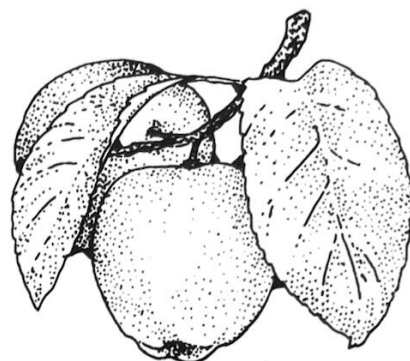


● APPLE SOFTWARE PROTECTION DIGEST

\$3.00



Vol. 1 No. 3

1986



SOME PUBLISHERS DROPPING PROTECTION

During the past few months we have seen an interesting trend starting to develop among software publishers. Many of them, under increasing pressure from corporate users, are starting to drop software copy protection from their products. The latest publisher to join the pack is Software Publishing Corp., which produces the PFS line of programs.

According to Software Publishing, they've finally realized that copy protection is a big inconvenience for the customer and they don't have to worry about protecting themselves so much any more because their business customers are protecting them and preventing employees from making copies.

I'm glad to see a movement away from copy protection. In fact, I think the whole issue of how much money is lost due to the production of illegal copies has been blown way out of proportion. There really hasn't been any hard research to determine just how much money is lost due to piracy. There also hasn't been any research done to determine how much money is lost due to the problems and inconvenience caused by copy protection, and I think that would be a real eye-opener.

Be that as it may, don't run out to buy PFS software yet. According to a Software Publishing spokesman, the protection is not going to be removed from existing products, but rather as new versions of the various programs become available.

In spite of the reasons given by Software Publishing for this move I tend to think the facts are a little different. I suspect that enough people have gotten fed up with copy protected software that

they've just decided not to buy any. This is where we, the people who keep these companies afloat, can really show our strength. There is plenty of good software available that is not copy protected.

Support unprotected shareware

Much of the unprotected software that is available is being put out on a "shareware" basis. What this means is the developer makes the diskette available to anyone who wants it for a nominal fee, usually between \$10 and \$20. He tells the purchaser that he is free to make copies of the diskette and distribute it to anyone he chooses provided he doesn't charge more than the developer does, doesn't modify the diskette in any way and if he's purchased documentation separately he must agree not to duplicate it and give it away or sell it.

These shareware programs generally come with a sufficient amount of documentation on the diskette so that the user can learn to run the program. The shareware developer's hope is that the user will like the program so much, he'll support the developer by paying an additional fee to register his copy, get more complete printed documentation and usually one or two future updates for free. Sometimes even the source code is made available.

If you think these developers are foolish for taking this approach, think again. Several shareware developers have made over \$1 million so far and their programs are as good as, or better than many commercial products costing hundreds of dollars more.

Contents

Editorial	1
Cracks Wanted	2
Crack Index	2
Letters	2
How to Crack a Program ...	3
Parameter Files for COPYP .	6
Add Undeletable Lines to Your Program	7
Moving the Catalog to Another Track	9
REVIEW: The quickLoader ROM Card	10

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.

Cracks Wanted

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

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

1. Sensible Speller — DOS Version
2. Sensible Speller — ProDOS Version
3. Crush, Crumble & Chomp
4. Wizardry
5. Educational Software from Compress
6. Dazzle Draw
7. Newsroom
8. Word Handler
9. Disk-O-Check

Crack Index

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

- Bookends — Vol. 1, No. 1, p. 7
 Homeword — Vol. 1, No. 2, p. 9
 Homeword Speller — Vol. 1, No. 2, p. 9
 PFS Series — Vol. 1, No. 2, p. 7
 Print Shop — Vol. 1, No. 1, p. 10
 Print Shop Companion — Vol. 1, No. 2, p. 6
 Sensible Grammar — Vol. 1, No. 2, p. 7
 Time Is Money — Vol. 1, No. 2, p. 9

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

Letters

Dear Editor:

I liked your courtesy copy of **ASPD** and I'm going to try it for a year. I'm also ordering your diskette. I do have a problem however.

I am a writer and like *Applewriter II*, but it has two limitations which center around its copy protection. Oh, it's easy enough to make a backup copy of it (I use Locksmith 4.1). The nature of its copy protection however, makes it impossible to either put it onto a hard disk, or use it with one. I can't get the computer to recognize the slot after I've booted AW II.

My problem can be solved in one of two ways, as I see it. I need a way to modify a copy program, like Locksmith 4.1 or a way to copy Apple Writer II that leaves it unprotected. Then I could put it onto my hard disk easily. Can you tell me how to do this?

I know this much, after my experiences over the last few months, I won't buy a copy-protected program again. They are just too inconvenient to use.

Lee Baldwin
Concord, CA

The quick answer to your first question is yes, we can help you. This issue contains a full-length article devoted to unprotecting Apple Writer. If you don't want to read the whole thing, but just want the parameters for the COPYP program, they can be found in the COPYP Parameter List. The reason Applewriter doesn't recognize your hard disk is that the modified DOS that is needed to recognize your hard disk is replaced by the protected DOS of Apple Writer. If your hard disk is a Sider and you have a way of breaking out of the program into the monitor mode, you may be able to patch DOS so that it can use your hard drive. The patching information can be found in the Sider User's Guide on page 133.

Dear Editor:

I just read about your Apple Software Protection Digest. If it is as excellent as it has been reported to be, I will gladly pay the annual subscription fee. Please send me a sample issue.

By the way, I bought a game called *Crush, Crumble and Chomp* and haven't been able to make a backup copy. If you know of any way of doing so, please give me full details.

Michael Jacobs
Wolcothville, IN

Here's your sample copy Mike. I hope we've lived up to our advance billing. I'm adding your request to our Cracks Wanted list. If any of our readers can tell Mike how to back up Crush, Crumble and Chomp, let's here from you.

Dear Editor:

I was happy to see that my programs (Two Solutions to the Hide-A-Line Problem) had won your contest and were published in **ASPD #2**. I must confess though, that I have encountered a problem with the concept of hiding a line by resetting the link bytes.

The problem stems from the fact that any line hidden in this manner will cause an UNDEF'D STATEMENT error if the program does a GOTO or GOSUB to it. Has anyone else run into this dilemma?

Because of this apparent glitch, I much prefer the method of adding five colons at the start of the program line. The GOTO/GOSUB problem does not occur when using this technique. I've had lots of fun using this technique and baffling "unaware" people with it.

Thanks for the extra six months, I look forward to future issues of **ASPD**. You present ideas and concepts that many of us can experiment with and learn from, something that is missing from many Apple periodicals nowadays.

