Deprotection Methods and Techniques Using the Senior PROM

January 1986

Cutting Edge Enterprises
Box 43234 Ren Cen Station
Detroit, MI 48243
313-349-2954 Modem 300/1200 Baud

0730Z

Deprotecting Methods and Techniques
Using the Senior PROM

Table of Contents

Chapter	Description	Page
1	Introduction to Deprotection with the Senior PROM	2
2	Where to Begin	4
3	Single Load Protection	8
4	Modified DOS Protection	17
5	Modified RWTS Protection	22
6	Disabling Minor Disk Access	26
7	Examining Protected Applesoft Basic Programs	28
8	Using COPYB to Convert Protected Programs	32

Cutting Edge Enterprises
Box 43234 Ren Cen Station
Detroit, MI 48243
313-349-2954 Modem 300/1200 Baud for Information and Support.

The Senior PROM is especially designed for allowing you to backup, view, or modify protected software. It is not designed for ANY illegal uses. We provide our hardware, software and documentation for archival and educational uses only!

As with any tool, you must possess the knowledge to use it before it can work for you! Read the Senior PROM instruction manual throughly and also read the other documentation suggested by this chapter. If you skip these two steps, the rest of this manual will not be as useful...

The thrust of this manual will be discussing the methods used in deprotecting copy protected Apple programs. After doing the suggested groundwork and reading (and understanding!) the following information, you should be able to deprotect most any commercially protected Apple program. Of course, this manual will not make you an instant expert on Apple copy protection, and there are many things to learn that I can not cover. This will be an excellent starting point to grow from. The key to your success will be keeping an open mind, understanding all concepts mentioned, ingenuity, and above all, practice.

Deprotecting protected programs requires several things that you should try your best to possess. The first is a good basic understanding of the Apple Computer and its architecture. Read your "Apple II Reference Manual (//e only)" for very informative discussions of your //e computer. Also a very good manual is the "DOS User's Manual" from Apple. Even better, pick up a copy of Don Worth's "Beneath Apple DOS" (and "Beneath Apple ProDOS" for some applications) and study it carefully. After you have done these basic things, you are ready to start down the road of deprotection.

I can not stress the importance of reading these well written manuals, and doing the best to understand their implications. After you have achieved the knowledge granted by these books, you are well beyond 95% of the Apple users.

Of particular importance is the "Apple II Reference Manual (//e only)". The most important part of this manual is chapter 5, "Using the Monitor" (page 87). *Study this chapter carefully.* This chapter will describe many of the monitor commands you will need to snoop protected programs. These commands include examining and changing memory locations, moving memory and comparing ranges of memory, and executing code. Also discussed is the Mini-Assembler and how to enter your own machine language programs.

For the ultimate understanding of the Apple the last step is to learn the forbidden language, 6502 Assembly language. Although this is not necessary for a great number of deprotecting chores, any program that is the least bit tricky will require you to understand Assembler. The reason is simple: most everything sold today is

written in Assembler. Ninety-five percent of the programs at your local computer store are written in Assembly language. The obvious reason for this is that Assembler is fast, and this is *very* important for many business applications and graphic games. Also, because protecting a disk on the Apple is done at the Operating System level, the protection really has to be written in Assembler.

For learning Assembler, I would suggest either Roger Wagner's "Assembly Lines", or Randy Hyde's "Using 6502 Assembly Language". Both of these books are excellent and are easy to understand for the beginning programmer.

Beyond this, the next best thing to do is to use your new found knowledge. Write some Assembly language programs to get familiar with the language. Instead of writing that "hello" program in BASIC, do it in Assembler. Get used to it, and keep good notes of what you learn!

Now you're ready for the big time.... deprotecting programs. I assume you have a good understanding of the monitor commands (list, move, verify, execute) from reading the Apple II reference manual. Of most importance is the "L" command to disassemble and list code presently in memory. Get familar looking at these disassemblies since you will never have "source" code from protected programs to examine. Therefore you will have to become fluent in Apple disassembly. The best way to achieve this is practice, and nothing else will substitute.

Good luck, and I am sure you find deprotection extremely educational and helpful.

Chapter 2: Where to Begin. A discussion on the first steps to deprotecting a program.

Before we really start digging into discussions on copy protection, I think we should first define a few things and state the objective of this manual.

The objective is to make the average user more informed about copy protection and how to defeat it using the Senior PROM. Copy protection is what a publisher/author does to prevent you, the user, from making "unauthorized copies".

The problem with copy protection is obvious: what do you do if:

- Your original disk fails?
- You need to modify the protected program for your particular application?

Copy protection makes copies with "FID" or "COPYA" impossible, and many users save turned to programs such as "Locksmith" to make back-ups. This is not defeating the copy protection, but merely makes a clone of the entire disk and its protection. The disk is still un-modifiable, and consistent copies are a problem and time consuming. Many people consider this unacceptable.

I will be discussing "deprotection" methods using the Senior PROM, as opposed to copying methods using Locksmith. Deprotection is defined as the techniques used to capture a protected program onto a normal DOS 3.3 disk that can be copied in a convenient manner, such as with COPYA or FID.

The primary objective of this manual is to provide you with the knowledge to deprotect programs using the Senior PROM. Of course, using this information for illegal and otherwise illicit uses is not the intent of this series, and is written only for your amusement and education. Now that the objectives are clear, lets begin...

The subject of deprotecting programs is truly a humongous one. There are probably thousands of ways to make a disk "uncopyable". This provides us with a mental puzzle that can (and probably will) consume much of your time and effort. But of course, the rewards are worth it not only in the obvious returns, but also in the gain in knowledge of programming and your Apple in general. If you enjoy puzzles and treasure hunts, you will enjoy deprotecting programs.

Part of any puzzle is to find the first piece or starting point. This is no exception in the puzzle of protection either, and is probably the hardest step for the novice. There is no substitute for experience, but there are some general guidelines to follow and some real dead giveaway clues to look for.

Protection generally falls into one of three categories. The first is protecting a single program in memory and/or on disk (called the "single load" protection).

The second is protecting a set of programs by using a modified DOS (called the "modified DOS" protection). The third is using a loader or a modified RWTS (read-write-track-sector) for protection (called "Modified RWTS protection"). Since there are so many ways to protect programs in the above three manners, I will be mentioning general techniques that you may apply to just about any protected program.

First, you must be able to recognize which category a particular program fits into. The remainder of this chapter will discuss this. After determining the type of protection used, you should refer to the particular chapters on deprotecting that protection type.

Recognizing the Single Load Protection:

This type of program is probably the easiest type of protection to deal with, but unfortunately is being seen less and less every day. Single load programs are loaded in from disk only once, and then run from memory with no or very little disk access. Several years ago just about all games on the Apple fell into this category, and deportecting these programs was not that difficult.

The identification of these programs is simple: When the program boots and loads into memory and starts running, there is no additional disk access. (Disk access is when a program turns the disk drive on and read some data from disk. Usually this is for loading additional program segments or game levels.)

Programs that first load a title page, turn off the drive, and then wait for you to press a key and load the program can also be single load if no other disk access is encountered. An example of this is Penguin Software's arcade games. Even though the drive turns off and then on after showing a title page, this is a single load program (title pages just don't count!).

In addition, programs that save high scores to disk or do only some other minimal disk access are also single load. Many times they are not actually loading any data but are checking to see if the original disk in still in the drive or are writing a high score to disk. An example of this is most Penguin arcade games and many of the Electronic Arts games (One on One, Axis Assassin, etc.). This small disk accesses can almost always be defeated in some manner.

