Chapter 1.

Introduction to kraking.

Anyone interested in the Apple seems to be intrigued by the "art of kraking", for a variety of reasons. Probably the foremost is this opens the way for worry free and ample software, that anyone can use and trade. Besides these immediate uses, kraking a program seems to impress all but the author of the program.

I will be discussing the methods used in kraking protected Apple programs. After reading (and understanding!) all the files on this disk, you should be able to deprotect most any commercially protected Apple program. Of course, these files will not make you an instant expert on Apple copy protection, and there are many things to learn that I can not cover. But 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.

Kraking 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 DOS manual and the Apple II reference manual for informative discussions of your computer. Even better, pick up a copy of Don Worth's "Beneath Apple DOS" and study it carefully. After you have done these basic things, you can probably impress everyone but Steve "The Woz", and turn the salesmen at Computer Land to shame.

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 99% of the Apple users that claim they know anything about anything.

Of particular importance is the "Apple II Reference Manual". The most important part of this manual is chapter 3, "The System Monitor" (page 40). STUDY THIS CHAPTER CAREFULLY. This chapter will describe many of the monitor commands you will need to krak 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, and to make your job as a "krakist" easiest, the last step is to learn the forbidden language, 6502 ASSEMBLY language. Although this is not necessary for a great number of kraking 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. Look at Softalk's top ten list and I bet you 95% of the programs are written in ASSEMBLY language. The obvious reason for this is because ASSEMBLER is fast, and this is VERY important for 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 use to it, and keep good notes of what you learn!

Now your ready for the big time.... kraking 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 disassmble and list code presently in memory. Get use to looking at these disassemblies since you will never have "source" code from protected programs to examine. Therefore be fluent in Apple disassembly. The best way to achieve this is practice, and nothing else will substitute.

Also, be aware of the existing documentation on kraking. Now just about anyone can read "cookbooks" on how to deprotect a particular program that they down loaded from Pirate's Harbor. So you want to take a step further. Don't discard or casually glance through these "cookbooks", but go through them and understand the protection and see how the author choose to krak the particular program. This will prove invaluable by opening your mind to previously used techniques that you can learn from. Try and understand every step the experienced krakist took to deprotect the program, and make careful notes (both mental and on paper) to guide you in your own efforts.

After you have a good understanding of the basic techniques, the next step is to make some small hardware modifications used in kraking tricky programs. These will make your job infinitely easier. So read on and be prepared to learn.

Chapter 2.

Where to Begin. A discussion on the first steps to kraking protected programs.

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 series.

The object to this series is to make the average user more informed about copy protection and how to defeat them. 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:

1) Your original disk fails?

2) You need to modify the protected program for your particular application?

Copy protection makes copies with "FID" or "COPYA" impossible. So 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. I consider this unacceptable.

I will be discussing "kraking" methods, as opposed to copying methods using Locksmith or Nibbles Away. Kraking 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.

The primary objective of this series is to provide you with the knowledge to krak programs for your own use. Of course, using this information for illegal and otherwise perverted uses is frowned upon by myself and this series is written only for your amusement and education (com'on, this is serious, don't laugh...). 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 kraking 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 krakist. There is no substitute for experience, but there are some general guidelines to follow and some real dead giveaway clues to look for.

In addition to finding the starting point, you will have to have some basic tools to krak most programs. Please refer to the chapter entitled "Basic Kraking Tools" for a complete description of what is absolutely minimal in attempting to krak protected programs.

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.

Now that we have categorized copy protection into three main categories, 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 may refer to the particular chapters on kraking that protection type.

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 everyday. 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 kraking 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 game levels, etc.).

Programs that first load a title page, turn off the drive, and then wait for you to press a key and load the program are also 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. 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 kraking of single load programs is finalized into a single (or small number of) files. Then from normal DOS this file can be BRUN. I am sure most of you have seen this type of program.

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) do. The difference is that the publisher/author have modified DOS slightly to read and write to their copy protected disk.

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 nibble copier owners.