Mark Landwehr

Thanks for your letter Mark. The problem you encountered with the GOTOs and GOSUBs in the first line hiding technique is to be expected. When Applesoft encounters a GOTO or GOSUB, the 6502 starts scanning the BASIC program from the beginning (via the next line pointers) and looks at the two bytes that follow each next line pointer to see if the line that is desired has been found. Since we've changed these pointers so that they bypass the line when the program is listed, the line will also be bypassed when GOTOs and GOSUBs are encountered.

HOW TO CRACK A PROGRAM

Many people have written to me and told me that they appreciate being given step-by-step instructions on how to backup or crack a particular program. They also indicated however, that they would like to know how one goes about starting to crack a program and develop their own technique for backing it up. That's really a tough question to answer because there are almost an unlimited number of ways to go about cracking a program. To get you started however, I'll take you step-by-step through the thought processes involved in cracking Applewriter //e (DOS version). The reason this program was chosen was because one of our readers sent in a letter asking how he could remove the protection from Applewriter so that he could use it with his hard disk.

If you're not interested in knowing how Applewriter //e is unprotected, but just interested in unprotecting it, you can skip the rest of this article and simply use the COPYP parameter file that is listed elsewhere in this issue to automatically make an unprotected backup.

Know it well before you start

Before you attempt to crack any program, it is essential that you use it and become thoroughly familiar with it. If you know how the program will act in various situations, you will have an enormous advantage working for you when you try to figure out how the program is protected.

If you had become familiar with Applewriter //e before you attempted to crack it, you would have discovered that the Applewriter disk uses a fairly normal form of DOS. In fact, the diskette can be CATALOGed like a normal DOS diskette and you can use COPYA to make a copy of the diskette. While no error is generated when the copy is made, don't rejoice too quickly, because the copy will not work. Nevertheless, since the modification to standard DOS seems to be small, chances are pretty good that it won't be too difficult to crack the protection. This is generally true for most protected programs. The closer they are to standard DOS, the easier they are to crack.

Since we know that DOS 3.3 is more or less intact, we can find out what the boot program is. It's important to understand how a program boots up in order to crack its protection. By using

a sector editor, such as the one available on the Copy II Plus diskette, we can look at track \$1, sector \$9, byte \$75 (remember the \$ indicates the number is a hexadecimal number), which is where we'll find the name of the program that is automatically run when the diskette is booted. In the case of Applewriter //e, the file is called OBJ.BOOT. It is a binary file and it is automatically BRUN when the diskette is booted.

The job of OBJ.BOOT is simple. It tests to see if the computer that is being used is a Apple //e. If it's a // Plus, it will run the program OBJ.APWRT][D which simply prints a message to the user telling him that an Apple //e is required to run the program. If the computer being used is a //e, OBJ.BOOT then checks to see how much memory is available. If only 64K is available, then the file OBJ.APWRT][E is run. On the otherhand, if 128K or more of RAM is available, OBJ.APWRT][F will be run.

Prepare all your tools

Before you attempt to crack a program, it is essential that you have some basic tools. To begin with, you need a sector editor that will let you read, modify and write diskette sectors. The second thing you'll need is a diskette searching program. This is a program that will let you enter a string of characters or hex values to be searched for on a diskette. Both of these capabilities are available in the Copy II Plus program (which normally retails for \$39.95 but costs only \$30 when you buy it through us). Next, you should have a short, in-memory search program that can be used to locate a sequence of bytes in RAM. For our Applewriter //e crack, we'll use a short machine language search routine that Bob Sander-Cederlof gives away with his S-C Macro Assembler. It's only 53 bytes long and can easily be keyed in when needed. The last thing you're going to need is a way of breaking out of the program once it's running. I have found that the Wildcard 2 board is excellent for this purpose. It allows you to press a button at any time and then gives you a menu that let's you do several things, including jumping to the monitor. It plugs into any slot in the computer and will work on any Apple or compatible except the Apple //c

(which has no slots).

STEP 1: Copy it if you can

When trying to crack a protected program, always try to copy it first with COPYA. If you can't get anywhere at all, another approach will have to be taken. But if you can copy it, as is the case with Applewriter //e, you've gone a long way towards your ultimate goal. Once you've made a copy of the diskette, try to boot it up. Pay close attention to what happens during the boot process and compare it to what happens during a normal boot of the original diskette.

In the case of Applewriter //e, when the copy is booted, OBJ.BOOT is run and if an Apple //e is being used, depending on how much RAM is available, either OBJ.APWRT][E or OBJ.APWRT][F is run after it. Whichever file is activated, if the diskette is a copy, the program will detect this, zero out all of memory and then jump to Applesoft BASIC, most likely via the 'cold start' entry point at \$E000. It is this jump to BASIC that turns out to be the Achilles Heel of Applewriter //e.

STEP 2: Boot the program

After you've made a backup copy of Applewriter //e, put it aside and boot the original diskette. Once the program is loaded and you get to the opening title screen, get into the monitor. I do this by pressing the button on my Wildcard 2 and selecting the jump to monitor option from the menu that is presented.

Once you're in the monitor, the whole world of Applewriter is open to you. You can examine the program code to your heart's content, but you won't be able to modify it and save it out to the diskette because you don't have access to a normal DOS. Nevertheless, there's a lot you can do to unprotect the program from this mode.

STEP 3: Search the program

Once you're in the monitor, you can search for Applewriter's Achilles Heel, the jump to BASIC which occurs when the program determines that an original diskette is not being used. You can of course search the computer's memory manually and examine every location yourself. A much easier way to do it however, is to let the computer do the

hard work. If you examine page 3 of memory (addresses starting at \$300) you'll find that most of it has been zeroed out. Thus, it is a handy spot for us to put in a short machine language program that will do the searching for us.

Since we don't have access to the disk drives from the monitor mode, we can't load the program in from the disk. The only other choice is to load it in from tape, for those people who still have this capability in their computers (I think it was eliminated with the enhanced Apple //e ROMs) or key it in directly. Since the program I use is short (only 53 bytes long) I key it in whenever I need it. Eventually, I'll probably put it in an EPROM that can be switched in whenever the program is needed, but that's another project.

