find line in program.
0335- 90 23 1670          BCC DONE2      Clear carry - no line.
0337- A5 9B 1680          LDA LOWTR      Load address of the
0339- A4 9C 1690          LDY LOWTR+1    desired line.
033B- 85 00 1700          STA PTR          Store it in zero
033D- 84 01 1710          STY PTR+1      page locations.
033F- 20 44 03 1720          JSR CHECK    Check for 5 colons.
0342- F0 16 1730          BEQ DONE2      Return to caller.
1740 *
1750 * Check for 5 colons at the start of the line, and, if
1760 * found, put a zero in place of first colon.
1770 *
0344- A0 04 1780 CHECK      LDY #$4        Initialize index.
0346- B1 00 1790 LOOP2     LDA (PTR),Y     Get line data byte.
0348- C9 3A 1800          CMP #' '       Is it a colon?

```

Hide lines without colons too

If you have already written a program and don't want to go back through it to add colons to it, you can use the second program, Applesoft Line-Hide #2. This program will hide any line or group of lines, regardless of whether there are five colons at the beginning or not. In fact, you might want to use both programs together to hide your program lines.

This program is somewhat similar to the first one in that it too uses the ampersand (&) to activate the program. In this case however, entering just the & by itself will give you a SYNTAX ERROR. This program requires that you enter two line numbers, the numbers of the lines that surround the one(s) you wish to hide. These two line numbers must be separated by a comma. Thus, in a program with lines numbered 10, 20 and 30, to hide line 20, you would type & 10,30. To hide several lines, just chose your first and last lines so that they surround the range to be hidden. The last line in your program can be hidden too, by simply specifying a larger, nonexistent line number for the second parameter in the ampersand command.

Since the command syntax for both of these programs is different, it would not be difficult to combine both of them into one program. Personally, I think that this may be a little confusing, which is why I left them as separate programs.

Thanks for your entry Mark. You won the best program award and will get an extra 6 months of ASPD.

Tutorial

continued from page 8

After going through all of this article and reaching this point, you just might be asking yourself, "If this copy protection scheme can be copied with a nibble copier, what good is it?" The answer is simple. Knowledge is power. If you understand how this scheme works, you can incorporate it with a variety of other techniques that we'll discuss in the coming months and be able to produce a diskette that will be awfully difficult to copy. I say awfully difficult, because nothing can be 100% impossible to copy (although recently I've seen some programs that sure seem like it). More importantly, if you're trying to unprotect or back up a disk that you've purchased, you'll want to know what to look for. Now at least you have the basic knowledge that you need.

Next time we'll look at a special type of byte that is used on diskettes and causes a lot of problems for those interested in overcoming copy protection. This is known as the sync byte. In the mean time, try experimenting with what you've learned so far. If you come up with any interesting cracking or protecting techniques, let us know about them. If we publish yours, you'll get a free six-month extension to your current subscription.

```

034A- D0 OE      1810      BNE DONE2      No, then forget it.
034C- C8         1820      INY
034D- C0 09      1830      CPY #59      Done 5 checks yet?
034F- 90 F5      1840      BCC LOOP2      No, do more.
0351- A0 04      1850      LDY #54      Yes, back to 1st byte.
0353- A9 00      1860      LDA #0       Get ready to store
0355- 91 00      1870      STA (PTR),Y  a zero there.
0357- 60         1880      RTS
                  1890 *
1900 * Exit routine. If entered from ALL routine, link bytes
1910 * must be pulled off of the stack. If from the ONE
1920 * routine, just return to the calling routine.
1930 *
0358- 68         1940 DONE      PLA
0359- 68         1950          PLA
035A- 60         1960 DONE2     RTS

```

```

1000 *****
1010 *
1020 *  A P P L E S O F T   L I N E - H I D E # 2
1030 *
1040 * Will hide any program line that user desires.
1050 * User must enter line preceding hidden line
1060 * and line immediately following hidden line.
1070 * (Also works for a group of lines)
1080 *
1090 *      COMMAND SYNTAX
1100 *
1110 *      &<line# preceding>,<line# following>
1120 *
1130 *****
1140 *
1150 *
1160 *
0000- 1170 PTR      .EQ $00      pointer for first byte
0002- 1180 PTR2     .EQ $02      pointer for second byte
009B- 1190 LOWTR    .EQ $9B      pointer used by FNDLIN
03F6- 1200 AMPER    .EQ $3F6     ampersand vector
D61A- 1210 FNDLIN   .EQ $D61A    search for line number
DA0C- 1220 LINGET   .EQ $DA0C    put line number in zero page
DEBE- 1230 CHKCOM   .EQ $DEBE    check for comma
1250 *
1260 *      .OR $300
1270 *
1280 * This routine initializes the ampersand jump vector.
1290 *
0300- 1300 INIT     LDA #BEGIN
0302- 1310          LDY /BEGIN
0304- 1320          STA AMPER
0307- 1330          STY AMPER+1
030A- 1340          RTS
1350 *
1360 * Get the first (preceding) line number and store
1365 * pointers.
1370 *
030B- 1380 BEGIN    JSR LINGET    Get the line number.
030E- 1390          JSR FNDLIN    Find it in memory.
0311- 1400          LDA LOWTR     Load pointers to the
0313- 1410          LDY LOWTR+1   desired line.
0315- 1420          STA PTR       Save to zero page
0317- 1430          STY PTR+1     locations.
0319- 1440          JSR CHKCOM    Check for a comma.
1450 *
1460 * Get the second (following) line number and store
1465 * pointers.
1470 *
031C- 1480          JSR LINGET    Get the line number.
031F- 1490          JSR FNDLIN    Find it in memory.
0322- 1500          LDA LOWTR     Load pointers to the
0324- 1510          LDY LOWTR+1   desired line.
0326- 1520          STA PTR2      Save to zero page
0328- 1530          STY PTR2+1    locations.
1540 *
1550 * Take the address of the link bytes of the second
1560 * (following) line and store them as the link bytes
1570 * in the first (preceding) line.
1580 *
032A- 1590          LDY #0         Initialize the index.
032C- 1600          LDA PTR2      Lo-byte link of second line
032E- 1610          STA (PTR),Y   stored in first line number.
0330- 1620          INY
0331- 1630          LDA PTR2+1    Hi-byte link of 2nd line number
0333- 1640          STA (PTR),Y   stored in first line number.
0335- 1650          RTS

```


HIDING PROGRAM LINES FROM BASIC

by Eric Wachtenheim

I recently bought your assembly language book, *Now That You Know Apple Assembly Language: What Can You Do With It?* With it I got a free copy of your **Apple Software Protection Digest** and read the article on finding Applesoft program lines. I also saw the challenge of making a program that automatically hides the lines and saw the 6 free issue bonus for completing this program and set at it.

My first step towards making this program was analyzing the format in which the Apple stores its Applesoft programs. I learned by looking through memory that the first two bytes of a line point in Lo/Hi byte format to the next line. After that was two bytes for the line number, also in Lo/Hi byte format, and then the tokens that represent the various BASIC keywords (e.g. PRINT, REM, GOTO, etc.) I saw for each line with five colons at the beginning that there were five "3A"s right after the line number. Seeing this I decided to check if the first five characters of each line were "3A"s and if they were, I change the first 3A to a 00. I proceeded to

do this using the next-line pointer to find out where the next line began. The result is the program below. It must be appended to the program on which you wish to hide the lines. It can be activated by typing **RUN 63000**.

This magazine reminds me of the very beginning of another protection oriented magazine which has grown to be very informative and popular. This rapid growth is in my

opinion do to the contributions of the readers. So come on all you **ASPD** readers, contribute!!

Thanks for your contribution Eric. Since you were the only one to submit a BASIC program to hide lines, the 6-month subscription prize is yours. Keep the contributions coming.