Usually the final step in deprotecting a single load programs is converting the program into a single (or small number of) file(s). From normal DOS the resulting file can then be run. Most of you have seen this type of program.

Recognizing the Modified DOS Protection:

Probably the most popular protection scheme is the Modified DOS Protection. These programs load much like a disk that you create (with the command INIT HELLO). The difference is that the publisher/author have modified DOS slightly to read and write to in their particular copy protection scheme.

Publishers like to use this protection scheme because it is easy to incorporate, easy (and cheap) to make multiple copies for retail sales, and does a good job at discouraging the Locksmith owners from making copies.

Fortunately, deprotecting modified DOS disks are generally more systematic and sure-fire than the other types of protection. Also, there are many programs already developed to aide you in deprotecting modified DOS programs.

There are some real dead giveaways to identifying whether a protected program is using a modified DOS. The foremost is the appearance of a BASIC prompt on the screen during the boot (the "]" prompt). Some protectors have started to bypass the routine that prints the prompt, but you can still guess there is a modified DOS present from the sound of the boot.

The sound of the boot is very important. Initialize a normal DOS 3.3 disk and boot it a couple of times. Listen to your disk drive as the disk boots. You will first here the drive chatter (this is making sure that the drive is ready to read track zero of the booted disk). Right after the chatter, the disk drive head will swing out to track two and read to track zero (most drives click each time the head swings to another track. Listen for the click.). This is the process of loading DOS into memory. This process takes about 3 seconds at the most. Next you will here the drive head swing out to the catalog track to locate the "hello" program and load and run it.

If upon booting a protected disk you hear the above sounds, there is a 99 percent chance the program uses a modified DOS. If a BASIC prompt appears, it is a 100 percent chance a modified DOS is present. I have devoted a chapter to this type of protection.

Recently, ProDOS is being used in protection. This is really a derivative of the modified DOS protection and is covered in that chapter.

Recognizing the Modified RWTS Protection:

The use of a modified RWTS is a simular to the Modified DOS protection scheme. RWTS is a portion of DOS that does the actual reading and writing of particular sectors of a disk. The main difference between a modified DOS and a modified RWTS is that a DOS interacts with the disk on a file basis. A RWTS interacts with the disk on a track and sector basis, and therefore actual files (as grouped in a directory or catalog) may not exist.

Modified RWTS (or "loaders" if they only read and do not write to disk) are popular for multilevel games that must read from disk. Since RWTS occupies a much smaller portion of memory than DOS, the actual program that is being protected can be larger than if an entire DOS is being used. The reason is DOS must manage files and then do track and sector reads using the file lists. RWTS only worries about tracks and sectors doesn't care if they are grouped into files.

(NOTE: many single load programs use a modified RWTS to do the initial loaing of the program. If there is no more disk access after the initial load, the program falls into the "Single Load Protection" category, not "Modified RWTS Protection. If additional disk access is encountered, it is considered a Modified RWTS Protection.).

The method of deprotecting these disks is quite similar to deprotecting modified DOS disks. This is because RWTS is merely a portion of DOS.

To identify a disk as using a modified RWTS is fairly simple. If the boot does not sound like a modified DOS boot (as described in the Modified DOS protection above) and a BASIC prompt does not appear, and there is additional disk access during the program, the protection may be using a modified RWTS. Basically, if the program does not fit into the other two categories, it is probably a modified RWTS protection. (Some other alternatives are a modified PASCAL operating system, like the PFS and Wizardry series use, or a ProDOS-based protection scheme).

Examples of programs using this kind of protection are programs like DB Master, all the Designware programs, Zaxxon, Miner 2049'er, Donkey Kong, and Jungle Hunt. I have devoted a chapter to this type of protection.

Conclusion:

Now that you can identify what type of protection is being used, refer to the chapter that discusses it. This will outline the methods used to deprotect that particular type of program.

Chapter 3: Deprotecting Single Load programs.

Single load protection encompasses those programs that are loaded in only once, and then run strictly from memory with no disk access. Some minimal disk access is allowed, for example, to save high scores or to check for the original disk, but ultimately, these will be defeated or altered to allow us to save our program into a normal DOS binary file.

To deal with this type of protection you will have to have some knowledge of ASSEMBLY language and some good working knowledge of the Apple's monitor commands. In addition, the ability to decipher the monitor's "disassembly" will prove invaluable. These tools will help a great deal since by the nature of the material, the rules are written in deceiving and uncommented 6502 machine language. There is no substitute for experience, but the only way to gain experience is to practice, so let's begin.

First let's outline the steps we want to follow in deprotecting a single load program:

1) Find the starting address. This is the address that will always restart the program.

2) Figure out what parts of memory should be saved (we can not save >>all<< of memory, so we must figure out what is really needed and what is not).

3) Save the program as a normal DOS binary file (B file) that includes all of the needed memory.

Step 1: Finding the Starting Address:

To further explain what a starting address is, remember when you are using "FID" from your DOS 3.3 System Master and you mistakenly reset from the program? Well many people get their system master disk back out and BRUN FID again to run the program. This is really unnecessary since all one must do is to enter the monitor and type the starting address with a "G" at the end (the "G" is the monitor's go or execute command):

```
]CALL -151
*803G
```

The starting address of $803 will always restart FID, and this is what we want to find in our protected program.

(Now some of you may be asking how I knew $803 was FID's starting address. A normal DOS file's starting address is kept at $AA72 and $AA73, in backassward order of course. After BRUNing or BLOADing FID, type:

```
*AA60.AA73
```

The first two bytes listed are the length of the file, and the last two bytes are the starting address. Remember they are listed backwards, so $0803 will be listed as 03 08.)

So boot a protected program, activate your Senior PROM and press CTRL - RESET, and then the DELETE key (giving us the "*" prompt). Now we can test our starting address by typing the address we think it is followed by a "G" to start execution at that address.

In the old days the starting address was even numbers like $800, $900, or $6000. It is definitely still worth checking these address as many people find it more convenient to program with these starting addresses.

Usually, you will be disappointed by your first attempts at guessing the starting address. Therefore, we need a more structured method for finding the programs's starting address (after all, there are more than 64,000 locations to choose from!).

Since many programs first display a hi-res title page before starting the program, a good place to start looking is the series of instructions that turn on the graphic pages. The graphic pages are turned on and off by a series of "soft switches" in the $C050 range. It doesn't matter what you do to these locations as long as you access them in some way:

C050:	Display graphic mode
C051:	Display text mode
C052:	Display all text or graphics
C053:	Mix text with graphics
C054:	Display primary page (page 1)
C055:	Display secondary page (page 2)
C056:	Display lo-res graphics mode
C057:	Display hi-res graphics mode

This means that the following commands will have the same effect of turning on the graphics mode:

```
LDA  $C050     STA  $C050     EOR  $C050
BIT  $C050     CMP  $C050     ROL  $C050
```

and if you understand the indexing from ASSEMBLY language:

```
LDY  #$71
AND  $BFAF,Y
```

However, most reasonable people have established the chore by writing:

```
LDA  $C057
LDA  $C054
LDA  $C052
```

to turn on the graphics page.

Now to find these instruction you can page through memory with the monitor's "L" command or you can use the Sector Editor's Find command and search for "50 C0" (refer to the Senior PROM documentation for how to use the Sector Editor's Find command).

After you find the code, trace backwards looking at the code just before it. Try and find an absolute end for the previous code before such as an RTS or a JMP. Your starting address should be immediately after the absolute end of the previous code.