As I mentioned earlier, this search program is one of several sample programs that come on the S-C Macro Assembler diskette from S-C Software. The program itself is easy to use and it allows you to specify the starting and ending locations of the range to be searched, the byte sequence to be searched for and it allows you to designate any particular character as a wildcard character. You can examine the full assembly language source code listing to see how it operates, or simply key in the seven lines listed below.

```
300:A9 03 8D FA 03 A9 10 8D
308:F9 03 A9 4C 8D F8 03 60
310:A2 00 A0 00 B5 02 C5 01
318:F0 04 D1 3C D0 11 C8 E8
320:E4 00 D0 F0 A5 3C 85 3A
328:A5 3D 85 3B 20 D0 F8 20
330:BA FC 90 DC 60
```

Once the search program has been keyed in, it must be initialized by typing **300G** from the monitor mode. Now you're ready to do your searching. Since we know that the jump to BASIC is associated with the protection scheme, if we find that, we can probably work our way back towards the beginning of the protection scheme code. Thus, it's important for us to locate this jump to BASIC. The machine code required to jump to BASIC should contain a three-byte sequence that looks like this:

4C 00 E0

so we'll search through memory for this sequence of bytes. We do this by placing a 3 in memory location 0 on the zero page of memory to indicate how many

bytes are in the sequence of characters we're looking for. Next, we place the value of the wildcard character in location 1. Since we have no need of a wildcard character in this particular search, I just place an FF in this location. Finally, we have to enter the sequence of bytes that are being searched for starting with location 2 on zero page. In summary, the following line must be entered (from the monitor mode) to set up the search parameters:

0:03 FF 4C 00 E0

All that's left to do now is specify the range of memory locations to be searched and activate the program. Both of these tasks are done together by typing in the starting and ending addresses, separated by a period, entering a Control-Y and then pressing the RETURN key. Since RAM ends at \$BFFF I decided to search all of memory from \$800 to \$BFFF. To do this the following line was entered while in the monitor mode:

800.BFFF <Control-Y>

Notice that I've placed spaces after the BFFF and the Control-Y. This is for ease of reading only and these spaces should not be used when you enter this line.

STEP 4: Find where it begins

When you search memory for this string of characters, you see that it appears only once, at location \$2D0C. This is indicated by the search program which prints out the following line in response to your search:

2D0C- 4C 00 E0 JMP \$E000

Since that's the only place in the program where the jump to BASIC occurs, chances are pretty good that starting at this location and working our way backwards, we should be able to find the protection code. As you work your way back, look for another JMP instruction or an RTS instruction. The first one you encounter on your journey back in memory will usually mark the end of some previous routine and the beginning of the routine you're examining. In the case of Applewriter //e, I encountered a JMP \$0200 at \$2CDD. At the time I didn't pay much attention to it, although I should have because it is very unusual

to jump to a subroutine that is located in the input buffer, unless of course you're trying to hide something. My failure to pay attention to this caused one false start, but that was quickly corrected. I'll get back to that in a little while.

Since the JMP instruction takes up three bytes, that means that the subroutine I was examining probably started at \$2CD0. The listing of the code that resides between \$2CD0 and \$2D0C, where control is passed to BASIC is shown below. This listing was made by using the Apple's built-in disassembler and I have added line numbers to this listing so that it will be easy to reference a particular line.

```
100 2CD0- AD 83 CO LDA $C083
110 2CD3- AD 83 CO LDA $C083
120 2CD6- A9 03 LDA #$03
130 2CD8- 85 01 STA $01
140 2CDA- A0 00 LDY #$00
150 2CDC- 84 00 STY $00
160 2CDE- 98 TYA
170 2CDF- 91 00 STA ($00),Y
180 2CE1- C8 INY
190 2CE2- D0 FB BNE $2CDF
200 2CE4- E6 01 INC $01
210 2CE6- F0 0C BEQ $2CF4
220 2CE8- A6 01 LDX $01
230 2CEA- E0 C0 CPX #$C0
240 2CEC- D0 F1 BNE $2CDF
250 2CEE- A2 D0 LDX #$D0
260 2CF0- 86 01 STX $01
270 2CF2- D0 EB BNE $2CDF
280 2CF4- AD 82 CO LDA $C082
290 2CF7- AD 82 CO LDA $C082
300 2CFA- 8D 0C CO STA $C00C
310 2CFD- 20 84 FE JSR $FE84
320 2D00- 20 2F FB JSR $FB20
330 2D03- 20 93 FE JSR $FE93
340 2D06- 20 89 FE JSR $FE89
350 2D09- 20 58 FC JSR $FC58
360 2D0C- 4C 00 E0 JMP $E000
```

STEP 5: Examine the code

Now that we know where at least some of the protection code lives, let's take a close look at it to see what it does. Because Applewriter //e wipes out all of memory before jumping to BASIC when a copied diskette is encountered, we would expect this routine to perform that task, and it does. Here is a detailed explanation of exactly what goes on when this routine is called.

In lines 100 and 110, the RAM card in the computer is write enabled, so that data can be stored in it. Lines 120 to 150 set up a pointer on page zero that will be used to indicate which memory locations are to be zeroed out. Here the program starts with location \$300. Line 160 loads a zero into the accumulator and line 170 is the line that actually zeroes out the current memory location that is being pointed to. The Y-register was in-

initially zero and is incremented in line 180. This allows the instruction in line 170 to point to every location on a particular page of memory. As long as the Y-register has not returned to zero (been incremented 256 times) the program loops back to line 170 and keeps storing zeroes in successive memory locations.

Once the Y-register does become zero again, line 200 increments the page pointer so that the next page of memory is set up to be zeroed out. As long as location \$01 does not contain a zero (and it won't until the very last byte of available memory has been addressed) control passes from line 200 to line 220 where the contents of location \$01 are loaded into the X-register. Line 230 checks this value to see if it is equal to \$C0. If it isn't, control is passed once again to line 170. In this manner, all of memory from \$300 to \$BFFF (which is one less than \$C000 which was just checked for) is zeroed out.

Because Applewriter //e also uses the RAM card, the program then goes on to wipe out all memory locations there as well, which is why the RAM card was write-enabled earlier. The RAM card's memory starts at \$D000 and so line 250 loads a \$D0 into the page counter at location \$01 on page zero. Control is once again passed to line 170 and the program loops once more to zero out all successive memory locations. This time however, after a zero is stored in location \$FFFF on the RAM card, the page counter is incremented again and thus returns to zero. This triggers the instruction in line 210, which causes the computer to jump to line 280 where the RAM card is turned off.

Since Applewriter //e uses the 80 column mode when it is active, the next instruction (line 300) turns off the 80 column card and activates the 40 column mode. Line 310 makes sure the Apple is set up for normal (not inverse or flashing) video while line 320 makes sure the text screen and not either of the graphic screens, is activated. Line 330 does a PR#0 to make sure the output hooks at \$36 and \$37 are returned to their normal condition. Similarly, line 340 does an IN#0 to restore the input hooks (\$38 and \$39). Finally, line 350 clears the screen and line 360 jumps to BASIC.

STEP 6: Find it on the disk

Now that we've located the protection routine and understand how it operates,

we have to find out where its located on the diskette, so that it can be modified. To do this, we're going to have to use a disk scanning program which, like the short machine language program we entered earlier, will let us search for a particular byte sequence. Several such programs are available commercially. I use the one that is on the Copy][Plus diskette. You get to it by selecting the Sector Editor option and then pressing the S key. The program will then ask you if you want to enter your search string as hex codes or text. Typing an H selects the hex code mode.