```

63000 REM *****
63010 REM *      LINE HIDER      *
63020 REM * BY ERIK WACHTENHEIM *
63030 REM *****
63040 REM
63050 REM LINES TO BE HIDDEN MUST
63060 REM BE PRECEDED BY 5 COLONS
63070 REM THIS SUBROUTINE MUST BE
63080 REM APPENDED TO THE TARGET
63090 REM PROGRAM AND CALLED FROM
63100 REM THERE WITHIN. GOOD LUCK
63110 REM
63120 BASE = 2049
63130 IF PEEK (BASE) = 0 AND PEEK (BASE + 1) = 0 THEN 63170
63140 IF PEEK (BASE + 4) = 58 AND PEEK (BASE + 5) = 58 AND
    PEEK (BASE + 6) = 58 AND PEEK (BASE + 7) = 58 AND PEEK
    (BASE + 8) = 58 THEN POKE BASE + 4,0
63150 BASE = PEEK (BASE) + 256 * PEEK (BASE + 1)
63160 GOTO 63130
63170 END : REM IF USING AS A SUBROUTINE PLACE A RETURN HERE
    INSTEAD OF AN END.
```

Hiding Machine Language

continued from page 11

a line is added or deleted), your machine-language program will get clobbered.

The only way you can save this program out to a diskette is to use the BRUN MAKER program. To do this, RUN BRUN MAKER and have it load in your Applesoft program. When BRUN MAKER returns you to the Applesoft mode, BLOAD your machine language program where it belongs and then from the immediate mode (with no line number) type **VTAB 24** and trace over the CALL statement. Your Applesoft program with the machine language program in the string will now be safely recorded on a diskette.

To check out the program, just type **RUN**. The computer will type out the message in line 10, delay for a few seconds, and then return with the Applesoft prompt (>). It never got to the last line of the program, because the call to the machine language routine erased all of memory. Go ahead, try to list the program. It's not there. Unlike the NEW or FP commands which only erase pointers to the program (allowing it to be reconstructed by an ambitious cracker), WIPEOUT 1, erased every memory location from the beginning of BASIC to the beginning of DOS. There's no way that the program can be resurrected.

While the example that I have given was for storing programs in strings, the identical technique can be used to store machine language program in REM statements.

continued from page 13

```

10 REM BASIC PROGRAM TO INSTALL
20 REM APPLESOFT LINE-HIDE #1
30 REM
40 TEXT : HOME
50 PRINT "APPLESOFT LINE-HIDE #1": VTAB 5
60 FOR X = 1 TO 90
70   READ Y
80   POKE 767 + X,Y
90 NEXT X
100 PRINT : PRINT : PRINT "INSTALLATION COMPLETE."
110 CALL 768
120 DATA 169,11,160,3,141,246,3,140
130 DATA 247,3,96,32,183,0,208,31
140 DATA 165,103,164,104,133,0,132,1
150 DATA 160,0,177,0,72,200,177,0
160 DATA 72,240,53,32,68,3,104,133
170 DATA 1,104,133,0,184,80,233,32
180 DATA 12,218,32,26,214,144,35,165
190 DATA 155,164,156,133,0,132,1,32
200 DATA 68,3,240,22,160,4,177,0
210 DATA 201,58,208,14,200,192,9,144
220 DATA 245,160,4,169,0,145,0,96
230 DATA 104,104,96
```

```

10 REM BASIC PROGRAM TO INSTALL
20 REM APPLESOFT LINE-HIDE #2
30 REM
40 TEXT : HOME
50 PRINT "APPLESOFT LINE-HIDE #2": VTAB 5
60 FOR X = 1 TO 53
70   READ Y
80   POKE 767 + X,Y
90 NEXT X
100 PRINT : PRINT : PRINT "INSTALLATION COMPLETE."
110 CALL 768
120 DATA 169,11,160,3,141,246,3,140
130 DATA 247,3,96,32,12,218,32,26
140 DATA 214,165,155,164,156,133,0,132
150 DATA 1,32,190,222,32,12,218,32
160 DATA 26,214,165,155,164,156,133,2
170 DATA 132,3,160,0,165,2,145,0
180 DATA 200,165,3,145,0,96
```

A HANDY DECIMAL/HEXADECIMAL CONVERTER

As you get more and more involved in protecting and unprotecting programs you'll find yourself using hexadecimal numbers more frequently. You'll find that having a quick and easy way of converting numbers back and forth between decimal and hexadecimal will become essential. If you've got a *Texas Instruments Programmer* calculator or a pad of our *Programmer's Number Conversion System* forms, then you've got the problem licked. If not, you'll probably find this short utility program to be extremely useful. It can reside in memory while you're working on other programs and be instantly called up by using the ampersand (&) character.

Of course, you can write a BASIC program to do the conversion, but that would mean it couldn't be conveniently available for use while you're working on other BASIC programs. The alternative is to use a machine-language program to do the conversions for you. That's where the HEX/DECIMAL/HEX CONVERTER program comes in. This program will allow you to convert numbers in either direction. An added advantage of the program is that it does not have to be used in an Applesoft program only, but can also be used in the immediate mode. A fully-commented assembly listing of the program and an Applesoft program to load and activate the converter are provided. *continued on page 10*

Editorial

continued from page 1

Free unclassified ads

Starting with the next issue, we're going to add an Unclassified Ad section to the Digest. Subscribers are each entitled to one free unclassified ad. Ads must be typed double spaced, contain no more than 50 words and include full name and address. Ads are printed on a first-come, first-served, space available basis.

Protection consultants wanted

Frequently we get requests from people who are interested in hiring a consultant to implement a copy protection scheme. If you're knowledgeable in this area and are available for consulting work, please write us a letter stating your experience in this area, and what your consulting fee is. Be sure to include your name, address and phone number. We will place you on a list that will go out to all those who inquire. While our Digest deals only with the Apple computer, several of the requests we've had deal with other computers as well, so please list all the computers that you offer protection consulting for.

Jules H. Gilder
Publisher & Editor