In addition, the code that turns on the hi-res page may just be a small subroutine, so you may have to search for a JSR to that location and trace backwards to find the starting location. Once again, the Sector Editor will help you in this task.

When you think you have found the starting address, test it with the "G" command (i.e. "9000G").

If you fail, reload the program and start over. It is a good idea to always reload the program since the code you executed might have disturbed some other memory locations. It is always best to start fresh.

Also keep in mind that that the hi-res routine may have been accessed by a "branch" instruction. These are conditional jumps that can reach $7F locations away in either direction. So search about 60 instruction before and after your the possible start. If you fine a BEQ, BNE, BPL, BCC, or BCS to your starting address, trace that routine back to its beginning and try it.

In addition, try looking for a JMP to your starting location with the Senior PROM's Sector Editor. This may produce another routine to trace back and find the starting address.

Keep in mind if you have to trace back more than two steps, you are probably in the wrong area of memory and on the wrong trail.

In addition, many programs wait for a key press before starting the program. You may also search for the code that accesses the keyboard. Usually the code looks like this:

```
LDA   $C010      Clears the keyboard
LDA   $C000      Get a keystroke
BPL   $XXXX      If no key, goto $XXXX
JMP   $START     keystroke found, jump to start
```

To find this code use the Sector Editor to search for the sequence "00 C0".

This is very common code and often produces a starting address. Another very good way of finding a starting address is to find the key that re-starts the program. Many times games use CTRL R to end the current game and to start a new one. This almost always produces a good starting address. Usually the code looks like this:

```
LDA   $C000      Check for keystroke
BPL   $XXXX      No key, JMP to next stage
CMP   #$93       Compare to CTRL S (sound)
BNE   NEXT1      Not equal, try another
JMP   SOUND      If equal, goto sound
CMP   #$92       Compare to CTRL R
BNE   NEXT2      Not equal, try another
JMP   START      If equal, goto start
CMP   #$9B       Compare to ESCAPE
BNE   NEXT3      Not equal, try another
JMP   HALT       If equal, halt program
```

Notice after the CMP #$92 (compare to CTRL R) the jump to a location. This is most likely your starting address. This address could have also been jumped to by means of a BEQ too, so keep that in mind.

Test your new found starting address by turning on the hi-res page manually (the game might not do it for you at that starting address), and then type the starting address followed by a "G". Note that after you turn on the hi-res page, you can no longer see what you are typing on the screen, so type carefully. For example:

```
*C050     (you will be blind after you type RETURN, so type carefully)
*C057     (turn on hi-res graphics)
*C055     (if using page 2 graphics)
*Start address G
```

If these two described processes do not provide you with a starting address, there are a couple of other things to look for. The first is a "jump table", which more experienced programmers generally use. This looks like this:

```
JMP   $4050
JMP   $4000
JMP   $900
JMP   $931A
.
.
.
```

Try any of these as starting locations, but you are kinda poking in the dark with this one. In addition, the above JMP's could also be JSR's too. Just try executing the beginning of the jump table in that case.

Lastly, a lot of programs start by setting up a bunch of zero page locations with parameters and so forth. This generally looks like this:

```
LDA   #$00
STA   $03
STA   $05
STA   $07
LDA   #$01
STA   $7F
LDA   #$80
STA   $FE
STA   $FF
```

Try starting the program with a starting address as the beginning of the zero page set-up routine.

As you can see, finding the starting address can be a time consuming endeavor, and may not even produce any results. Do not be discouraged since practice will make it easier.

If you can not find the starting address, it may be because the program uses some "volatile" memory location that get distroyed when you hit reset. When you try and re-start the program, it sees these location do not contain what they should, and refuses to run. These locations include the text page ($400 to $7FF) and pages $01 and $02, and some zero page locations. The most obvious clue to the use of volatile memory is when you reset into the monitor the text page is filled with "garbage". This garbage is actually code displayed in ASCII form across the text page. When you type something, it is echoed on the text page (and hence memory from $400-7FF). Ordinarily, it's impossible to capture this memory without some additional hardware.

The Senior PROM has the ability to handle this. Use the option from CTRL - RESET that moves $00-8FF up to $2000-28FF. This Senior PROM reset option is especially designed to allow you to view these volitile memory locations. The volatile memory is moved up to $2000-28FF, so you can easily see what was on the text page by typing "2400L".

Occasionally, the author will be kind enough to provide the starting address for us! Note on some program that a normal Apple //e reset will restart the program. This is a blessing, since all we must do is to activate the Senior PROM, reset into the monitor, and check locations $3F2 and $3F3 to find the starting address. These two locations will point right to the starting address, in backwards order. For example, if you reset into the monitor and type "3F2.3f3" and see "00 60", the starting location is $6000. Many authors provide a program restart on a normal reset, so look for it. It's the easiest way to find the starting location.

If all else fails, and you can not find the starting address, there is one more choice: NMI. The Senior PROM will allow you to use the NMI (Non Maskable Interrupt) to interrupt the program and restart it without knowing the actual starting location. This technique will allow you to stop the program and restart it right where you left off. The use of the NMI is throughly described in the Senior PROM documentation, and will not be reveiwed here.

Step 2: What Portion of Memory to Save:

Now that the starting address of the protected program has been found (or you are using the NMI), the next step is to determine what portions of memory should be saved in the final binary file. We cannot save all of the 64K memory since we must use DOS to load the program back in. Basically, we have $8E pages of memory to play with and save as a maximum, out of the available $FF pages that exist. NOTE: There are ways of saving more that $8E pages of memory in a standard Bfile, but this is beyond the scope of this manual.

With our memory limitations in mind and the starting address at hand, we can find what portions of memory we need to run our program. The best way to start out is to turn your Apple on, and use the Senior PROM to put a memory test pattern in all 64k of RAM. This way, after the program is loaded, we can examine memory and see what is loaded in by the program.

Now boot your protected program by pressing CTRL - RESET, and then the reboot key. As the drive recalibrates, deactive the Senior PROM (it's not always necessary to deactive the Senior PROM when booting a disk, but it's highly suggested as some programs will not load properely with it activated).

After the program is loaded, activate the Senior PROM, reset into the monitor and get the inspector up by typing "D000G". Don't go through the normal Senior PROM menu system as this will write over memory $B700-BFFF with RWTS (unless you use the "protect RWTS" option). Now search through memory for blank pages using the "<" and ">" keys. Write down on paper any blank memory areas you find (don't try and remember them, just write them down).

Alternatively, you can flip through memory using the monitor "L" command, but this will take an incredible amount of time. Also you can use the Sector Editor's Disassembly command (CTRL D).

Also be aware of "garbage memory" and shape tables". Garbage memory is unused junk that does not disassemble. Shape tables look like garbage memory but actually contain graphic shapes and the such. Write down these suspect garbage memory areas on paper.

Also check how the program starts. Does it turn on the hi-res page and use what is already there, or does it re-draw the hi-res page? If it re-draws, you do not have to save that hi-res page (either $2000-3FFF or $4000-5FFF). Write that down too.

The best way to check to see if a memory area is used is to load the program and reset, zero the suspect memory area, and restart the program. Run the program for a while and if it works OK, then that memory area is not needed. IMPORTANT: BE SURE TO CHECK ALL FACETS AND LEVELS BEFORE DISCARDING A MEMORY RANGE.

The best way to zero a memory portion is to use the monitor's move command. For example, say you want to test to see if hi-res page 1 is re-drawn ($2000-3FFF). From the monitor prompt type:

*2000:0
*2001<2000.3FFFM

This will zero out $2000 to $3FFF. Now restart the program and check it out.