Since we know from the disassembly listing that starts at \$2CD0 what we're looking for the job is not too difficult. I decided to search for the first eight bytes of the routine that starts at \$2CD0 — AD 83 C0 AD 83 C0 A9 03. It seemed to me that this sequence of bytes would be fairly unique, and would probably only be encountered in the protection code. After entering this data, I found that the code was stored on track 3, sector 9 and started at byte 1 (which is the second byte in the sector because we start with 0). I continued searching the diskette to see if the code would crop up somewhere else, and sure enough it did, at track 6, sector B, byte D4. The fact that the code was found in two places on the diskette suggested that the code was located in the OBJ.APWRT][E and OBJ.APWRT][F files.

STEP 7: Disable the protection

Once you find where the offending code is located on the diskette, all you have to do is disable the protection. In this case, that can easily be done by storing a \$60 (the RTS code) at the very beginning of the protection scheme code. To do this you'll need a sector editor that will let you read, modify and then write a diskette sector. Always remember, NEVER WRITE ON THE ORIGINAL. Do all of your work on a copy only. Once again, I have found that Copy][Plus is the tool to handle the job. Using the copy that I made, but wouldn't boot, I stored a \$60 at track 3, sector 9, byte 1 and track 6, sector B, byte D4.

STEP 8: Test it out

With patches applied to the non-working copy, it is now time to test it out. Booting up this diskette, I was delighted to see that the boot proceed-

ed as normal and was rewarded with the normal Applewriter opening screen. But my joy was to be short-lived. After going through a short sample session, I attempted to quit Applewriter and lo and behold, I couldn't get out of it. It seems, that as part of the procedure for quitting the program, the memory wipeout routine that I just disabled, gets called into action. Back to the drawing board.

STEP 9: Find why it failed

With a lot of luck, you'll be able to skip this step, but such was not to be the case with Applewriter. At this point, I remembered the strange JMP \$0200 instruction that I saw immediately before the memory wipeout routine and proceeded to examine it and the code before it more carefully. I discovered that immediately preceding the protection routine was another routine that transferred the original wipeout code into the input buffer where it was then executed by the JMP \$0200 instruction. The short routine that does the moving starts at \$2CBF. This apparently was the real protection routine.

Once again, I typed in the memory search program and looked for the place where this code was called from using the byte sequence: 4C BF 2C. It turns out that this sequence of bytes is located at \$3B04 (when OBJ.APWRT][F is loaded). Examining the code that precedes this JMP instruction reveals the routine that is used to check and see if the disk that is in the drive is an original. This code starts at \$3AF1 when OBJ.APWRT][F is loaded and is listed below.

Without going into a lot of detail, this routine calls another one that immediately follows it (at \$3B08) and reads the disk. This routine checks for the correct prologue bytes in the address field of the sectors (for more information see Protection Tutorial - Part II, ASPD Vol. 1, No. 2, p. 20). It then checks for special information that is only present on the original Applewriter //e diskette. If it doesn't find that information, it jumps to the memory wipeout routine at \$2CBF.

By the way, it's important to note that any protection scheme that accesses special information on a diskette, be it special sync bytes, extra data or a nibble counting routine, must access the diskette with an instruction such as LDA \$C08C,X where X contains the slot number that the disk drive is connected

```

3AF1- 20 08 3B JSR $3B08
3AF4- 85 82 STA $82
3AF6- 20 08 3B JSR $3B08
3AF9- C5 82 CMP $82
3AFB- D0 0A BNE $3B07
3AFD- 20 08 3B JSR $3B08
3B00- C5 82 CMP $82
3B02- D0 03 BNE $3B07
3B04- 4C BF 2C JMP $2CBF
3B07- 60 RTS
3B08- AE E1 02 LDX $02E1
3B0B- BD 8C C0 LDA $C0BC,X
3B0E- 10 FB BPL $3B0B
3B10- C9 D5 CMP #$D5
3B12- D0 F4 BNE $3B08
3B14- EA NOP
3B15- BD 8C C0 LDA $C0BC,X
3B18- 10 FB BPL $3B15
3B1A- C9 AA CMP #$AA
3B1C- D0 F2 BNE $3B10
3B1E- EA NOP
3B1F- BD 8C C0 LDA $C0BC,X
3B22- 10 FB BPL $3B1F
3B24- C9 96 CMP #$96
3B26- D0 E8 BNE $3B10
3B28- EA NOP
3B29- EA NOP
3B2A- BD 8C C0 LDA $C0BC,X
3B2D- 10 FB BPL $3B2A
3B2F- 2A ROL
3B30- 85 80 STA $80
3B32- BD 8C C0 LDA $C0BC,X
3B35- 10 FB BPL $3B32
3B37- 25 80 AND $80
3B39- 60 RTS

```

to. Searching a diskette for the bytes that represent this instruction — BD 8C C0 — will usually get you to the protection code, eventually. The problem is that any other routine that has a legitimate need to access the disk drive will also use similar code, so you'll have to examine a lot of code before you get to what you want. Therefore, always try to get close to your protection routine by

other means first, just as we did here.

Now that we know what the true protection scheme looks like, let's use our disk scanner again and try and locate on the diskette. Remember, we're working with the copy of the original that we made only! Using the search pattern C5 82 D0 03 4C, I located three possible candidates. Finding three patterns disturbed me a bit because I had expected only two, as in the previous case. The three patterns were located on track 4, sector C, byte B1; track 7, sector C, byte 8 and track 7, sector E, byte 1. Since I had one more bit pattern than I anticipated I examined each carefully and found that while the last pattern started out the same as the other two, it was significantly different after the first several bytes and most important, it did not have the disk access instructions. This routine was therefore discarded and only the first two were used.

To eliminate the protection, once again the first byte of these two routines was replaced with a \$60, the machine language code for the RTS instruction (ReTurn from Subroutine). Thus, a \$60 was placed on track 4, sector C, byte B1 and track 7, sector C, byte 8. After testing the diskette this time, the program worked perfectly. The interesting thing about this whole thing is that all it took was a two byte change on the copy of the Applewriter //e diskette to make it

a working copy. The only reason it was necessary to change two bytes was because there were two different versions of the program on the diskette. Otherwise, only a single byte change would have been necessary. Isn't the power of information astounding?

If you don't have all of the tools that I used to crack this program, I suggest you get them right away, if you anticipate cracking more software. In the meantime, if you just want to make an unprotected copy of Applewriter //e, you can use the COPYP program that appeared in the last issue and the Applewriter //e parameters that are listed in the COPYP Parameters section of the digest.

I've spent a lot of time explaining the step-by-step process that I went through to crack this program so that you could understand the thought process that went on. This will help you to crack other programs, but don't expect any miracles. There are many different ways in which programs can be protected and this has only been one of them. Hopefully, however, you've gained some insight that can be applied to other programs. If you do crack any other programs, let us know how you did it so that we can share the procedure with the rest of our readers. We'll reward you with an extension of your subscription. Let's hear from you soon.