```

1000 *****
1010 ***
1020 ***   HEX/DECIMAL/HEX CONVERTER   ***
1030 ***
1040 ***   COPYRIGHT (C) 1982 BY   ***
1050 ***   JULES H. GILDER   ***
1060 ***   ALL RIGHTS RESERVED   ***
1070 ***
1080 *****
1090 *
1120 *       .OR $300
1140 *
1150 * EQUATES
1160 *
003E- 1170 A2L      .EQ $3E
0050- 1180 LINNUM   .EQ $50
00B1- 1190 CHRGET   .EQ $B1
0200- 1200 IN       .EQ $200
03F5- 1210 AMPERSD  .EQ $3F5
DD67- 1220 FRMNUM   .EQ $DD67
E199- 1230 IQERR    .EQ $E199
E752- 1240 GETADR   .EQ $E752
ED24- 1250 LINPRT   .EQ $ED24
F941- 1260 PRNTAX   .EQ $F941
FDDA- 1270 PRBYTE   .EQ $FDDA
FDED- 1280 COUT     .EQ $FDED
FFA7- 1290 GETNUM   .EQ $FFA7
1300 *
1320 * This is where the ampersand (&) vector
1330 * jump is set up.
1340 *
0300- A2 4C      1350      LDX #$4C      Get JMP op code and
0302- A9 10      1360      LDA #START    the low and high bytes
0304- A0 03      1370      LDY /START    of START's address and
0306- 8E F5 03   1380      STX AMPERSD   store them in locations
0309- 8D F6 03   1390      STA AMPERSD+1 $3F5, $3F6 and $3F7.
030C- 8C F7 03   1400      STY AMPERSD+2
030F- 60         1410      RTS
1430 *
1440 * This part of the program checks to
1450 * see if the character immediately following
1460 * the ampersand (&) was a dollar sign.
1470 * If it was, control is passed to the
1480 * routine that converts from hexadecimal
1490 * to decimal. Otherwise the number is
1500 * decimal and converted to hexadecimal.
1510 *
0310- C9 24      1520 START    CMP #$24      Is it a dollar sign ($) ?
0312- F0 17      1530      BEQ HEXIN    Yes, convert hex to decimal.
0314- 20 67 DD   1540      JSR FRMNUM   No, evaluate number or formula.
0317- 20 52 E7   1550      JSR GETADR    Convert to integer form.
031A- A9 A4      1560      LDA #$A4      Output a dollar sign ($).
031C- 20 ED FD   1570      JSR COUT
031F- A5 51      1580      LDA LINNUM+1  Get most high byte.
0321- F0 03      1590      BEQ PRINTLO   If zero, get low byte.
0323- 20 DA FD   1600      JSR PRBYTE   Otherwise print high byte.
0326- A5 50      1610 PRINTLO  LDA LINNUM   Get low byte.
0328- 4C DA FD   1620      JMP PRBYTE   Print it.
1630 *
1640 *
1650 * This routine handles the hexadecimal
1660 * to decimal conversion.
1670 *
032B- A0 00      1680 HEXIN   LDY #$0      Zero offset index.
032D- 20 B1 00   1690 HEXIN2  JSR CHRGET   Get the next character.
0330- F0 08      1700      BEQ PUTBUF   Store in buffer and convert.
0332- 49 80      1710      EOR #$80      Set high bit.
0334- 99 00 02   1720      STA IN,Y     Store in input buffer.
0337- C8         1730      INY          Increment offset index.
0338- D0 F3      1740      BNE HEXIN2   Get next character.
033A- 99 00 02   1750 PUTBUF   STA IN,Y     Store zero in buffer.
033D- A8         1760      TAY          Zero offset index.
033E- 20 A7 FF   1770      JSR GETNUM   Convert ASCII to hex.
0341- A6 3E      1780      LDX A2L      Store low byte in X-register.
0343- A5 3F      1790      LDA A2L+1  Store high byte in Y-register.
0345- C0 06      1800      CPY #$6      Check if number too large.
0347- 90 03      1810      BCC INRANGE  No, it's okay.
0349- 4C 99 E1   1820      JMP IQERR    Yes, print error message.
034C- C0 03      1830 INRANGE  CPY #$3      Converting only 1 byte?
034E- B0 02      1840      BCS PRINTIT No, do both.
0350- A9 00      1850      LDA #$0      Yes, do just one.
0352- 4C 24 ED   1860 PRINTIT JMP LINPRT   Convert and print number.

```

APPLESOFT LINE FINDER AND VANISHER

by Adam Levin

I'll take a shot at seven free issues of your fine magazine! I have written a routine to automatically hide BASIC program lines that have five colons in front of them. The routine is short, sweet and simple, and it even overwrites those four pesky 3As (the hex code for a colon).

To save time and effort, the program has been written as an addition to the Applesoft Line Finder program that was published in the last issue. All you have to do is add the lines listed below to that file and the resulting program will not only display the BASIC line as it's stored in memory, but it will also prevent it from being listed out from BASIC.

As an extra added bonus, my routine will add from one to four characters in place of from one to four 3As. The characters to be substituted for the 3As, are typed directly in from the keyboard. If you don't want to change the 3As, just press RETURN. Also note, that my additions will never alter a line which doesn't begin with a colon (which no normal line needs to do).

You did a nice job Adam, but one thing you forgot to do was to check that all five colons were present. If they're not (let's say only four are present) the modifications that your program makes will cause the BASIC program to crash. This is not the best program submitted, but we will reward your effort by giving you a free three month subscription. Let's see what else you can do. For those of you who missed the last issue, I've included a BASIC program listing that implements the Applesoft Line Finder with Adam Levine's modification to it.

CRACKS WANTED

Every month we will devote one section of the magazine to listing programs that you, our readers would like to see cracked (unprotected). Anyone submitting a technique for unprotecting any of these programs will receive 2 months of the Digest for free. Below is a short list of program cracks requested so far, so let's see you get to work on these.

1. Sensible Speller — DOS 3.3
2. Sensible Speller — ProDOS
3. Newsroom
4. Disk-O-Check
5. Dazzle Draw
6. Crush, Crumble and Chomp
7. Wizardry
8. Word Handler

1120	.OR \$2C0	
1255	RDKEY .EQ \$FDOC	Monitor read routine.
1660	START JSR LINGET	These 3 lines are from your
1670	JSR FNDLIN	original program, just to show
1680	BCC NOLINE	where my lines get inserted.
1681	LDY #\$4	Offset to skip next line pointer & number.
1682	LDA (LOWTR),Y	Is 1st character a colon?
1683	EOR #\$3A	If it is, make it zero.
1684	BNE .1	It's not continue Jules' routine.
1685	.0 STA (LOWTR),Y	Store zero in line, and
1686	INY	get ready to index to next byte.
1687	CPY #\$9	Past all possible colons?
1688	BCS .1	Yes, continue Jules' routine.
1689	STY TEMP	No, save index for later.
1690	JSR RDKEY	Get a character to put in place
1691	JSR COUT	of colon and echo it to screen.
1692	LDY TEMP	Restore the index.
1693	CMP #\$8D	<RETURN> key?
1694	BNE .0	No, put it into line & repeat.
1695	.1 LDY #0	This is your line with local label.
1880	RTS	Return to BASIC instead of monitor.