Make sure you keep careful notes of what is needed and what is not. WRITE DOWN EVERYTHING ON PAPER. AFTER YOU HAVE FOUND ALL NEEDED MEMORY AREAS, ZERO ALL UN-NEEDED AREAS AND RUN THE PROGRAM FOR A WHILE TO VERIFY.

Step 3: Saving the Program as a Bfile:

Now that you have found the starting address and what portions of memory the protected program encompass, you have to get the memory portions to a standard DOS 3.3 disk. The best way to do this is to use the Senior PROM reset option that moves all of main 64k RAM to auxilary 64k RAM. Then you can boot a DOS disk and use the Memory Management Menu options to move memory back to main RAM and save it to disk.

In saving small portions of memory to disk, I should also note the importance of a "48k Slave disk". Booting a 48K slave disk does not disturb memory from $900 to $95FF. The best way to make a 48K slave disk is to boot any normal DOS 3.3 disk (fast DOS with INIT preferred such as Pronto-DOS by Beagle Bros, or Diversi-DOS by Diversi), and type:

]FP
]INIT HELLO

The disk created will be a 48K slave disk. Don't use any "hello" program as this might load over memory between $900 and $95FF.

A slave disk will give us a total of $8D pages of memory undisturbed when booted (a page of memory is $100 hex locations or 256 decimal locations). If a program is bigger than $8D pages of memory or doesn't lie in the $900-95FF range, use the Senior PROM's memory-move option that moves main 64k memory to Aux 64k memory, or save the program in several steps and pieces with a slave disk and the monitor's move command.

Congratulations, you now have all of your protected program saved on a normal DOS 3.3 disk. But you are not finished, since you can not simply run any of these files. Now you must put them all into one file and move the pieces of memory back to where they belong.

In addition all parts of memory can not simply be re-loaded since some might overwrite DOS. All program portions between $800 and $95FF can easily be replaced, but the $9600 to $BFFF region must be loaded below DOS and moved back up to where it belongs (if this memory region is needed), and then jump to the beginning of the program.

This is accomplished through the use of memory moves. A memory move is a short Assembly language program that moves portions of a program from one part of memory to another.

Here is an example memory move program:

```
4A00-   A2 00       LDX #$00
4A02-   BD 00 08    LDA $0800,X
4A05-   9D 00 96    STA $9600,X
4A08-   E8          INX
4A09-   DO F7       BNE $4A02
4A0B-   EE 04 4A    INC $4A04
4A0E-   EE 07 4A    INC $4A07
4A10-   AD 07 4A    LDA $4A07
4A13-   C9 B0       CMP #$A7
4A15-   DO E8       BNE $4A00
```

This short assembly language program moves memory from $800-18FF up to $9600-A6FF. Since DOS lives from $9600-BFFF, we could not just load the memory area $9600-A6FF right from the disk using DOS. First we have to load it into lower RAM (somewhere between $800-95FF) from a B-file and then use the above program to move the memory back up to $9600-A6FF.

Memory Move Writer (included on the Senior PROM disk) will write this type of memory move routine for you. Remember DOS lives from $9600-BFFF, so you have to organize (store) required memory portions that original lived in the $9600-BFFF memory area between $800-95FF. Then the memory move program will move the appropriate memory back to where it belongs.

Now to put it all together. Bload the files into the appropriate place. If the start of the program is not at the beginning of the file, we must enter the monitor and tell the program where to jump to when the file is BRUN. For example, lets say we have the program loaded at $800 to $7FFF, and the start address is $4A00. This is what we would have to do after all the file sections are loaded in the correct places:

```
]CALL -151
*7FD:4C 00 4A
*A964:FF                (enables us to save large binary files)
```

Note that we enter a JMP to the memory move routine at $4A00 three bytes before $800. This is because the jump instruction takes three bytes (4C 00 4A).

Now you can save the whole file to your disk using a standard BSAVE command:

```
*BSAVE WHOLETHING, A$7FD,L$7803
```

You can determine the length parameter by typing the starting and ending pages from the monitor, separated by a minus sign. For our example, we would type:

*80-08

and $78 will be returned. Since we want those three extra bytes for the initial jump to the memory move routine, we need to add three to that. Hence we arrive at a length of $7803.

BRUNing this file should restart the program and all should be fine.

Final Notes:

If your program has some additional minimal disk access for saving high score and the like, refer to the appendix entitled "Removing Minor Disk Access" for help with that.

By far the most popular protection scheme ever used is the modified DOS protection. This scheme bases its protection upon normal DOS 3.3, but makes some vital changes to the DOS to read a perverted disk structure To understand this protection scheme, I will first have to describe how a normal DOS 3.3 disk in formatted.

Each normal DOS disk has 35 tracks (labeled 0-34 or $0-$22 in hex) and 16 sectors per track (labeled 0-15 or $0-$F in hex). Each sector represents one page of data (256 or $100 hex bytes). Within each sector there are two separate fields: an address field and a data field. First let's discuss the address field.

If you read a disk with a "nibble read" routine (use the Sector Editor's "N" command from with the Senior PROM), you will see a disk's data in the raw form. Scanning through the data you should see something like this:

```
FF FF FF FF D5 AA 96 FF FE AA AA AA AA FF FE DE AA EB
   (1)          (2)    (3)  (4)    (5)    (6)    (7)

FF FF FF FF D5 AA AD -342 bytes- XX DE AA EB
   (8)          (9)      (10)    (11)  (12)
```

The first few FF's (1) are known as syncbytes, and are used as separators, or borders. The next three bytes (2) are called the address prologue bytes, and are very important. This sequence of three bytes will not be found anywhere else on a disk except the address field. These bytes serve as unique identifiers to DOS so it can find what track and sector it is reading (hence "soft sectoring"). The data following the address markers are therefore, address identifiers. This includes the disk's volume number (3), track number (4), sector number (5), and checksum (6). The format is a little strange, and is called 4+4 nibblizing. This format stores data in which the even bits of a byte are stored in one 8-bit sequence and the odd bits are stored in a second 8-bit sequence. In other words, it takes two bytes to store one byte. This was originally done because of limitations imposed by disk drive hardware.

The address identifiers are all pretty obvious, except for the checksum. The checksum tells DOS that everything checks out OK when reading the disk.

The next set of bytes are the address field epilogue bytes (7). These are used to mark the end of the address field. A total of three bytes are used (DE AA EB), but only the first two are checked when the field is read. The epilogue bytes are really not needed, but provide added assurance that the drive is still in sync with the bytes on the disk.

The next set of bytes (8) are more syncbytes which separate the address field from the data field. The second part of a sector is the data field. The first three bytes are the data prologue bytes (9) and they tell DOS that the data follows. In raw form, 256 bytes of memory data is represented in 342 bytes of disk data (10). Each disk byte represents 6 bits of a memory byte (remember there are 8 bits to a byte). Therefore it takes 342 disk bytes to represent 256 memory bytes. Once again, this is used because of disk drive hardware limitations.

At the end of the data is a single checksum byte (11). The checksum is a number which when exclusive ORed with the rest of the data in a sector equals zero. If this number does not equal zero, DOS thinks thinks something is wrong and gives you an "I/O ERROR".

Finally, there are data field epilogue bytes (12) which also make sure the drive is in sync with with the disk.

Now this brings us to our first and most popular protection trick: changing the epilogue bytes of either the data and/or address fields from the standard. This is really not a very good protection scheme. If the prologue bytes are not changed, normal DOS can still find the address and data fields (and knows how long each should be), and can still read the data. An I/O ERROR will occur though, because the epilogue bytes can not be found to correctly mark the end of the data.