PARAMETER FILES FOR COPYP

Listed below are several parameter files for use with the COPYP program that was presented in **ASPD** Vol. 1, No. 2. You can key these lines in directly or you can do what I do and create text files that contain these lines. Then you can load in COPYP and EXEC in the appropriate file for the program you want to copy. You can keep these text files on a diskette and re-use them again whenever you want to make another copy.

Some of these files have been tested by me and some haven't. It gets to be an expensive proposition to buy each of these programs in order to test out each file. Therefore, if you submit files to us, please make sure you test them thoroughly. Also, please give us the version number of the program you're cracking. Nothing is more discouraging

than trying to use a routine that is supposed to work, only to find out that it doesn't. Your reward for doing this will be a one-month addition to your subscription.

Applewriter //e

A complete explanation of how *Applewriter //e* is unprotected is included in a separate article in this issue. For those of you who are merely interested in making an unprotected copy of it and not interested in the how and the why, just add the following lines to your COPYP program and make as many unprotected copies as you want.

```

1000 DATA 4,12,177,76,96
1010 DATA 7,12,8,76,96

```

Financial Cookbook

Financial Cookbook from Electronic Arts is a program that can be easily unprotected by changing just a single byte in the program. The reader who supplied the information for unprotecting this program and the next couple of programs neglected to tell us what the value of the original byte that is being changed was.

Since we have been unable to contact him so far, we decided to add a slight modification to COPYP that would enable you to unprotect a program without having to know what the original value of the byte that's being changed is. This of course bypasses the built-in safeguard that doesn't let you modify a different version of the program (which is determined by checking to see that the original value of the byte is what it should be) but it can come in handy sometimes.

continued on page 11

ADD UNDELETABLE LINES TO YOUR PROGRAM

For those of you who write programs in Applesoft BASIC this next program will be of some interest to you. Did you ever see some commercial programs that had a copyright notice and author credit listed at the bottom of the program using line numbers that somehow could not be deleted? Have you ever wondered how you could do the same thing? It's easy and this program will help you to do it.

Most Applesoft BASIC programmers eventually discover that the highest line number that can be entered in an Applesoft program, from the keyboard, is 63999. That next to the last phrase "...from the keyboard..." is the key here however. The designers of Applesoft, for some reason, decided to make it illegal to have line numbers that were greater than 63999. Therefore, everytime that a line is entered from the keyboard, the computer checks to see if the line number is greater than 63999. If it is, the computer gives you a SYNTAX ERROR message. Since the checking is done as the line is entered, if we can find some way to bypass the input routine, we can have larger line numbers in our program and everything will work fine. As an added bonus, once we have this larger numbered line in our program, there will be no convenient way to delete it.

To make life easy, what I generally do is write out the lines that are to become permanent additions to my program as a separate file and then I run this short program, while that file is in memory. I then save the resulting program, which is usually just REMs with my name and a copyright notice in them, out to disk. When I finish developing an Applesoft program, I then merge the newly developed program with my short credit file program and I now have a program that contains the proper credit to me in lines that are difficult (but not impossible) to delete.

This program, *All Line Numbers to 65535* does just what its name implies, it changes all line numbers in any particular program to 65535. For those of you who may be wondering just how its possible to have a program with more than one identical line number, I'll explain in a minute. Suffice it to say that you can't have it if the lines are being entered from the keyboard, because the moment you enter a second line with a number that is identical to an already existing line, the old line with the same

```

1000 *****
1010 ***
1020 ***      ALL LINE NUMBERS TO 65535 ***
1030 ***
1040 ***      Copyright (C) 1986 by ***
1050 ***      Jules H. Gilder ***
1060 ***      All Rights Reserved ***
1070 ***
1080 *****
1090 *
1100 *
1110 * Equates
1120 *
0006- 1130 TXTPTR      .EQ $06
0008- 1140 POINTER    .EQ $08
0067- 1150 TXTTAB     .EQ $67
00AF- 1160 PRGEND     .EQ $AF
FC58- 1170 HOME       .EQ $FC58
FD0D- 1180 COUT       .EQ $FD0D
1190 *
1200 *
1210 *
0300- 20 58 FC 1220      JSR HOME          Clear the screen.
0303- A9 49 1230      LDA #TEXT          Point to the text
0305- A0 03 1240      LDY /TEXT          that is to be printed.
0307- 20 38 03 1250      JSR MSGPRT      Print it.
030A- A5 67 1260      LDA TXTTAB        Get the start of
030C- A4 68 1270      LDY TXTTAB+1      program pointer
030E- C5 AF 1280      ENDCHK CMP PRGEND  Compare with the
0310- 90 05 1290      BCC NEXT          end of program
0312- C4 80 1300      CPY PRGEND+1      pointer to see if
0314- 90 01 1310      BCC NEXT          we're done yet.
0316- 60 1320      RTS                Yes, we're done.
1330 *
1340 * This section of the program saves the
1350 * pointer to the current line and then
1360 * replaces the line number in that line
1370 * with the number stored in LINNBR, in
1380 * this case $FFFF which is 65535. Next,
1390 * the pointer is updated to point to the
1400 * next line. The program then jumps to
1410 * a routine that checks to see if we've
1420 * reached the end of the program.
1430 *
0317- 85 08 1440      NEXT STA POINTER    Save the line
0319- 84 09 1450      STY POINTER+1      pointer.
031B- A2 00 1460      LDX #0           Initialize the
031D- A0 02 1470      LDY #2           offset counters.
031F- BD A2 03 1480      LDA LINNBR,X   Get the low byte
0322- 91 08 1490      STA (POINTER),Y  of the new number
0324- E8 1500      INX                and replace the
0325- C8 1510      INY                old one.
0326- BD A2 03 1520      LDA LINNBR,X   Get the new high
0329- 91 08 1530      STA (POINTER),Y  and replace old.
032B- A0 00 1540      LDY #0
032D- B1 08 1550      LDA (POINTER),Y  Update POINTER to
032F- 48 1560      PHA                point to the next
0330- C8 1570      INY                line.
0331- B1 08 1580      LDA (POINTER),Y
0333- A8 1590      TAY
0334- 68 1600      PLA
0335- 4C 0E 03 1610      JMP ENDCHK
1620 *
1630 * This is the message printing routine.
1640 *
0338- 85 08 1650      MSGPRT STA TXTPTR  Store the pointer
033A- 84 07 1660      STY TXTPTR+1      to the message.
033C- A0 00 1670      LDY #0           Initialize offset.
033E- B1 08 1680      LDA (TXTPTR),Y   Get next character.
0340- F0 08 1690      BEQ ENDPRT      Done yet?
0342- 20 ED FD 1700      JSR COUT      No, print it.
0345- C8 1710      INY                Point to next character.
0346- D0 F8 1720      BNE LOOP        Go get it.
0348- 60 1730      ENDPRT RTS          Return to caller.
1740 *
0349- C1 CC CC
034C- A0 CC C9
034F- CE C5 A0
0352- CE D5 CD
0355- C2 C5 D2
0358- D3 A0 D4
035B- CF A0 B6
035E- B5 B5 B3
0361- B5
0362- 8D 8D 1760      TEXT .AS -"ALL LINE NUMBERS TO 65535"
0364- C2 D9 A0 1760      .HS 8D8D

```


number gets erased and is replaced by the new one.