```

10 TEXT : HOME : PRINT "APPLESOFT LINE VANISHING PROGRAM": VTAB 5
20 FLASH : PRINT "INSTALLING MACHINE LANGUAGE PROGRAM."
25 NORMAL
30 FOR X = 1 TO 272
40   READ Y
50   POKE 703 + X,Y
60 NEXT
70 VTAB 5: PRINT "INSTALLATION HAS BEEN COMPLETED. TO USE"
75 PRINT "THE PROGRAM, TYPE '& LINENUMBER'. IF THE";
80 PRINT "LINE IS PRECEDED BY 5 COLONS, IT WILL BE";
85 PRINT "HIDDEN AND A HEX DUMP OF THE LINE WILL"
90 PRINT "BE DISPLAYED. IF THE 5 COLONS ARE NOT"
92 PRINT "PRESENT, JUST THE HEX DUMP WILL BE"
93 PRINT "DISPLAYED."
94 PRINT : PRINT "PRESS ANY KEY TO CONTINUE";: GET A$: PRINT
96 CALL 704
100 DATA 32,88,252,169,218,160,2,32
110 DATA 169,3,162,76,169,47,160,3
120 DATA 142,245,3,141,246,3,140,247
130 DATA 3,96,193,208,208,204,197,211
140 DATA 207,198,212,160,204,201,206,197
150 DATA 160,198,201,206,196,197,210,141
160 DATA 141,194,217,160,202,213,204,197
170 DATA 211,160,200,174,160,199,201,204
180 DATA 196,197,210,141,195,207,208,217
190 DATA 210,201,199,200,212,160,168,195
200 DATA 169,160,177,185,184,178,141,193
210 DATA 204,204,160,210,201,199,200,212
220 DATA 211,160,210,197,211,197,210,214
230 DATA 197,196,141,141,141,141,0,32
240 DATA 12,218,32,26,214,144,73,160
250 DATA 4,177,155,73,58,208,21,145
260 DATA 155,200,192,9,176,14,132,8
270 DATA 32,12,253,32,237,253,164,8
280 DATA 201,141,208,235,160,0,132,8
290 DATA 32,135,3,169,160,32,237,253
300 DATA 177,155,208,8,165,8,201,5
310 DATA 176,13,169,0,32,218,253,32
320 DATA 160,3,202,240,227,208,228,169
330 DATA 0,32,218,253,32,142,253,96
340 DATA 169,191,160,3,76,169,3,32
350 DATA 142,253,169,160,32,237,253,162
360 DATA 8,165,156,32,218,253,165,155
370 DATA 32,218,253,169,186,76,237,253
380 DATA 230,155,208,2,230,156,230,8
390 DATA 96,133,24,132,25,160,0,177
400 DATA 24,240,11,32,237,253,230,24
410 DATA 208,245,230,25,208,241,96,141
420 DATA 206,207,160,211,213,195,200,160
430 DATA 204,201,206,197,135,141,0,0

```


WIPEOUT: THE ULTIMATE WEAPON OF DESTRUCTION

The ultimate method of preventing unauthorized access to your program is to completely wipe it out of memory. I don't mean you should implement a NEW or even an FP command, if the program is in BASIC. That only changes the program pointers, and leaves the bulk of the program intact in memory, ready for the would be cracker to resurrect it. And if you did take that approach, how would you handle machine code programs? No, the ultimate answer to getting rid of your program is to totally obliterate it by filling all of memory with zeros, or some other byte if you've a mind to.

Drastic as this may sound, implementing the actual task is really quite simple and requires only very short and simple machine language programs. Of course, a BASIC routine could be used to do most of the job, but that would be easily detectable, and would take quite a bit of time too. No, machine language is definitely the right choice for this job, and here's how to do it.

Two different approaches

Since the task is really quite simple to implement, I will show you two different ways to do it. Both have their advantages and disadvantages. The first program is called, appropriately enough, WIPEOUT 1, and while it is the longer one of the two, it is also more flexible and harder to detect.

WIPEOUT 1 makes use of the fact that if we stuff the input buffer, which starts at location 512 (\$200), with a valid string of monitor commands and then jump to the appropriate place in the F8 ROM, that series of commands will be implemented. This approach to wiping out memory is quite unconventional from a programming point of view, which is one of the reasons it's harder to detect (people don't know what they're looking at). For those of you who have done some work while in the Apple's monitor mode however, you'll recognize this as being a quite common approach to the situation.

Looking at the assembly language source listing you'll notice that the first thing the routine does is to store a zero in the auto-run flag. I've included that just in case your program set it, so that you don't have to remember to reset it. The rest of the code simply moves the command string from its current location into the input buffer. And finally, line 1200 jumps to the ROM routine that causes the command in the input buffer to be executed.

If you look at line 1220, you notice that the command string ends with a hex \$82. This is the code for a Control-C, and it cause the computer, which would normally return to the asterisk prompt in the monitor mode, to instead return to the Applesoft mode. As long as this short program resides somewhere in the \$800 to \$95FF memory range, the pro-

```

1000 *****
1010 ***
1020 ***      WIPEOUT 1      ***
1030 ***
1040 *****
1050 *
1060 *
1070      .OR $2D0
1080 *
00D6- 1090 RUNFLAG      .EQ $D6
FF70- 1100 EXECUTE      .EQ $FF70
      1110 *
      1120 *
02D0- A0 00      1130      LDY #$0
02D2- 84 D6      1140      STY RUNFLAG
02D4- B9 E2 02   1150 LOOP    LDA STRING,Y
02D7- 99 00 02   1160      STA $200,Y
02DA- C8         1170      INY
02DB- C9 82      1180      CMP #$82
02DD- D0 F5      1190      BNE LOOP
02DF- 4C 70 FF   1200      JMP EXECUTE
02E2- B8 B0 B0
02E5- BA B0 A0
02E8- CE A0 B8
02EB- B0 B1 BC
02EE- B8 B0 B0
02F1- AE B9 B5
02F4- C6 C6 CD   1210 STRING .AS -"800:0 N 801<800.95FFM"
02F7- 82         1220      .HS 82

```

```

1  REM BASIC PROGRAM TO INSTALL
2  REM WIPEOUT 2 PROGRAM
3  REM
30  TEXT : HOME
40  FOR X = 1 TO 26
50    READ Y
60    POKE 543 + X,Y
70  NEXT X
80  PRINT : PRINT "INSTALLATION COMPLETE."
90  PRINT : PRINT "CALL 544 TO USE WIPEOUT 2"
100 DATA 169,8,133,7,169,0,168,133
110 DATA 6,133,214,145,6,200,208,251
120 DATA 230,7,165,7,201,150,208,236
130 DATA 96,82

```

```

1000 *****
1010 ***
1020 ***      WIPEOUT 2      ***
1030 ***
1040 *****
1050 *
1060 *
1070      .OR $220
1080 *
0006- 1090 POINTER      .EQ $06
00D6- 1100 RUNFLAG      .EQ $D6
9600- 1110 STARTDOS      .EQ $9600
      1120 *
      1130 *
0220- A9 08      1140      LDA #$8
0222- 85 07      1150      STA POINTER+1
0224- A9 00      1160 LOOP1   LDA #$0
0226- A8         1170      TAY
0227- 85 06      1180      STA POINTER
0229- 85 D6      1190      STA RUNFLAG
022B- 91 06      1200 LOOP2   STA (POINTER),Y
022D- C8         1210      INY
022E- D0 FB      1220      BNE LOOP2
0230- E6 07      1230      INC POINTER+1
0232- A5 07      1240      LDA POINTER+1
0234- C9 96      1250      CMP /STARTDOS
0236- D0 EC      1260      BNE LOOP1
0238- 60         1270      RTS

```

gram will not only wipe out the program you want to protect, but itself as well. To make it easy for you to key in and use the program, it has been converted to a short Applesoft BASIC program.

memory wipeout begins, the autorun flag is reset to zero.

Once again for your convenience, the machine language program has been convert-

ed to a BASIC one. By using either one of these techniques, you can make it much harder for prying eyes to find out what you're doing.

More conventional destruction

A more conventional way of wiping out memory is shown in the program. This is probably the way most programmers would approach the task if asked to implement it. The advantage of this approach is that it is much shorter than the other one. It takes only 26 bytes instead of the 40 required by the other routine. This might be an important factor if memory space is of the essence, as it so often is. Again, before the actual