To read a disk in which the epilogue bytes have been changed is very easy. From with the Senior PROM RWTS menu, enter option 5, "Change Prolog bytes". Now just press RETURN for each of the prompted values. This will maintain the normal DOS prologue bytes, but defeat the DOS epilogue byte error checking. You can now use option 2 (copy a disk) to copy the protected disk.

Alternatively, you can use option 4, defeat the DOS error checking at $B942. I would suggest this as a second choice to the previous method. The reason is that this defeats all of DOS's error checking, and is not quite as reliable as just defeating the epilogue byte error checking.

A program that uses this changed epilogue byte protection exclusively is "Money Street" from Bullseye Software. But more traditionally, this mild protection is combined with other modifications.

Another a very common protection technique is to change the address and/or data field prologue bytes. This will immediately choke up any normal DOS copy program and even confuses Locksmith and Essential Data Duplicator (EDD) on occasion. This is because normal DOS can not find where the address identifiers are, so it can not figure out what sector it is reading. Also, if the data field prologue bytes are changed, it cannot find where the data starts.

The way to find out if any of these bytes have been changed is to do a nibble read (using the Sector Editor's "N" command from the Senior PROM) of the protected disk and to examine the data. Look for the landmarks as described above. It is usually best to read a series of non-DOS tracks like track 3, 17 and 20 when doing a nibble read of a disk to make sure you get the same results (but of course, some protectors have even used different address and data markers for each track!).

Alternatively, you can load the program, reset into the monitor and examine the DOS locations that hold the address and data field bytes. Compare them to what they should be and note any differences.

Once you have determined what address or data field bytes have been changed, it is a breeze to copy the disk with the Senior PROM. Just enter the Main Menu and use Option 5 to change the Address and Data Prologue bytes. Then use Option 2 to copy the disk.

Here is a table of locations in RWTS to check and their original values in regards to the address and data field bytes:

Address Prologue Bytes:			
47445	($B955):	213	($D5)
47455	($B95F):	170	($AA)
47466	($B96A):	150	($96)
Address Epilogue Bytes:			
47505	($B991):	222	($DE)
47515	($B99B):	170	($AA)
Data Prologue Bytes:			
47335	($B8E7):	213	($D5)
47345	($B8F1):	170	($AA)
47356	($B8FC):	173	($AD)
Data Epilogue Bytes:			
47413	($B935):	222	($DE)
47423	($B93F):	170	($AA)

Now that you have converted a disk to a normal DOS 3.3 format, that does not guarantee that it will work! You may have to find other routines that expect certain address and data markers to be different than normal and modify or defeat them. If it is a BASIC program, look for PEEKs and POKEs above 47000 and track them down.

Probably the most common "fix" is to change track 0, sector 3, byte $42 from $38 to $18. This is actually changing location $B942 in the DOS tracks, and this one single mod is the eqivalent of defeating the normal DOS error checking at $B942 permanently. This will easily make many programs that use modified epilogue bytes only run, but I do not suggest it for the same reason I don't suggest copying a disk with the DOS error checking at $B942 defeated. Use this as a last resort.

It is better to check track 0, sector 3, bytes $35 and 3F for modified data epliogue bytes, and track 0, sector 3, bytes $91 and 9B for modified address epiloge bytes. Then change them to the standard DOS 3.3 epilogue bytes of $DE AA. This is safer for disk integrity.

If the program used modified prologue bytes, you will have to search for the routines in the program's DOS tracks. Once you have found them you will have to modify or change the routine to accept normal DOS prologue bytes. I would check track 0, sector 2, bytes $E7, F1 and FC for modified data prologue bytes. Also check track 0, sector 3, bytes $55,

5F and 6A for modified address prologue bytes. If these are different than the standard $D5 AA AD and $D5 AA 96 respectively, change them.

It should be noted that some programs change the address and/or data marker bytes as the program runs. Examples of this are adventure games and business programs that run from a protected disk but save the game or other data to a normal DOS 3.3 disk. In this case you must find these "byte swap" routines and defeat them or change them so they only read or write using the normal DOS epilogue and/or prologue bytes.

An easier and more general method of converting protected DOS disks to a normal format is COPYB. COPYB simply uses the RWTS (read - write - track - sector) portion of the protected DOS to read the original disk, and uses normal DOS RWTS to write to a normal DOS disk. On the surface this is a much easier and a more general method of deprotection than trying to figure out exactly what the disk is actually using for prologue or epilogue bytes, but is also not as flexible. Its main advantage is that you just save the foreign DOS's RWTS, and use it to read the protected disk. It is best to become familiar with both methods and how to use them.

I have devoted a appendix to the subject of using COPYB in deprotecting programs. Please refer to that appendix for more information on COPYB.

To summarize so far, we have discussed the most popular methods of protecting a disk using a modified DOS. This usually means modification of the address and/or data field epilogue and/or prologue bytes. At this point, I would like to mention modified DOS protection used in ProDOS disks.

Recently, ProDOS is being used in protection. This is really a derivative of the modified DOS protection scheme since software publishers leave the ProDOS operating system alone, and don't try to modify it for protection. Alternatively, they use a SYSTEM file to incorporate their protection. An example of this is Apple's Logo // (128k version). But of all the protection types, the ProDOS based protection schemes most closely resemble modified DOS protection.

The ProDOS boot process is considerably different than the DOS 3.3 boot process. First, track 0, sector 0 is read into $800-8FF by the disk controller PROM code. Then this code uses the disk controller PROM code as a subroutine, and reads in track 0, sector 1 into $900-9FF.

Next the code at $800-9FF searches the ProDOS directory (on the remainder of track 0) for the ProDOS file and loads it at $2000. This code is the ProDOS operating system, and gets relocated into the upper 16k of RAM.

Next the director is searched for a SYSTEM file and loads the first SYSTEM file it finds. Normally, this is BASIC.SYSTEM which contains the BASIC shell. The original purpose of SYSTEM files was to allow ProDOS to load in any language, in the form of a SYSTEM file (but to date Apple has only released BASIC.SYSTEM).

Since Apple periodically releases new versions of ProDOS and encourages its users to update their disks, publishers really can't modify ProDOS like they did to DOS 3.3 for protection schemes. Instead, the protection is added to the SYSTEM file.

From what I have seen, a nibble count or some specially formatted tracks are the protection. The whole disk is not protected so a user can update the disk to the new ProDOS release. Maybe only one or two tracks are formatted in a perverted form and are checked using a "nibble count" or a prologue/epilogue check routine (these are described in the next chapter).

The defeat this type of protection involves finding the checking code and changing it to expect normal DOS prologue/epilogue bytes, or defeating the nibble count routine. This is explained in more detail in the next chapter.

Chapter 5: Modified RWTS Protection.

The modified RWTS protection is an offshoot of the modified DOS protection, but instead of using an entire DOS, just the RWTS portion of DOS is used. Note that this chapter overlaps chapter 4, Modified DOS Protection, in several areas.

RWTS is roughly the upper third of DOS. Many programs use only RWTS, or a portion of RWTS because of memory limitation, better protection, or pure finesse. It is also very easy to control disk drive access from ASSEMBLER, so it is a natural to use RWTS calls from ASSEMBLY language programs and to ignore the rest of DOS altogether.

Sometimes mere portions of RWTS are used in the form of a loader. A short program (around $300 bytes) can be written to just read from a disk, thus saving memory and also making the disk access harder to find. This is typical of many of the older Sirius games, along with some others.