Multiple lines with the same number are possible however. To accomplish this bit of micro magic, we must first type in lines with ordinary line numbers. Since we know what the structure of a BASIC line is as it is stored in memory (see **ASPD**, Vol. 1, No. 1, p. 2) we can go into the monitor mode and change the line number bytes manually so that they contain any line numbers that we wish.

To simplify and automate the process, all you have to do is enter this short program and it will do the job in a fraction of a second.

Make it hard to modify

While my primary use for this program is to produce undeletable credit lines for Applesoft programs that I write, it is also possible to use this program to make it difficult for someone else to modify your programs. If you give all your lines in a program the same high number, it will be hard to change any line in the program. A word of caution is needed here however. If entire programs are going to have identical line numbers in them, then you won't be able to use GOTOS, GOSUBs and THENs followed by a line number, because there will be no way of telling the computer to go to a specific line.

How it works

Operation of the program is fairly straight-forward. After clearing the screen and printing out the title page (lines 1220-1250) the program gets the start of program pointer (lines 1260-1270) so it knows where to begin and then falls into a routine to check if the end of the program has been reached yet (lines 1280-1320). The check is made by comparing the current location in memory, which is stored in the accumulator and the Y-register with the end of program pointer located on Zero page at AF and B0.

If the end of the Applesoft program has not been reached, the machine-language program branches to line 1440 where the current memory pointers are saved. In lines 1460 and 1470 the X and Y registers, which are used as offset counters, are setup so that they can be used to retrieve the two bytes of the new line number and store it in place of the old line number. This actual replace-

```

0367- CA D5 CC
036A- C5 D3 A0
036D- C8 AE A0
0370- C7 C9 CC
0373- C4 C5 D2 1770      .AS -"by Jules H. Gilder"
0376- 8D          1780      .HS 8D
0377- C3 CF D0
037A- D9 D2 C9
037D- C7 C8 D4
0380- A0 A8 C3
0383- A9 A0 B1
0386- B9 B8 B6 1810      .AS -"COPYRIGHT (C) 1986"
0389- 8D          1820      .HS 8D
038A- C1 CC CC
038D- A0 D2 C9
0390- C7 C8 D4
0393- D3 A0 D2
0396- C5 D3 C5
0399- D2 D6 C5
039C- C4          1830      .AS -"ALL RIGHTS RESERVED"
039D- 8D 8D 8D
03A0- 8D 00        1840      .HS 8D8D8D8D00
                1850 *
                1860 * Number to which all lines are changed.
                1870 *
03A2- FF FF        1880 LINNBR .HS FFFF

```

```

1  REM BASIC PROGRAM TO INSTALL ALL LINES TO 65535
2  REM
10 TEXT : HOME
20 PRINT : PRINT : PRINT : PRINT
30 PRINT "INSTALLING 'ALL LINES TO 65535'..."
40 FOR X = 768 TO 931
50   READ Y
60   POKE X,Y
70 NEXT X
80 PRINT : PRINT : PRINT : PRINT "INSTALLATION COMPLETE."
90 PRINT : PRINT "TYPE 'CALL 768' TO RUN PROGRAM."
100 DATA 32,88,252,169,73,160,3,32
110 DATA 56,3,165,103,164,104,197,175
120 DATA 144,5,196,176,144,1,96,133
130 DATA 8,132,9,162,0,160,2,189
140 DATA 162,3,145,8,232,200,189,162
150 DATA 3,145,8,160,0,177,8,72
160 DATA 200,177,8,168,104,76,14,3
170 DATA 133,6,132,7,160,0,177,6
180 DATA 240,6,32,237,253,200,208,246
190 DATA 96,193,204,204,160,204,201,206
200 DATA 197,160,208,213,205,194,197,210
210 DATA 211,160,212,207,160,182,181,181
220 DATA 179,181,141,141,194,217,160,202
230 DATA 213,204,197,211,160,200,174,160
240 DATA 199,201,204,196,197,210,141,195
250 DATA 207,208,217,210,201,199,200,212
260 DATA 160,168,195,169,160,177,185,184
270 DATA 182,141,193,204,204,160,210,201
280 DATA 199,200,212,211,160,210,197,211
290 DATA 197,210,214,197,196,141,141,141
300 DATA 141,0,255,255

```

ment is done in lines 1480 to 1530 and the pointer that tells the program where the next line to be worked on is located is updated in lines 1540 to 1600. Lines 1650 to 1730 comprise the routine that is used to print text out to the screen and lines 1770 to 1840 contain the text itself. Finally, line 1880 contains the line number that is going to be used to replace all of the existing ones. It is stored low byte first and in this case is \$FFFF, which represents the number 65535.

To make entering the program easy, I have included a short Applesoft BASIC program that will automatically load the machine language program for you. After the program is entered, you can execute it by typing **CALL 768**. If you do that while the BASIC loading program is in memory, all its line numbers will be changed to 65535. If you then try re-running the program, you see that it executes without a problem because it contains no GOTOS or GOSUBs.

MOVING THE CATALOG TO ANOTHER TRACK

One very effective way to protect programs from being copied is to change the track that the catalog is stored on. Normally it's stored on track 17 (\$11) from sectors 15 (\$F) to 1. While changing the catalog track will stop the standard copy programs (COPYA, FID, MUFFIN) it, of course, will not stop the bit copier programs from duplicating a diskette that uses this technique. However, if you combine this technique with some of the other copy protection schemes we've discussed so far, or if you move the catalog to track 37 or higher, even the popular nibble copier programs won't be able to duplicate your diskette.

While most of track 17 is devoted to the catalog, sector 0 of this track has a special job. It contains what is known as the **VTOC** (for **V**olume **T**able **O**f **C**ontents) and it keeps track of which sectors have been used and which are available for future storage. In addition to keeping track of which sectors are available, the presence of this special sector makes it possible for you to lock out any track and/or sector on the diskette and make sure that it won't be written to. This is helpful if you want to "bury" a serial number on the diskette somewhere. We'll get back to a more detailed discussion of the VTOC later.

Moving the VTOC

There are several different approaches that can be used to move the contents of track 17. You can choose to move just the VTOC, just the catalog data or both. If you move just the VTOC, you'll still be able to run programs under normal DOS, you just won't be able to save anything out to the diskette without the risk of damaging it. Nevertheless let's start by moving the VTOC. Moving the VTOC is perhaps a misconception because we're not going to take an initialized diskette and then move its VTOC, although it is possible. Instead, we're going to modify DOS and then use this modified DOS to initialize a new diskette. The modification that we make to DOS will cause the new diskette to be initialized with the VTOC on another track. The modification that we have to make to DOS to move the VTOC is a simple one and consists of changing only a single byte of memory. That byte is located at **44033 (\$AC01)**, and it is the location that contains the number of the track on which the VTOC is located.