```

1  REM BASIC PROGRAM TO INSTALL
2  REM WIPEOUT 1 PROGRAM
3  REM
30 TEXT : HOME
40 FOR X = 1 TO 40
50   READ Y
60   POKE 719 + X,Y
70 NEXT X
80 PRINT : PRINT "INSTALLATION COMPLETE."
90 PRINT : PRINT "CALL 720 TO USE WIPEOUT 1"
100 DATA 160,0,132,214,185,226,2,153
110 DATA 0,2,200,201,130,208,245,76
120 DATA 112,255,184,176,176,186,176,160
130 DATA 206,160,184,176,177,188,184,176
140 DATA 176,174,185,181,198,198,205,130

```

BECOME AN ASSEMBLY LANGUAGE PROGRAMMING WHIZ

"Now That You Know Apple Assembly Language: What Can You Do With It?" will take you step-by-step through the assembly language programming experience. You'll delve into the mysteries of the 6502 stack and learn how to use it to increase the power and versatility of your programs. You'll also learn how to use the Apple's built-in routines to minimize the amount of coding you must do.

Control the output and the input

Frequently it's desirable to gain total control of the computer's output. This book shows you how to *steal control away from the Apple's normal output routines and redirect it to your own pro-*

gram. Thus if you wanted, you could see the normally invisible control characters, display text on your screen as black on white instead of the normal white on black, format text sent to a printer into pages and much more.

Expand the power of your Apple by *stealing control away from the normal input routines.* Do things like adding a screen print capability, or *convert part of the normal keyboard into a numeric keypad.* It's even possible to *produce self-modifying programs* by EXECing in commands from RAM instead of from the disk drive. Think about the possibilities that offers for protecting your programs. When you want to go back to Applesoft programming, *you'll be able to do it faster with the aid of Applesoft Shorthand*, an assembly language program that types in one or more Applesoft commands at the press of a key, or use another program in the book to *automatically count the number of lines in your Applesoft program.*

With this book you'll also learn about *generating tones and how to figure out the frequency, producing sound effects, teaching your Apple to send Morse code, restoring accidentally erased Applesoft programs, adding new commands to Applesoft and running two Applesoft programs in memory together*, to name a few.

As an extra bonus for prompt ordering, you'll receive a **FREE coupon worth \$5 off** the price of a disk with all the assembled programs on it or a disk that contains the source code. These disks normally sell for \$15 each. We're offering these **FREE** gifts for a limited time only, so hurry! **Order today!**

Money-back guarantee*

We're so confident that you'll find this book invaluable and want it in your library, that we're offering a 10-day, no-questions-asked, money-back guarantee. Order the book. Read it and try the programs for ten days. At the end of ten days if you don't think it's worth every penny you paid for it, just send it back in resalable condition and we'll refund your money immediately, no questions asked.

Redllg Systems, Inc., Dept. A 9783
2068—79th St., Brooklyn, NY 11214

Please rush me _____ copies of **"Now That You Know Apple Assembly Language: What Can You Do With It?"** at \$19.95 each plus \$2 shipping and handling. I understand that if I am not delighted with the book I may return it within 10 days for a prompt and courteous refund. In any case, the Programmer's Number Conversion System and \$5 coupon are mine to keep.

☐ Enclosed is my check for \$ _____

Please charge my credit card:

☐ American Express ☐ MasterCard ☐ Visa

Card No. _____ Exp. _____

Signature _____

Name _____

Address _____

City _____ State _____ Zip _____

REVIEW: Locksmith 5.0 Level G

by J. Scott Barrus

First let me preface my review with the comment that I really wanted to review the much publicized *Locksmith 6.0*, but as late as February 17th, Alpha Logic Business Systems was still not shipping the program. They have my order, and money, for over a month and say that it should be shipping any day now, but I've heard that before. With this in mind, I'll therefore go ahead with a review of the latest, currently available version of *Locksmith*.

Locksmith was one of the first anti-copy protection programs out on the market and was originally produced and marketed by Omega MicroWare. The original *Locksmith* could copy programs with almost any protection scheme that was in use when it first came out. Naturally, upon hearing this, software manufacturers rushed to change their schemes so that they would be *Locksmith* resistant. As they changed, so did *Locksmith*. The race was on. Over the years, a total of seven different revisions have come and gone, with the last currently available one being 5.0 level G.

It has an impressive manual

The program comes with an impressive 139 page manual that discusses everything from the history of *Locksmith* and protection schemes, to how to use the program. To supplement the manual, Omega MicroWare decided to come out with a *Locksmith Newsletter*. They came out with one issue. (*Editor's note: Alpha Logic Business Systems decided to pick up on that idea too, but in spite of the fact that they collected money for it, no further issues were ever produced.*) The issue of the newsletter that did come out gives suggestions on how to use some of the harder to understand functions of *Locksmith*.

The *Locksmith* manual is very technical, and most people will find it hard to read and use. Alpha Logic Business Systems has said however, that they are in the process of re-writing the manual. They claim that this will be available to all former *Locksmith* owners, but I suspect that it will really only be pertinent to the next release, *Version 6.0 of Locksmith*, and that it would be of limited value to *Version 5.0* owners. Nevertheless, the company says that it is being written with the average user in mind. This contrasts with the former manual which assumed you were an expert, or at least an advanced user.

When the program is booted up, the user is asked if he wants to load the language card. Upon further investigation of the manual, I found out that this is how *The Inspector* is loaded into memory. The next thing that the user sees is the main menu. There are many options available to the user from this menu. The program always goes to an option and then returns to this main

menu. The utilities that come with the program, as well as the copier, are:

- Backup/copy disk
- Parameter changes
- Text editor
- Quick scan disk
- 16 sector utilities (fast disk verify, backup, format, compare, sync signature)
- Inspector/Watson
- Erase diskette
- Nibble editor
- Disk speed
- Certify disk

One of the important aspects of *Locksmith* is that there is built into it a *Locksmith Programming Language*. Files (called LPL files), produced by the editor for use with this language, allow the user to change different algorithms, parameters and send special instructions to the screen. The manufacturer makes available a file of parameters for a variety of programs, or if you wish, you can key the file in yourself from the newsletter. There is also a default file that comes on the *Locksmith* diskette.

It's not easy to use

The *Locksmith Programming Language* is not easy and it takes some time and effort to study and master it. The language is not needed however, because the user can change parameters and algorithms manually. The problem with the manual approach, however, is that if at a later date you want to use the same changes, you must again enter them manually. Nevertheless, I have found that for occasional copying of a diskette, the manual approach is fine.

Locksmith comes with a text editor that can be used to change the LPL file. There's nothing special about the editor and it can in fact be used to edit any DOS 3.3 text files. Due to the slow loading nature of *Locksmith*, however, I'd really recommend other text editors for non-*Locksmith* applications. After loading in an LPL file with the text editor, you can then copy a disk using this parameter file by selecting the Backup option from the main selection menu.

Many options are available

There are many other options available with *Locksmith* as well. There is a Disk Scan option which lets the user visually (through hi-res graphics) check a diskette to see what the sync byte format is. A Disk Speed option can be used to check, and if necessary adjust, your disk drive speed for best copying results. Another handy tool is the nibble editor. With it, the user can access any nibbles on the diskette to try and figure out what protection scheme is being used. With it, a test of the different algorithms and parameters can be made to see what is neces-

sary to make the diskette readable. It can also be used to find specific byte patterns. This searching capability comes in very handy when you're trying to locate nibble counting code.

It's a powerful copy program

Locksmith is a very powerful copy program. It can copy almost anything. The biggest drawback that I have found with it is that the algorithms, although briefly described in the manual, are still difficult to understand. There are not enough examples provided to make their functions crystal clear. I have been using *Locksmith* for quite a while now, and am still not overly comfortable using it and the manual to make backups. Also, it would be very helpful if the manual had an index in it so that pertinent information could be easily located. Another shortcoming of the program and its documentation is that it is not clear why one would want to use a particular algorithm. What's the selection process? Why should algorithm A be used and not B? The manual never tells us.

Another problem with the program is the everything starts and ends with the main menu, and there is not enough memory in the Apple to hold all the *Locksmith* files in memory. Thus, the diskette is constantly being accessed any time a different utility or change must be made. This is particularly annoying when you try to copy a diskette and fail. You now have to go back to the LPL file and make some changes there. To do this, the diskette you were copying must be removed from the drive, the *Locksmith* diskette must be re-inserted, the editor loaded, the LPL file must be loaded, the changes made, the LPL file must be saved, the copy option selected, the *Locksmith* diskette removed and finally the protected diskette must be re-inserted. If another change has to be made, and that's a real strong possibility, the whole process must be repeated again. It's certainly not a shining example of ease of use.

Literature on the new Version 6.0 indicates that it supports the expanded RAM cards from Applied Engineering and Checkmate. Hopefully *Locksmith* will set up a RAM disk in these cards to speed up operation of the program.

Overall, I think *Locksmith* is a good program, although at present it is not easy to use. It is certainly not recommended for the faint of heart or those who do not want to invest the time and effort to learn how to use it properly. If you're thinking of getting *Locksmith*, don't. Wait until the new version of it is out. If it's half as good as the ads claim it is, it will be a must buy product. **Source:** Alpha Logic Business Systems, 4119 North Union Rd., Woodstock, IL 60098. **Call:** (815) 568-5166.

PROTECTION TUTORIAL — Part II

In the previous article in this series, we talked about some of the early techniques that were used to make it difficult to copy diskettes. Early techniques were simple, but effective. But as Apple users became more sophisticated, more complicated ways of implementing copy protection became necessary.

Before we get into some of these more advanced techniques, it will be helpful for us to understand just how a diskette is formatted and how information is stored on it. While this is normally a subject that is generally meant for technical types, don't be concerned. We are going to assume you have no technical expertise and explain everything from scratch. So sit back and get comfortable, you're about to learn a little bit more about the *black magic* of copy protection.