Because a loader or RWTS tends to be a short program, much of the normal DOS error checking is defeated and forms of protection are included. For example, perhaps a nibble count is added to the loader to check if the disk is an original. Also very common is the address and/or data prologue or epilogue bytes are changed from the norm.

To deprotect a program which is using a loader or RWTS can be difficult or very easy, much like any of the other kinds of protection. Generally, you have to do three things: By-pass the nibble count or checksum routine, modify the loader to read normal DOS address/data headers, and copy the modified disk to a normal DOS format and make the changes to that disk.

The first thing you will have to do is to get the data from the protected disk to a normal DOS 3.3 format. Try defeating the epilogue byte check routine by using option 5, "Modify Prolog Bytes". Just press RETURN for the default prologue bytes and the epilogue byte error checking will automatically be defeated. Now select option 2 and copy the disk.

If this doesn't work the second thing to try is to defeat the DOS error checking routine at $B942 using option 4. Then use option 2 and try and copy the disk. This routine sets the carry flag when ever DOS thinks it can not read a sector. DOS monitors the carry bit, and if set, bomb out with some dumb error message.

Many times you will only have to read a portion of the protected disk, because it does not use all the tracks. You can use option 2, "Copy a Disk, to copy any range of tracks you wish.

As described in the previous chapter (Modified DOS Protection), you may have to determine what the prologue and epilogue bytes being used are. Then you can use option 5 to modify the prologue bytes and copy the disk. Beyond this, the protection is more involved and you will have to dig for the answer....

After making a normal DOS copy of the protected program, it probably won't run, and you will have to change the modified RWTS so it can read a normal DOS format. To do this, reset from the protected program and check these locations for anything funny (remember that RWTS normally lives from $B700 to $BFFF):

Address	Normal Value	Use
$B853	D5	Data Prologue Byte 1-WRITE
$B858	AA	Data Prologue Byte 2-WRITE
$B85D	AD	Data Prologue Byte 3-WRITE
$B89E	DE	Data Epilogue Byte 1-WRITE
$B8A3	AA	Data Epilogue Byte 2-WRITE
$B8E7	D5	Data Prologue Byte 1-READ
$B8F1	AA	Data Prologue Byte 2-READ
$B8FC	AD	Data Prologue Byte 3-READ
$B935	DE	Data Epilogue Byte 1-READ
$B93F	AA	Data Epilogue Byte 2-READ
$B92A	D9 00	Location Checksum Compare
$B942	38	Set Carry for I/O Error
$B955	D5	Address Prologue Byte 1-READ
$B95F	AA	Address Prologue Byte 2-READ
$B96A	96	Adrress Prologue Byte 3-READ
$B991	DE	Address Epilogue Byte 1-READ
$B99B	AA	Address Epilogue Byte 2-READ
$BC7A	D5	Address Prologue Byte 1-WRITE
$BC7F	AA	Address Prologue Byte 2-WRITE
$BC84	96	Address Prologue Byte 3-WRITE
$BCAE	DE	Address Epilogue Byte 1-WRITE
$BCB3	AA	Address Epilogue Byte 2-WRITE

If any are different, locate the different byte sequence on the normally formatted version of the disk using the Sector Editor's Locate command and change the locations back to normal using the Sector Editor.

Many times, this is all that is necessary, but not always. Maybe there was a "nibble count" that will prevent the deprotected copy from running. A "nibble count" is an overused and general term on the Apple for a routine that looks for a specially formated track or sector. Originally it was a disk routine that read the number of nibbles on a track, and compared it to some benchmark. If it didn't match the benchmark, the program won't run.

The reason this was such good protection was because each track has a different number of nibbles on it due to differences in disk drive

speeds. DOS compensates for these different drive speeds using "Sync Bytes", which are written between the address and data fields. These Sync Bytes act like fillers, allowing drive with differnce speeds read the data. Because of this, it was nearly impossible to bit copy a disk and produce a duplicate with exactly the same number of nibbles on a track as the original copy protected disk.

Finding the "nibble count" could be the trickiest part. Search through the program looking for any JMP's or JRS's to the DOS area, or for any instruction codes that turn the disk drive on or off. Beyond this it's up to you.

After you find it in memory, locate the routine on the disk (using the Sector Editor's Locate command) and defeat it. You can do this by putting a $18 60 at the beginning of the "nibble count" subroutine (Clear the Carry Flag, Return from subroutine), or by "NOPing" the call (no operation) to the subroutine with three "$EA"s.

Another way to find a nibble count is to try running your converted, unprotected version, and when the nibble count is encountered, interrupt the program and examine memory then. Using the NMI feature of the Senior PROM can be helpful to find the location of the "nibble cont" (refer to the NMI portion of the Senior PROM manual).

In addition, the program may be using a loader that doesn't live in the $B700 to $BFFF range. These can be tricky to find, and even tricker to make work in a modified DOS environment.

I would first do a nibble read of the disk and find out what the address and data prologue and epilogue bytes are. Then look for code like this:

```
0350-    B9 8C C0      LDA $C08C,Y
0353-    C9 D5         CMP #$D5
0355-    D0 F8         BNE $0350
0357-    B9 8C C0      LDA $C08C,Y
035A-    C9 AA         CMP #$AD
035C-    D0 F8         BNE $0357
.
.
```

This type of code accesses the drive and compare some data header to make sure it is the protected format. All you need to do is to find this routine on the disk, and change the headers to normal DOS. Easy, huh?

Also try to get familiar with direct calls to RWTS. If you can recognize these, you can usually find from where the data is being loaded from. Here is the parameter list for using DOS from ASSEMBLY language:

$B7EA: Drive number to use ($01 or 02)
$B7EB: Volume number ($00 = any volume number)
$B7EC: Track number to read
$B7ED: Sector number to read
$B7F0: Lo-byte of memory page to read/write
$B7F1: Hi-byte of memory page to read/write
$B7F3: Partial sector read (0=whole Sector)
$B7F4: Command code (0=seek, 1=read ,2=write)
$B7F5: Error code (valid only if Carry Flag is set)
$B7B5: Entry point for actual read/write.
$BD00: Alternative entry point for read/write

Here is a sample program that uses these parameters:

```
9000-   A9 04      LDA  #$03    :track 3.
9002-   8D EC B7   STA  $B7EC   :store at track parm.
9005-   A9 0F      LDA  #$0F    :sector $F.
9007-   8D ED B7   STA  $B7ED   :store at sector parm.
900A-   A9 00      LDA  #$00    :use any volume number.
900C-   8D EB B7   STA  $B7EB   :store at volume parm.
900F-   8D F0 B7   STA  $B7F0   :use whole pages of memory (lo-byte).
9012-   A9 4F      LDA  #$20    :page number desired (hi-byte).
9014-   8D F1 B7   STA  $B7F1   :store at page parm.
9017-   A9 01      LDA  #$01    :command code (in this case, READ).
9019-   8D F4 B7   STA  $B7F4   :store at command parm.
901C-   A0 E8      LDY  #$E8    :set parms to use
901E-   A9 B7      LDA  #$B7    :parm table at $B7E8.
9020-   20 B5 B7   JSR  $B7B5   :do the read.
9023-   CE ED B7   DEC  $B7ED   :decrement the sector.
9026-   EE F1 B7   INC  $B7F1   :increment the page.
9029-   AD ED B7   LDA  $B7ED   :compare current sector to see if
902C-   C9 FF      CMP  #$FF    :need to reset sector and change track.
902E-   D0 EC      BNE  $901C   :still on same track so go back and read.
9030-   A9 0F      LDA  #$0F    :reset sector number.
9032-   8D ED B7   STA  $B7ED   :store at sector parm.
9035-   CE EC B7   DEC  $B7EC   :decrement track.
9038-   AD EC B7   LDA  $B7EC   :load accumulator with track number.
903B-   C9 01      CMP  #$01    :compare to last track - 1 desired.
903D-   D0 DD      BNE  $901C   :not last track, go back and read more.
903F-   60         RTS          :return to calling subroutine.
```

Try and figure out what this does...This reads track 3, sector F down to track 2, sector 0 into $2000 to $3FFF. This would be a perfect example for reading a hi-res picture into hi-res page 1 using RWTS. Be able to recognize these routines. The dead giveaway is the JSR $B7B5. All the parameters could have been loaded through other locations, such as zero page locations, then re-loaded into RWTS. The JSR $B7B5 gives it all away!