Fortunately, kraking modified DOS disks is generally more systematic and sure-fire than the other types of protection. Also, there are many programs already developed to aide you in kraking 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 (either the "]" or "<" prompts). 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 one (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.

Now upon booting a protected disk you here the above sounds, there is a 99 percent change 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 and some examples to this type of protection.

MODIFIED RWTS PROTECTION.

The use of a modified RWTS is really just an off-shoot of the Modified DOS protection scheme. RWTS is a portion of DOS that does the actual reading and writing of particular sectors of a disk. 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 only occupies a portion of DOS, the actual program that is being protected can be larger that if an entire DOS is being used.

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

The method to deprotecting these disk 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 is using a modified RWTS. Basically, if the program does not fit into the other two categories, it is probably a modified RWTS protection. (The only other real alternative is that the disk uses a modified PASCAL operating system, like the PFS and Wizardry series do).

Examples of programs using this kind of protection are games like Zaxxon, Miner 2049'er, Donkey Kong and Jungle Hunt. I have devoted a chapter to this type of protection along with some practical examples.

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

Chapter 3.

Minimal Hardware Necessary for Kraking Protected Programs. Disk Jockey.

IMPORTANT NOTE:

The first thing I should mention is that for doing any serious kraking you will need an Apple II or II+. Apple //e's are fine computers, but for the hardcore krakist, they will simply not do. The reason being is that the //e, Apple uses many "proprietary" chips that are not easily available, thus making any hardware modifications difficult. I'm not saying it can not be done, just its more difficult. For example, changing the Apple's monitor ROM is a considerably most difficult job on the //e than on the II+. Also, since slot zero is unusable by anything other than the built-in RAM card, we can not put a ROM card in its place. The II+ is just easier to modify on the hardware level, so I will be expecting you to have one.

In order to krak most protected program you will have to have some way to break out of the program. The standard way of doing this has been to hit the reset key. But depending on the program on hand, this may not enable you to break out of the program without disturbing a minimal amount of memory.

I am sure you have noticed that after hitting reset (assuming a normal Apple II+) that many things can happen. Usually, the program clears memory and reboots. This is evident by the screen filling up with some single character (the most popular is the inverse "@" character, representing a hex $00) and the disk drive starts up just like you hit "PR#6".

This is rather inconvenient since our objective in kraking is to break out of a program, save memory to a normal DOS disk, and then reload and restart the program. This is very difficult if the program clears memory in the process of breaking out of it. This is where we need some help from the god of hardware.

What we need is the infamous "old style F8 monitior ROM". This F8 monitor ROM is one of those big fat chips that lives inside your Apple, just in front and to the left of the peripheral slots. It should be labeled "ROM-F8". The old style F8 monitor was the monitor ROM that came with the Apple II. It enables you to enter the monitor whenever reset is pushed. (NOTE: If you have a RAM card in slot zero, a program can fool the Apple into looking at the memory in it instead of the F8 monitor ROM, and do funny things accordingly when reset is pushed. For a full discussion of this protection technique, read the chapter on "Kraking Flip-Out".)

Now days, the Apple II+ (and //e) have the new style F8 monitor ROM, called the "Autostart" monitor ROM. This ROM allows the reset key to be programable, or do what ever you want when reset is pushed. For example, with the autostart ROM you can program it to jump to a routine that clears memory and reboots when the reset key is pushed. All that is involved is to change three locations in memory to point to your reset routine, and there is no way for you to stop it from executing when the reset key is pushed, unless you have an old style monitor ROM (or you do some tricks with your RAM card. See the chapter entitled "Kraking with a RAM card").

So to conclude this chapter, I am saying you will need, at minimum:

1) An Apple II or II+.

2) An old style F8 monitor ROM (available at most good Apple repair stores).

But before you run out with your wallet in hand, I suggest you finish reading all chapters in this series for alternative and cheaper ideas. But I am for warning you that you will need this hardware, at the minimum.

Chapter 4.

Deprotecting Single Load programs.