Formatting a disk

If you have ever placed a new diskette into a disk drive and tried to save a program onto it, you undoubtedly heard your drive make a lot of noise and finally give up, beep and print the message **I/O ERROR**. Go ahead, try it, you won't damage anything. The reason this happens is that diskettes fresh out of the box have absolutely no information stored on them. Such a disk is not divided up into tracks or sectors and thus the Apple doesn't know where to store data, which is why it gives the **I/O ERROR** message.

For the Apple computer to be able to use a diskette, certain information must first be stored on it that can later be used to tell the computer exactly where the magnetic head is located at any particular point in time. This information is put on the diskette when you initialize it by using DOS's **INIT** command. The **INIT** command calls up a machine language program that turns the drive on, moves the magnetic recording head in the disk drive as far away from the center of the diskette as possible (similar to moving the arm and needle of a record player to the very first selection of a record) and finally starts recording magnetic information on the diskette. This information divides the diskette up into 35 tracks that each have 16 sectors in them. More about this later.

The magnetic head in the disk drive is mounted on an arm that moves in and out, from the outer circumference of the diskette towards its center. The mechanics of the disk drive are designed such, that this magnetic head can be moved in discrete steps that are each one half the width of a normal disk track. This is done by a special type of motor called a *stepper motor*.

When a DOS command is issued for the first time, the Apple has no idea where the disk's magnetic head is currently located, so it issues instructions to the stepper motor to move the head back to track zero. If the head were on track three it would only have to move back three tracks, while if it were on

track 20, it would have to move back 20 tracks. Since we said the Apple doesn't know where the head is, it assumes the head is on track 80 (which doesn't exist on standard Apple disk drives) and tells the stepper motor to move the head back 80 tracks towards the outer circumference of the diskette. Since most Apple drives only work with 35 tracks, the head reaches track zero long before it moves back 80 tracks. When this happens, the arm that the head is mounted on bounces against a bumper inside the disk drive and makes that clattering noise that you hear. You can reduce the noise the disk drive makes and speed DOS up a bit by changing this number from 80 to 40. You can do this from Applesoft BASIC by typing in **POKE 48844,40**. This can be done either in the immediate mode (without a line number) or from within a program. Alternatively, if you wish to modify this number from the monitor mode, simply type **BECC:28** and then press **RETURN**. If you do this and then initialize a new diskette, the version of DOS that is put on the newly initialized diskette will have this modification on it.

Getting on the track

As I mentioned earlier, the Apple divides a diskette into 35 individual tracks that are concentric with the large hole in the center of the diskette. These tracks are not connected together in a spiral the way the grooves of a record are, but instead are more like the rings around a bullseye. Each of the 35 tracks is given a number from 0 through 34. Thus, the first track is called track 0 and the 35th (or last) track is called track 34. Track 0 is located the farthest away from the center of the diskette and track 34 is located the closest to the center.

In addition to electronically dividing a diskette into 35 tracks (remember the marks are magnetic ones and invisible to the eye) the computer also divides the diskette into sectors, which can be thought of as the equivalent of slices of a pie. DOS 3.3 divides a diskette into 16 of these wedge-shaped sections called *sectors*. Thus each track has 16 sectors. So with 35 tracks of 16 sectors each, we have a total of 560 sectors on an Apple diskette. Each sector is capable of storing 256 bytes of information, for a total storage capacity of 143,360 bytes of data. On a normal diskette, not all of this space is available to store programs because some of it is used to store DOS and one track (17) is used to hold the diskette's catalog information. Total usable space on a normal DOS 3.3 diskette then, is 126,976 bytes, or 496 sectors.

What a formatted track looks like

You will recall that earlier we said that the Apple couldn't use new diskettes fresh out of the box because they hadn't been initial-

ized and thus didn't have certain information on them. Understanding just what that information is and how it can be changed, is the subject of this next section and is the key to understanding how and why copy-protected diskettes can be produced.

To better understand the discussion that follows, take a look at Fig. 1, which is a representation of a portion of a formatted track. From this drawing, you can see that there are many different components that make up the data storage area of a track. After a quick glance at Fig. 1, you can see that each sector of information that is stored on a diskette contains two areas known as *fields* and two areas known as *gaps*. You'll also notice that the beginning of a track also starts with a gap. We'll get back to this in a little while, but suffice it to say for now, that while Gap 1 originally starts out to be unique from the other two types of gaps, eventually, it gets converted into a Type 3 gap.

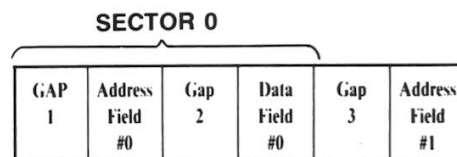


Fig. 1 — Formatted track

If you think you're getting a little lost, just hang in there for another minute or so, and everything will start to become crystal clear. Now back to our explanation. The purpose of the gaps on the track is simply to provide for a separation between the two types of fields and also to allow the computer to have enough time to read the information off the diskette and interpret it. The only difference between the three types of gaps is their length. Some are longer than others.

Now, let's concentrate on those areas in Fig. 1 that are labelled as *fields*. If you look closely, you'll see that there are only two types of fields present on a track: *address fields* and *data fields*.

What's the address?

The first field that we're going to look at is known as the *address field*, and Fig. 2 shows it in more detail. When a diskette is formatted (initialized) address fields are written onto the surface of the diskette for each and every sector. Thus, if the Apple wants to locate a particular track and sector, all it has to do is check the address fields until it finds the one it is looking for. Each address field consists of 14 bytes. The first three bytes are known as the *prologue* or starting bytes and they form a unique sequence of bytes. These specific three bytes will never be found anywhere else on a normal DOS 3.3 diskette except in the address field. This makes it possible to use them as *sign posts* to indicate where an address field starts. The three bytes that have been reserved by DOS 3.3 as address field markers are **D5 AA 96** (these are hexadecimal numbers). Every time DOS sees

a **D5 AA 96** byte sequence, it knows that it has found the start of an address field.

The two bytes that immediately follow the prologue, contain information on the volume number of the diskette. Since this number is never higher than 255, it can be represented by only a single byte of data. You may therefore wonder why two bytes are used in the address field. The reason is simple. The Apple hardware cannot be used to read all 256 possible byte combinations from a diskette. It is therefore necessary to encode the data so that all 256 bytes can somehow be represented. This is done in the address field (except for the prologue and epilogue bytes) by writing out all the odd bits of a byte as one byte (XX) and all the even bits of the

ADDRESS FIELD

Prologue	Volume	Track	Sector	Checksum	Epilogue
D5 AA 96	XX YY	XX YY	XX YY	XX YY	DE AA EB

Fig. 2 — Address field format

same byte, as a second byte (YY). Splitting one 8-bit byte into two separate bytes means there will be four extra bits that have not been defined in each of these new bytes. The Apple makes sure that each of these new bytes starts with a 1 and that each bit from the original byte is separated from the next by a 1 also. Thus the XX byte would look like this:

1 B7 1 B5 1 B3 1 B1

while the YY byte would look like this:

1 B6 1 B4 1 B2 1 B0

Now, if the volume number on our disk was 1, the XX and YY bytes would be:

XX = 1 0 1 0 1 0 1 0 = AA

YY = 1 0 1 0 1 0 1 1 = AB

This method of coding bytes is called by a special name, *4 + 4 encoding*. If you're interested in figuring out what the 4 + 4 encoded values are for any particular number between 0 and 255, just run the *4 + 4 Encoder* program listed below and enter the number of interest. It will tell you what the coded value will be.

Following the two bytes reserved for the volume number in the address field are two bytes which are reserved for the track number and another two bytes which are reserved for the sector number. As was the case with the volume number, the track and sector numbers are also stored in 4 + 4 encoded format.

There are only two more pieces of information left to talk about in the address field: the checksum and the epilogue. The checksum, which again is stored in a 4 + 4 encoded

format, is used by DOS 3.3 to insure that the previous three pieces of information, the volume, track and sector numbers, are correct. DOS calculates the checksum when the diskette is formatted, by performing a mathematical operation known as an exclusive OR on these three pieces of data. It then stores the result on the diskette. This makes it possible to check for data integrity when reading the diskette by reading the first three pieces of information and calculating the checksum. This can then be compared with the checksum stored on the diskette. If they match, all is well. If not, data stored on the diskette is probably corrupted, and thus unreliable.

Just as everything must start somewhere, so too, it must end somewhere, and the three epilogue bytes mark the end of the address field for DOS. The epilogue bytes have the values **DE AA EB**. The address fields are only written once, when the diskette is initialized. Because of that, in spite of a variety of utility programs that claim to do otherwise, it is generally not possible to change the volume number of a diskette after it has been formatted. What the programs that change the volume number do is simply change the number that is displayed when the diskette is cataloged. That does not affect the true volume number of the diskette that is embedded in the address field of every sector.

Unlike the address field, which is only written once (when a diskette is formatted),

DATA FIELD

Prologue	Used Data	Checksum	Epilogue
D5 AA AD	342 BYTES DATA	XX	DE AA EB

6+2 encoded

Fig. 3 — Data field format

the data field is re-written every time information is saved out to the diskette. The data field is similar in many ways to the address field. Like the address field, it features a prologue that uniquely identifies it to DOS as a data field, a checksum to verify that the data is stored accurately and an epilogue to mark the end of the data field (see Fig. 3). In between the prologue bytes and the checksum is an area where data is stored, just as it is in the address field. The difference here, however, is that in the address field the data consisted of the volume, track and sector numbers. In this case, it consists of the user's data. This could be a program, data file, etc.

Encoding the user data

You may recall that earlier we said that since we couldn't use all of the 256 possible byte combinations to store our data on a diskette, that we had to encode it. For the volume, track and sector numbers, Apple chose to use a 4 + 4 encoding scheme. While this is useful, it is also very inefficient, because with that scheme, you'd have to write