Also notice how the routine reads sectors "backwards", or down. This is done for speed reasons, and none other. If we incremented the sectors instead of decrementing them, this routine would take 4 times longer to load. It does not matter if we increment the tracks or memory pages though...just the sectors (this has to do with "sector skewing").

Chapter 6: Disabling Minor Disk Access.

Many single load programs have a small amount of disk access that you will have to defeat in order to deprotect the program into a single file. I will give you a short cut that should help in your efforts.

Many programs use some minor disk access to write high scores to the original disk, or to check if the original disk is still in the drive at some predetermined point. Of course if we want to deprotect the program into a single file, we must defeat this.

A example is just about all of Penguin's $19.95 arcade games such as Spy's Demise, Crime Wave, The Spy Strikes Back, Bouncing Kamungas and others. If you carefully examine these programs (they can all be deprotected into a single file with no trouble) you will find a JMP $BD00 somewhere in the code. This call to RWTS does the disk access for their games. You can easily defeat the disk access by replacing the "4C 00 BD" with "EA EA EA".

If you aren't so fortunate as to find (or know about) this RWTS call, there is another way to defeat their disk access. That is to replace all of RWTS with the byte sequence "18 60". This represent a "clear the carry" and a "return from subroutine". This can easily be done with a small routine that you can put in a deprotected single file.

Usually, I call this routine before I do any memory moves and after all disk access. The routine looks like this:

```
6000-   A0 B7     LDY  #$B7
6002-   84 01     STY  $01
6004-   A0 00     LDY  #$00
6006-   84 00     STY  $00
6008-   A2 60     LDX  #$60
600A-   A9 18     LDA  #$18
600C-   91 00     STA  ($00),Y
600E-   8A        TXA
600F-   C8        INY
6010-   91 00     STA  ($00),Y
6012-   C8        INY
6013-   D0 F5     BNE  $600A
6015-   E6 01     INC  $01
6017-   A5 01     LDA  $01
6019-   C9 C0     CMP  #$C0
601B-   D0 ED     BNE  $600A
601D-   60        RTS
```

This routine is relocatable (will run at any address) and will wipe out RWTS ($B700 to $BFFF) with the byte sequence "18 60". Notice it uses zero page locations $00 and $01, so you should call this routine before you move or alter any zero page locations, or save locations $00 and $01 somewhere and then restore them after using this routine.

The reason we use "18 60" is simple: "18" represents a "clear the carry". DOS monitors the carry bit, and if set, it knows there was a I/O error (obviously we don't want the program to think there was an I/O error). The "60" returns us to the calling routine. This has the effect of letting the program think it did the disk access, without really doing it. Therefore, the program goes happily on its way.

The reason we wipe out all of RWTS with this routine is because we don't know where the calling routine is jumping to in RWTS, just that it is in the RWTS region of DOS somewhere.

This is a nice way of defeating disk access in a program even if you don't know where the disk access is actually calling to or from!

Of course, the above routine assumes that RWTS is in its normal place of $B700 to $BFFF. If it is not, we can change the second (2nd) byte of the routine to the starting page of RWTS, and the fourth (4th) from last byte to the ending page of RWTS.

For another example, you could use a similar routine to defeat the disk access in "Mating Zone" from Datamost. In this case, the loader (RWTS) lives from $400 to $7FF, and can be replaced with a sequence of $60's across this memory range. Then when the program jumps to the loader in the $400-$7FF range, it encounters a "60" (return from subroutine) and it immediately returns thinking it did the disk access!

Many protected programs are written in APPLESOFT. Of course, most publishers are sly enough to protect against breaking out of their program with CTRL C or reset. Also, most protect re-entering BASIC from the monitor by changing the typical BASIC re-entry point (at $3D0) so that it points to disaster. And lastly, many change the RUN flag vector at $D6 so if you manage to get out of their program and into BASIC, anything you type will automatically RUN their BASIC program. This chapter describes how to beat all these protection schemes using the Senior PROM.

First, we must determine if the protected program is written in APPLESOFT. If after you boot the program a BASIC prompt appears, this is a good indicator that at least some of the program is written in BASIC. Furthermore, if the program prints a lot of text on the screen, or requires a good deal of user inputs, it is a good guess that the program is written in BASIC. The reason for this is that printing text on the screen and inputing data from the keyboard are easily accomplished from BASIC using PRINT and INPUT statements. To do this from ASSEMBLY language requires a great deal more work. Also, we should realize why a programmer uses ASSEMBLY language. The only real advantage to ASSEMBLER is speed. If speed is not critical, most (non-sadist) programmers will use BASIC.

With this in mind, look at how the program runs and prints on the screen. If it runs at about the same speed as the BASIC programs you have written run, it is a good guess that it is in BASIC. Remember, ASSEMBLY language is considerably faster than BASIC in every respect.

Finally, read the package the program came in. It usually says what it was written it. If it doesn't, a dead giveaway is in the hardware requirements. If the program requires APPLESOFT in ROM, then at least part of the program is probably written in APPLESOFT.

Now that you have figured out your protected program is written in BASIC, it is time to LIST their code. The first step is to interrupt the program when it is running. To do this, deactivate the Senior PROM and press Reset. Now move the toggle switch to the middle transparent position, and boot the protected Basic program. Then at the strategic moment, press the NMI button, and use the Senior PROM as desire (save memory and create a Resurrect disk, or enter the Monitor, etc.).

It is important to use the middle transparent switch position and the NMI button only to interrupt a basic program. The reason is if you activate the Senior PROM, the Basic program will immediately halt, limiting some of your Senior PROM snooping abilities. This is because BASIC has been replaced with the many utilities within the Senior PROM when the Senior PROM is activated. You are actually pulling the BASIC interruptor out from under the BASIC program, which will usually break you into the monitor or freeze the program.

This is NOT TO SAY that when you active the Senior PROM you will lose the BASIC program in memory. On the contrary, you only lose the BASIC interpretor in ROM, and not the BASIC program in RAM.

Now you can try to enter the immediate BASIC mode by deactivating the Senior PROM switch and typing:

```
*[RETURN]              (press the RETURN key)
*3D0G
```

This is the normal BASIC re-entry point, but if the protection is worth anything, this will not work.

Assuming that it didn't work, reload the program and reset into the monitor again using the Senior PROM, and then press RETURN. Deactivate the Senior PROM and then type "9D84G" or "9DBFG" instead of "3D0G". These are the DOS cold and warm start routines, respectively. If you are lucky enough to get a BASIC prompt, you have done well. Most of the time, you won't.

If in either case you succeed in getting a BASIC prompt, try LISTing the program or CATALOGing the disk.

Remember you must alwasy deactivate the Senior PROM before trying to re-enter Basic!