This location is used both by the routine that initializes a diskette and by the routines that load or save programs to a diskette. Under normal circumstances the number stored in **44033** is **17 (\$11)**.

We can change the VTOC track by simply **POKE**ing a new number into location **44033**. For example, from **BASIC** we can type in the following line in the immediate mode (no line number) and cause the VTOC to be saved out to track 37 where it cannot be copied by standard copy programs.

POKE 44033,37

As you will recall, we discussed how to add up to 5 tracks to your diskettes in the first issue of **ASPD** (Vol. 1, No. 1, p. 8). Since none of the commercial bit copier programs copies more than 36 tracks, storing the VTOC and catalog onto one of these extra tracks is an effective copy protection scheme.

You can move the catalog too

To make your files inaccessible to normal DOS, you can move the catalog sectors to track 37 as well. This would make it almost impossible to access the programs on a copied version of the diskette, especially if the space on track 17 is released and used to store programs on. To change the number of the track that DOS goes to, to look for the catalog information, it is only necessary to change two locations in DOS. They are **46012 (\$B3BC)** and **44764 (\$AEDC)**. The first location is used to tell DOS which track to write the directory onto, and the second location tells DOS which track to format as a directory track. Both of these locations must contain the same track number, which is normally 17. Once again, by changing the number in these two locations, you can produce a diskette that has its catalog information on track 37, which cannot be copied by most copy programs that are available. The following BASIC line executed in the immediate mode will make the changes for you:

POKE 46012,37 : POKE 44764,37

There is a big advantage to doing this. By putting the catalog and VTOC on a track that cannot be copied, you can tell users of your protected diskette that they can make backup copies of the diskette with **COPYA**, which will not run, and

save them in case the original becomes damaged. Since odds are small that the damage will occur on the 37th track, most of the time, users will be able to restore their damaged original by recopying their backup copy onto the original. Of course, if the damage does occur on the 37th track, the diskette will have to be replaced.

Tell the VTOC about the changes

Once you've modified DOS so that it will produce a 37-track diskette and moved your VTOC and catalog onto it, you must also tell the VTOC that track 37 is used and that track 17 is available. You do this by changing two locations in DOS which contain information that tells DOS where the catalog track is located. These locations are **44741 (\$AEC5)** and **44745 (\$AEC9)**. The first location (**44741**) contains the number of the track on which the directory is located, multiplied by four, while the second location (**44745**) contains four times the directory track number plus four. Thus, for standard DOS, **44741** contains **68 (\$44)** while **44745** contains **72 (\$48)**. Interestingly enough, if we use **68** as an offset into track 17, sector 0 (the VTOC sector) we get exactly to the point where we have four bytes that are used to mark the availability of track 17. This is not just a coincidence, because these two locations are used by DOS to create the VTOC bit map.

Thus, after we make the necessary changes to move our directory (or catalog) onto track 37, we have to make these changes too. They can easily be implemented by typing in the following line in the immediate mode:

POKE 44741,68 : POKE 44745,72

Now you have completed the modifications necessary to move the VTOC and the catalog to a different track. All that's left for you to do is initialize your new diskette. Don't forget, if you're using more than 35 tracks on your diskette, you'll have to modify DOS as described in **ASPD** Vol. 1, No. 1. While I've concentrated on showing you how to move the catalog and VTOC to an extra, non-standard track, there is obviously no reason why it cannot be moved to any other track on the diskette. The reason I've concentrated on these extra tracks is, as I mentioned earlier, they cannot be easily copied with the current crop of copy programs.

To make the whole task of producing

custom diskettes easier, I have included a short BASIC program that will take care of all the nitty gritty details for you. All you have to do is answer the questions it asks about the number of tracks you want on the diskette and where you want the catalog track located. The program does the rest.

A challenge and a gift

For those of you who have the time and the inclination, here's a small challenge for you. Use the information I have just presented to you to come up with a program that will produce a diskette that has two catalog tracks on it. You should provide a means of switching back and forth between the standard catalog and the hidden one so that programs can be saved on one or the other. Provision should also be made to update both VTOCs whenever either one is modified so that nothing will be accidentally overwritten.

The simplest way to update the

VTOCs is to first make sure that the track with the hidden catalog on it is locked out as an in-use track on that normal VTOC. You can do this by storing zeroes in the appropriate bit map locations on track 17, sector 0 (remember the offset to the starting location of the bit map can be calculated by multiplying the track number by four. Thus the bit map for track 37 would start at byte 148 (\$94) in the VTOC sector. To lock out an entire track, four successive bytes must be set to zero. After the standard VTOC has been modified to protect the hidden VTOC, just read the standard VTOC sector into memory, modify byte number 1 (the second byte in the sector) so that it is equal to the track that the hidden catalog is stored on and write it out again to the VTOC sector of the hidden catalog track. Now, if you always copy the the VTOC sector of the catalog that has just been modified (the currently active one) to the VTOC sector that was not modified (the inactive one), remembering to adjust that second byte

to the correct track number, you'll always have both VTOCs properly updated and not have to worry about accidentally overwriting something on your diskette.

For your efforts in developing this program, the winner will get fame and recognition by having it published in **ASPD**, the satisfaction of having developed your own protection scheme and a *free 6-month extension to your subscription*. The second best entry will get a *free 3-month extension*. You may use any programs that have already appeared in **ASPD** as a starting point if you think they'll be helpful, so let's see you get those entries in.

If you need more information on just how the catalog track is formatted you can look in Apple's DOS manual or in one of the handiest books I've found on the subject, *Beneath Apple DOS*. If you have difficulty find this book, which is published by Quality Software, you can order it through us at a 10% discount, for only \$17.95.

REVIEW: The quikLoader ROM Card

One of the most useful accessories that you can purchase to help you crack programs is the quikLoader ROM Card from the Southern California Research Group (SCRG). Designed to be used in an Apple II, Apple II Plus or Apple IIe, the quikLoader card can hold up to 256 programs in ROM with a total memory capacity of almost half a megabyte. Programs can be written in machine language or either Integer or Applesoft BASIC.

The main reason the quikLoader card is such a helpful tool in cracking programs is that it can be used to interrupt a running program and let's you select from a menu, any other program you'd like to have run. When the card is enabled, one of the eight possible ROM chips that are installed on it, is selected for response to addresses in the \$C100 to \$FFFF memory range. At the same time, motherboard response in this range is inhibited. This is great for the cracker because he can now reset directly into the monitor or run any other program without having to worry about overcoming protection schemes in programs that make it impossible to break out of the program by pressing RESET. When you have a quikLoader in your computer, anytime the RESET or

Control-RESET key is pressed, the card is enabled and ROM chip 0 is selected. The result is that when RESET is pressed, a program in chip 0 is run. This program is called QLOS (for quikLoader Operating System).