```

1000 *****
1010 *
1020 * HEXADECIMAL TO BINARY CONVERTER *
1030 *
1040 *****
1050 *
1060 *
1070 * .OR $0300
1080 *
1090 PRHEX .EQ $FDDB
1100 *
1110 *
1120 * This routine converts a hexadecimal number
1130 * into individual binary bits that are
1140 * temporarily stored in the 8 bytes that begin
1150 * with the location marked TEMP.
1160 *
1170 LDX #$8 Get length of byte
1180 CLC Set carry to 0
1190 LOOP ASL BYTE Shift high bit into Carry
1200 LDA #$0 Load the Carry
1210 ADC #$0 byte into the accumulator
1220 STA BYTE,X and store it
1230 DEX Update bit counter
1240 BNE LOOP If not end get next bit
1250 RTS End, return to caller
1260 BRK Number to be converted
1270 TEMP .HS FFFFFFFF Temporary storage for
1280 * individual bits
1290 *
1300 *
1310 * This routine takes the byte that is stored in
1320 * location NUMBER and prints it out as a
1330 * hexadecimal number.
1340 *
1350 NUMBER BRK
1360 PBYTE LDA NUMBER
1370 JMP PRHEX

```

Table 1
Legal DOS Bytes (Hex)

96	AC	BA	D5	E6	F4
97	AD	BB	D6	E7	F5
9A	AE	BC	D7	E9	F6
9B	AF	BD	D9	EA	F7
9D	B2	BE	DA	EB	F9
9E	B3	BF	DB	EC	FA
9F	B4	CB	DC	ED	FB
A6	B5	CD	DD	EE	FC
A7	B6	CE	DE	EF	FD
AA	B7	CF	DF	F2	FE
AB	B9	D3	E5	F3	FF

out 512 bytes to a diskette in order to save just 256 bytes of data. That would allow for a total diskette storage capacity of only 88K, certainly not very attractive.

The software wizards at Apple however, devised a different encoding scheme that increased the storage capacity by over 50%. Instead of having four data bits per byte as in the 4 + 4 scheme, they decided to use 6 data bits per byte. The remaining 2 bits are split up among other bytes and the whole encoding process is somewhat complex, so we won't go into it here. Suffice it to say, that the new scheme is called 6 + 2 encoding and it converts 256 data bytes into 342 bytes that are written to the diskette, and results in the previously mentioned diskette storage capacity of 143K.

What is important for us to know from this whole discussion is that only certain bytes are used to store data on a diskette and that by changing the way they are used, we can produce copy-protected diskettes. The 6 data bits of the DOS 3.3 encoding scheme can be used to represent 64 different bytes. In addition, two bytes, D5 and AA, are reserved for use in the address and data field prologues only. Thus, only 66 unique bytes are required to store anything on a diskette. A list of the valid bytes that can be used is shown in Table 1.

Making your own protected DOS

Now that we have an idea of how information is stored on a diskette by the Apple, we can go ahead and produce our own protected DOS by making minor changes to DOS 3.3.

To prevent any standard copy program from duplicating diskettes it is only necessary to change any one or more of the prologue, epilogue or checksum bytes. The only thing to be wary of here is changing the last byte of the epilogue on either the address or data fields. A change to this byte will have no affect because, while it is always written to the diskette, its value is never checked.

In another approach to copy protection, the track or sector numbers can be modified. This is what Muse Software did with their products. They modified a version of DOS 3.2 so that it would increment the track number by two instead of one. Thus, while there

were physically 35 tracks on a diskette, they were numbered 0 through 70 in increments of 2.

If you're going to change the prologue bytes, you must be careful. You may recall that earlier we said that two of the bytes in the address field (D5 and AA) were reserved bytes that don't appear anywhere else on the diskette. That is essential to insure that DOS knows how to locate the address and data fields. If you're going to change these bytes, you must make sure that whatever pattern you do decide on is unique.

One common technique that was used in earlier protection schemes was to simply reverse the first two bytes of the prologue. Alternatively, you could substitute a new value for bytes 2 or 3 of the prologue. Just make sure you use one of the bytes that are listed in Table 1.

In order to substitute new values for the standard ones, you have to know where to put them. That's the job of Tables 2 and 3. They list the locations in DOS 3.3 that have critical bytes that can be easily changed to produce a custom, protected DOS. One thing to pay attention to, is that for every parameter that you change, two separate locations must be modified: one for the routine that deals with reading data, and the other for the routine that deals with writing

Table 2
Address Field DOS Locations

Location	Byte	Address	
		Hex	Decimal
Prologue Read	D5	B955	47445
	AA	B95F	47455
	96	B96A	47466
Prologue Write	D5	BC7A	48250
	AA	BC7F	48255
	96	BC84	48260
Epilogue Read	DE	B991	47505
	AA	B99B	47515
Epilogue Write	DE	BCAE	48302
	AA	BCB3	48307

data. If you do one and not the other, you're going to wind up with problems.

To put our new-found knowledge to work, let's produce a protected DOS by swapping the first two bytes of the address field prologue. To do this you can get into the monitor mode by typing **CALL -151** and type in

continued on page 8