If anything you type starts the program running again, the protection has changed the RUN flag at $D6. The RUN flag is a zero page location (at $D6) which will run the BASIC program in memory if $D6 contains $80 or greater (128 or greater in decimal). This is easy to defeat after you have reset into the monitor by typing:

```
*D6:00
```

This resets the RUN flag to normal. Now if 3D0G, 9D84G or 9DBFG previously rewarded you with a BASIC prompt, this will solve the problem of the program re-running when you type a command.

In protection schemes, the RUN flag is usually changed from within a BASIC program by issuing the code:

```
10 POKE 214,255
```

or by poking location 214 with anything greater than 127. From ASSEMBLY language, the code would most likely look like this:

```
800-    A9 FF    LDA #$FF
802-    85 D6    STA $D6
```

or by loading any register with $80 or greater and storing it at $D6.

If 3D0G, 9D84G or 9DBFG did not produce a BASIC prompt, then the DOS being used is more elaborate. Load the program and reset into the monitor after it is running.

Now comes the final steps in trying to examine BASIC programs. Boot the suspect program and use the Senior PROM to enter the monitor. Reset the RUN flag to normal, just to be sure. Type:

```
*D6:00
```

Now deactivate the Senior PROM switch and press the RETURN key. This will activate BASIC. Then type:

*[CTRL C]

You should see an APPLESOFT prompt. Now type:

]LIST

and your APPLESOFT program should now appear.

Applying this to a real world example, try this method with one of Strategic Simulations releases (SSI). SSI uses a highly modified DOS called RDOS for their protection. SSI uses all the tricks mentioned to prevent you from LISTing their programs. But using the above procedure, you can LIST their BASIC programs.

In addition, the DOS used by SSI (RDOS) uses the ampersand in all of its DOS commands. So if you see any ampersands from within their BASIC program, you know they are DOS commands. For example, to catalog a SSI disk, after you follow the above procedure and you are in BASIC, type:

]&CAT

This will display SSI's catalog.

If you want to save an APPLESOFT program to a normal DOS disk, do these steps:

1) Reset into the monitor after the program is running, using the Senior PROM as described earlier

2) Now type:

*D6:00
*9500<800.8FFM

3) Check where the APPLESOFT program ends by typing:

*AF.B0

4) Write down the two bytes listed.

5) Deactivate the Senior PROM and boot a 48K normal DOS 3.3 slave disk with no HELLO program.

6) Enter the monitor by typing:

]CALL-151

7) Restore the APPLESOFT program by typing:

*800<9500.95FFM
*AF: enter the two bytes you wrote down here, separated by a space.

8) Enter BASIC and save the program by typing:

```
*3D0G
]SAVE PROGNAME
```

What you have done is to move $800 to $8FF out of the way so you can boot a slave disk. After normal DOS is up, you restore $800 to $8FF from $9500 to $95FF, and then restore the end of APPLESOFT program pointers so DOS knows how big your BASIC program is. Next you just save it to your disk. Of course there are other more automated ways of getting programs to a normal DOS 3.3 disk, but this is a quick and dirty method that will almost always work. Keep in mind that the program may not run from normal DOS because of more secondary protection from within the BASIC program itself. Any curious CALLs, POKEs or PEEKs to memory above 40192 (this is memory where DOS resides) or below 256 (zero page memory) should be examined closely.

This should help you learn more about the protected programs you own that are written in APPLESOFT.

There are probably hundreds of ways to protect a program from being copied, and the most common is using a Modified DOS protection scheme. The classic program for dealing with modified DOS's is DEMUFFIN PLUS. It works much the same way as Apple's MUFFIN program works. MUFFIN was written to read files from a DOS 3.2 disk and then write them to a DOS 3.3 disk. DEMUFFIN was a variation of MUFFIN, allowing the hardcore 3.2 user to copy files from DOS 3.3 to DOS 3.2. DEMUFFIN PLUS operates on the same principle, but uses whatever DOS is in memory to read the disk, and then writes out to an initialized DOS 3.3 disk uses normal DOS 3.3. While this is a powerful utility, it only works with programs that are based on DOS file structures and that have a catalog track.

COPYB (originally written by "Krakowicz") is a highly modified version of COPYA which converts a protected disk with a modified DOS and/or RWTS to normal DOS 3.3 format. The protected disk may have a normal DOS file structure, or it may not. Since COPYB copies on a track by track basis, this does not matter. This makes COPYB a far more flexible tool than DEMUFFIN PLUS.

COPYB uses the protected disk's RWTS to read in the tracks and then uses normal DOS 3.3 to write them back out to an initialized disk. Unless otherwise instructed, COPYB copies track $03 to track $22, sector $0F to sector $00 of each track. Here are the internal parameters for COPYB:

Location			Normally		
Hex	Dec	Description	Hex	Dec	Note
$22E	558	First track to read	$03	03	(1)
$236	556	First sector to read	$0F	15	(2)
$365	869	Reset sector number	$0F	15	(2)
$3A1	929	Stop on error ($18=NO)	$38	56	(3)
$302	770	Track to stop reading+1	$23	35	(4)
$35F	863	Track to stop reading+1	$23	35	(4)

1) This is the first track that COPYB reads. This is normally set at track 3, so not to copy the protected DOS which normally resides on track 0 through track 2.

2) These two parameters are normally set to $0F for 16 sector disks. Change these two parameters to $0C for 13 sector disks. Most of today's protection schemes are based on 16 sectors, yet there are still a few using 13 sectors (such as Muse). Interestingly enough, there is a handful of authors that also us sectoring other than 13 or 16 sectors per track. An example of this is "Thief" from Datamost. This program uses 11 sectors per track. COPYB can also accommodate these programs.

3) This parameter is normally set so that upon reading a 'bad sector' COPYB will stop and display an error. To let COPYB keep going after a read error, change this byte to $18 (24 in decimal). The equivalent sector on the copied disk will be written blank.

4) These two parameters determine where COPYB will stop reading the protected disk. Normally, this is set to the last track, $22 (34 in decimal), plus one. To change this, add one to the last track you want to copy and change these two parameters.

Using COPYB:

To use COPYB, you must capture the foreign RWTS and put it at locations $8000 through $88FF. You can do this one of two ways:

1) Boot the protected disk and after the foreign DOS is loaded, reset into the monitor. The foreign DOS will usually be loaded a few seconds after the boot starts. You can tell this because many times a BASIC prompt will appear at the bottom of the text screen. Use the monitor move command to move RWTS down to $8000 as so:

*8000<B700.BFFFM

Now boot a 48k slave disk (this will not destroy memory from $900 to $95FF) and run COPYB.

2) Alternatively, read in track 0, sector 1 through track 0 sector 9 of the protected disk into memory $8000 to $88FF with the Sector Editor. Then run COPYB.

Running COPYB and Entering the Parameters:

Run COPYB by typing:

]RUN COPYB

The program will come up and ask what parameters to use, all described above. COPYB will poke in the values for you. Enter all values in DECIMAL.

After entering the parameters, you will be asked if your selections are correct. If you answer YES, the next set of prompts will appear, which should look familiar. Enter the original and destination drive and slot numbers, just as in COPYA. Lastly, you will be asked if you want the destination disk to be initialized, respond yes or no. Now press the RETURN key to start the copy.

When the copy is completed, assuming all went correctly, you will have a normal DOS 3.3 version of your protected disk which may run or be examined and changed more easily then the original disk.

This method of deprotection is more dependable that using DEMUFFIN PLUS and covers more types of programs. You will find COPYB an excellent utility to have.