The action that QLOS takes when a reset is encountered depends on which key was pressed just before or, at the same time as the RESET key and the status of the power-up byte at \$3F4. By pressing the appropriate key concurrently with Control-RESET, you can choose from a variety of resets. Included in this list are a normal reset, forced power-up reset, forced disk boot, a catalog of the quikLoader ROM(s), execution of specific programs from the ROM(s). Below is a list of the actions allowed when a reset is encountered, and the keys that must be pressed to implement them.

Z - Move Integer BASIC, the monitor ROM and DOS into RAM, initialize DOS and enter Applesoft BASIC.
n - (number 0-7) Run a program on chip n.
Q - quikLoader catalog (referred to as katalog to differentiate from a disk catalog).
H - Do a Z-Reset and then run the HELLO program.

B - Boot DOS by moving DOS to RAM, initializing it and entering Applesoft.
D - Disk boot.
C - Catalog a diskette.
M - Enter the monitor mode.
S - Soft reset (slot 0, 16K RAM card reset).
X - Go to the mini assembler.

In addition to these reset options, there is an additional one call A-reset. This is encountered when an undefined key is pressed. In an A-reset, the computer jumps to the address contained in \$FFFC and \$FFFD on the motherboard (the main Apple circuit board) if the power-up byte is good. If the power-up byte is not good, the computer jumps to the power-up routine on chip six of the quikLoader.

If you replace the power-up routine that is stored in the chip 0 ROM with one of your own programs, you can have the quikLoader card do anything you want when the RESET (or Control-RESET) key is pressed.

Uses very little RAM

The design philosophy behind the quikLoader is to transfer programs that are stored on EPROMs into RAM for execution. However, programs can be run while they reside in the ROM card

(that's how QLOS is run). When this is done, very little user RAM is needed. The memory that it does need is primarily used for pointers, counters and temporary storage. Zero page locations are used for most temporary storage and certain routines that must run in RAM are loaded either into the input buffer at \$200 or the bottom of the 6502 stack at \$100. These locations were chosen by the board's designer Jim Sather (author of *Understanding Your Apple*) so as to minimize any likely interference with user programs. And his goal was achieved.

Using the board's B-RESET feature, you can load DOS into the Apple and very few memory locations will be changed. There's no multi-stage boot process involved that wipes out large chunks of memory. DOS is loaded directly from the card into the final location in memory where it will be run. Thus, it is a simple matter to boot up a protected diskette, load the program into memory, boot normal DOS from the quikLoader card, and then save the program out to an unprotected diskette.

It comes with software

The quikLoader card has eight sockets on it for EPROMs. When you buy the card, you'll get it with some EPROMs already in it. These will contain DOS 3.3, Integer BASIC, FID and COPYA. These programs are licensed

from Apple Computer. SCRG also has other programs available in ROM for use with this board. These include the Beagle Bros. Double-Take utility and Central Point's Copy][Plus. And, if you send them your configured version of AppleWorks on a diskette, they'll burn it into ROM for you so that you can be up and running with it in less than 2 seconds.

The board is extremely versatile and can accommodate any combination of EPROMs from the 2K 2716s to the 64K 27512s. If completely populated with 64K EPROMs, the quikLoader can store almost a half megabyte of data. Since the card can be plugged into any slot except slot 0 (but including slot 3 in the Apple //e) it is possible to store over 3 MB of data in EPROMs and have any of it instantly available.

Support is superb

For those of you who may not have heard of SCRG, it's a small company that sells a number of hardware accessories for the Apple. But don't hold its size against it, because you'll be hard pressed to find another company, even a Fortune 500 one, that supports its products as well as SCRG does. While my quikLoader card has always worked well and I've never encountered any problems, the company sent around a notice that a change had been made to the card and the ROMs to enable it to

work with the 27512 EPROMs (I bought mine before these chips were available). And though SCRG was not obligated to, it offered to update my card for free if I sent it back to them. I did, and they turned it around very quickly. In the years that I have owned this card, I have had several occasions to call the company and ask technical questions about how or why something was done, or if the card could be used for specific applications. I always found someone technically knowledgeable to talk to, usually Phil Wershba himself (he's the president). In those rare circumstances when he couldn't answer my questions, the designer, Jim Sather could.

SCRG offers a 6 month warranty on everything it sells, but I have found that if you encounter a problem even after the warranty expires, that the company will take care of it for you anyway (if possible), at no charge. Another nice practice of this company is its 10-day free trial period. You can order any of their products and try them for 10 days. If for some reason you're not happy with a product, all you have to do is return it for a prompt refund. If you're going to be doing a lot of program cracking, or if you just want the added convenience of having your most frequently used programs in ROM ready for instant access, then this card is a must for you. **Price:** \$179. **Source:** Southern California Research Group, P.O. Box 593, Moorpark, CA 93020.

COPYP Files continued from page 6

The checking routine is bypassed by adding line 485, which is listed below and setting the old value of the byte to 0.5. Actually, any value between 0 and 1 will do, but 0.5 is easy to remember. Thus the parameter file for *Financial Cookbook* becomes:

```
485 IF OV > 0 AND OV < 1 THEN
500
1000 DATA 1,6,8,0.5,98
```

COPYP will now make a backup copy of the program and, without checking to make sure this is the correct version of the program, modify the diskette so that it will work.

Hayes Terminal Program

Like the previous program, this one was submitted without the value of the

byte that is to be changed. Since we do not have a copy of this program, we can't check it and find the value of the original byte.

```
485 IF OV > 0 AND OV < 1 THEN
500
1000 DATA 16,3,157,0.5,37.
```

Microwave

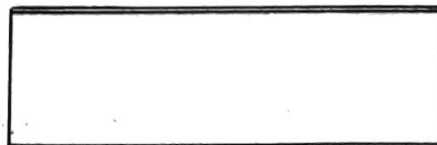
Although this program is a few years old, it has proven to be a very popular one and for that reason, cracking information is included here.

```
485 IF OV > 0 AND OV < 1 THEN
500
1000 DATA 2,1,218,0.5,173
1010 DATA 2,1,219,0.5,3
1020 DATA 2,1,220,0.5,129
1030 DATA 2,1,221,0.5,96
```

That's all it takes to unprotect Microwave.

Give us a hand

In order for this publication to be really useful to you, you have to get involved. Tell us what you'd like to see and also contribute your favorite cracks to us. If you took the time out to unprotect a program, chances are that someone else is interested in it too. Remember, you get a one-month subscription extension for every program crack you submit and we use. So let's see those helpful hints.



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*.

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

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