```

10 TEXT : HOME
20 A$ = "4 + 4 BYTE ENCODER": GOSUB 420
30 PRINT :A$ = "COPYRIGHT (C) 1986 BY": GOSUB 420
40 A$ = "JULES H. GILDER": GOSUB 420
50 A$ = "ALL RIGHTS RESERVED": GOSUB 420
60 PBYTE = 795
70 FOR X = 1 TO 33
80   READ NUM
90   POKE 767 + X,NUM
100 NEXT X
110 PRINT : PRINT : PRINT : INPUT "ENTER NUMBER TO BE CONVERTED: ";
    NUM
120 IF NUM > 255 THEN PRINT : PRINT CHR$(7);"NO NUMBER LARGER
    THAN 255 MAY BE ENTERED": GOTO 340
130 POKE 785,NUM
140 CALL 768
150 FOR I = 8 TO 2 STEP - 2
160   X(I) = 1
170   X(I - 1) = PEEK (785 + I)
180   Y(I) = 1
190   Y(I - 1) = PEEK (784 + I)
200 NEXT I
210 X = 0
220 Y = 0
230 FOR I = 1 TO 8
240   X = X + 2 ^ (I - 1) * X(I)
250   Y = Y + 2 ^ (I - 1) * Y(I)
260 NEXT I
270 PRINT : PRINT : PRINT "THE 4 + 4 ENCODED VERSION OF ";NUM
280 PRINT : PRINT "IS: ";
290 POKE 794,X
300 CALL PBYTE
310 PRINT "+";
320 POKE 794,Y
330 CALL PBYTE
340 PRINT : PRINT : PRINT : INPUT "CONVERT ANOTHER NUMBER? ";A$
350 IF LEFT$(A$,1) = "Y" OR LEFT$(A$,1) = "y" THEN J10
360 PRINT : PRINT : PRINT
370 DATA 162,8,24,14,17,3,169,0
380 DATA 105,0,157,17,3,202,208,243
390 DATA 96,10,255,255,255,255,255,255
400 DATA 255,255,0,173,26,3,76,218,253
410 END
420 L = LEN (A$)
430 PRINT TAB( (40 - L) / 2);A$
440 RETURN

```


SOFTWARE PROTECTION TECHNIQUES EXPOSED!

Now, for the first time, owners of Apple // series computers can learn all about the tricks and techniques used to protect Apple software. Apple Software Protection Digest, a new monthly publication, will show you how to protect, unprotect and backup your software.



- Prevent others from accessing your programs
- Make your programs difficult to copy
- Overcome protection schemes on commercial software
- Build a library of protection-oriented utility programs
- Get help with your specific problems
- Learn about the latest advances in protection hardware and software

All this and more can be yours by subscribing to the Apple Software Protection Digest. A one-year subscription is \$24, two years is \$42.

SUBSCRIBE TODAY!

APPLE SOFTWARE PROTECTION DIGEST

Vol. 1 No. 1

Premiere Issue

LEARNING TO LIVE WITH PROTECTION

Welcome to the Apple Software Protection Digest. This is the first issue of what will be a monthly publication that is dedicated to the subject of protection and how it relates to software for the Apple // series of computers. If you're like me, you're doubtless seen scores of articles in the various computer publications that tackle the subject of software protection. Most of them take a scolding at or against it and that's the last you hear of the subject. But more is needed. Software protection is a fact of life and we must learn to live with it.

For those purchasers of programs get stuck. They have a program only to discover that it can't be used with a particular accessory board or disk drive. And because the program is protected, it can't be modified. Spelling checkers are a good example of this. They're frequently difficult or impossible to use with non-standard or hard disk drives, even though unprotected programs work without difficulty on these drives.

Apple computer owners need a place where they can get more information about software protection. They need a forum where they can exchange ideas with others who face the same or similar problems. They need to know what software protection is, how it's implemented, what are the consequences of it, how it can be overcome, if necessary, and if there are any conceivable unprotected alternatives to particular protected software packages.

Apple Software Protection Digest will provide you with this information and more. It will show you new ways to protect, unprotect and backup your programs. It will teach you how to prevent others from accessing your programs and it will show you how to make them more difficult to copy. In addition, you'll learn how to overcome these and other protection schemes that are in use. You'll learn how to use the powerful, but complex, and subtle copy programs. You'll also learn how to crack or remove protection entirely from many programs.

With the programs that are included in each issue of the digest, you'll build a virtual library of utility software that will make the job of protecting, unprotecting and backing up software easier. And, as every issue will keep you up to date on both simple and sophisticated protection techniques. In addition to all this, every month you will get reviews of hardware, software and books that are of particular importance to the software protection field.

We do not advocate software piracy. We cannot firmly believe that on the long run piracy will lead to more expensive and low quality programs and less user support. Programmers work long hard hours to get their software working and they deserve to be compensated fairly for it. This cannot happen if software is stolen. In the other hand, the honest consumer should not be penalized and limited in his or her application of a particular program simply because the publisher decided to protect it. You may have a legitimate need to backup a program and we hope to supply you with the knowledge you need to do that. Alternatively, you may wish to protect a program that you've written so that others can't copy it. We'll show you how to do that too.

Apple Software Protection Digest will contain your comments, tips and article contributions. If you have a problem backing up a particular program, let us know. We'll try to help. If you've discovered a way to copy or crack (unprotect) a particular program, let us know about that too. Most likely there are other people who would like to know how to do it. If finally, if you've come up with an ingenious hardware or software oriented protection scheme, write us about it so that we can share it with others. If your article is used, you'll receive a free 12 month subscription to our publication. Let's hear from you soon.

Index H. Goldstein
Editor & Publisher

REDLIG SYSTEMS, INC., Dept. A1357
2068 - 79th St., Brooklyn, NY 11214

Please enter my _____ year subscription to Apple Software Protection Digest.

☐ Enclosed is my check for _____

☐ Please charge my credit card: ☐ VISA ☐ MasterCard ☐ American Express

Card Number _____ Exp. Date _____

Signature _____

Name _____

Address _____

City _____ State _____ Zip _____

ASPD PROGRAM DISKETTE AVAILABLE FOR ONLY \$15

Starting with this issue, we will make a DOS 3.3 diskette available every month that contains all of the programs from the current issue of the *Apple Software Protection Digest*. Disk 1, which is available now, also contains the programs from the premiere issue of the Digest.

To order send a check, money order or your charge 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