At this point I must assume that you have read the introductory chapters and that you have obtained the minimal hardware necessary for kraking, as described in chapter three (since the hardware reset is an absolute necessity to begin kraking). Now the path divides and I will have to assume that you have determined that the program at hand fits into the "single load protection" category, as described in chapter two. But for review, remember that 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.

Unfortunately (or fortunately,depending how you look at it) 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 crimes we are committing, 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 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.)

In the old days starting address were 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.

So usually the first key you will want to press after the program is loaded is the reset key to reset into the monitor (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. 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.

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 $C050
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 Inspector's Find command and search for "50 C0" (refer to the appendix concerning the ROM card for a further explanation of the Inspector).

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.

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 reset 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 or a BNE 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 Inspector. 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     THIS CLEARS THE KEYBOARD
LDA $C000     THIS GETS A KEYSTROKE
```

```
BPL $XXXX     IF NO KEY, GOTO $XXXX
JMP $START    KEYSTROKE FOUND JMP START
```

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 RE-START
BNE NEXT2     COMPARE TO CTRL R
JMP START     IF EQUAL, GOTO START
CMP #$9B      COMPARE TO ESCAPE
BNE NEXT3     NOT EQUAL, TRY ANOTHER
JMP HALT      IF EQUAL, HALT PROGRAM
CMP .....
```

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 be jumped to by means of a BEQ too, so keep that in mind.

So to test your new found starting address, turn on the hi-res page manually (the game might not do it for you), and type the starting address followed by a "G":

```
*C050       (you will be blind, 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 programmer 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. But do not be discouraged, and 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. This requires a different approach and will also call for some additional hardware. Refer to the chapter on NMI's when you have tried everything and can still not find the starting location.

STEP 2: WHAT PORTION OF MEMORY TO SAVE

Now that you have found the starting address of the protected program, we must determine what portions of memory we should save in our final product. We cannot save all of the 48K 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 $C0 pages that exist. NOTE: There are ways of saving more that $8E pages of memory in a standard bfile, and I address these methods in the chapters entitled "Using Second Stage DOS" and "Memory Packing".

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 get into the monitor and clear memory to zeros. This way, after the program is loaded, we can examine memory and see what is loaded in by the program. The best way to do this is to type:

```
]CALL -151
*800:0
*801<800.BFFFM
```

This takes whatever we put in location $800 and copies it to $801 to $BFFF. NOTE: if after doing this your Apple hangs, just hit reset to recover.

Now boot your protected program by typing:

```
*C600G
  or
*6<CTRL P>
```

(assumes disk controller card in slot 6).

After the program is loaded, reset into the monitor and get the inspector up and search through memory for blank pages. 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 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 is turn on the hi-res page and use what is already there, or does it re-drawn 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). Type:

```
*2000:0
*2001<2000.3FFDM
```

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 a 48K slave disk (I will ignore anything you say about a tape recorder!).

A 48K slave disk does not disturb memory from $900 to $95FF when it is booted. The best way to make a 48K slave disk is to boot any normal DOS 3.3 disk (fast DOS preferred), and type:

```
]FP
]INIT HELLO
```

The disk created will be a 48K slave disk.

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 decmal locations). If a program is bigger than $8D pages of memory, you will have to save the memory portions in steps, or use some additional hardware as described in the chapter about NMI's.

The best way to explain this is to use an example. Say you have found the starting address of a program and have found it lives from $800 to $2000 and $6000 to $BFFF. This means that you must relocate $800 to $8FF and $9600 to $BFFF, to clear the way for a slave disk boot. Since $2000 to $5FFF is unused, we can use that memory portion to temporarily store $800-8FF and $9600-BFFF.

We can let the Apple do some of the work to determine if we have enough room. type:

```
*C0-96
```

and the Apple will return a $2A. Now subtract $60-20 by typing:

```
*60-20
```

and $40 will be returned. We obviously have enough room since we only need $2B pages ($2A+01). Now use the monitor move commands to move memory around:

```
*2000<9600.BFFFM
*4A00<800.8FFM
```

Now boot your 48K slave disk and save the memory portions to disk:

```
]BSAVE ^09-20,A$900,L$1700
```

```
]BSAVE ^96-C0,A$2000,L$2A00
]BSAVE ^08,A$4A00,L$100
]BSAVE ^60-96,A$6000,L$3600
```

Once again, use the monitor's addition and subtraction capabilities to figure out the lengths of the files if you are not good at hex math.

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.

You can not simply re-load all the parts of memory since some will overwrite DOS, so we must load the memory portions in between $800 and $95FF and move the $9600 to $BFFF region back up to where is belongs, then jump to the beginning of the program.

This is accomplished through the use of memory moves. A short program will move portions of memory from one part of memory to another. We can use the program MEMORY MOVE WRITER to do this work for us (there is additional doc on this program in one of the appendices).

Remember that the unused portion of memory from $2000 to $5FFF is partially taken up from $2000 to $4A00 (we can move $4A00-$4AFF back to $800-8FF). So we can put our memory move program at $4A00.

To do this, BRUN the file called MEMORY MOVE WRITER and choose a running location of $4A00. We want 1 move which will move $2000 to $4A00 to $9600. Now we select the viewing pages we want (hi-res, or text) and then enter the starting location of the program. A file will be saved to disk containing our memory move program.

Now to put it all together. Bload the files into the appropriate place by typing:

```
]BLOAD ^08,A$800
]BLOAD ^09-20,A$900
]BLOAD ^96-C0,A$2000
]BLOAD MEMORY MOVE$4A00,A$4A00
]BLOAD ^60-96,A$6000
```

Now enter the monitor and make a couple changes so the file will run when BRUN:

```
]CALL -151
*7FD:4C 00 4A   (this jumps to $4A00
                 when the file is BRUN)
*A964:FF        (enables us to save
                 big 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 file to your disk by typing:

```
*BSAVE FILE,A$7FD,L$8E03
```

You can determine the length parameter by typing:

```
*96-08
```

and $8E 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 $8E03.

Brunning this file should restart the program and all should be fine. Note that we have saved some unnecessary memory in the file, memory from $4B00 to $5FFF. We could pack the file move and use more memory moves to make the file smaller, but I will leave that to you.

FINAL NOTES:

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

Also, if your program requires saving "volatile" memory locations (text page, etc.), refer to the chapter entitled NMI's for help with that subject.

Chapter 5.

Modified DOS Protection.

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. Before being able to krak disks using this protection, it is best to understand a normal DOS disks' structure. Then we can evaluate how the normal structure is modified.

Each normal DOS disk has 35 tracks (labeled 0 thru 34) and 16 sectors (labeled 0 thru 15), where each sector represents one page (256 bytes) of data. Each sector consists of two separate parts: an address field and a data field. First let's discuss the address field.

If you read a disk with a "nibble read" routine (found in The Inspector, Nibbles Away and Locksmith), 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
   !    (1)    !   (2)  ! (3) ! (4) !

    AA AA FF FE DE AA EB FF FF FF FF...
   ! (5) ! (6) !  (7)   !    (8)

    D5 AA AD -342 bytes- XX DE AA EB
   !   (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 prologue address 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 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".

Also, 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. This involves 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 played with is very easy. Just enter the monitor and change byte $B942 from $38 to $18. What this does is to "clear the carry" instead of "setting the carry" when a disk error is encountered (DOS used the carry bit as a flag for errors). Just type:

```
]CALL-151
*B942:18
*9DBFG
```

THIS IS PROBABLY THE MOST IMPORTANT MODIFICATION YOU CAN LEARN TO MAKE WHEN TRYING TO DEPROTECT A MODIFIED DOS PROGRAM!

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

Also very common is to change the address or data field prologue bytes. This will immediately choke up any normal DOS copy program and even confuses Nibbles Away and Locksmith 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 can not find where the data starts.

The way to find out if these bytes have been changed is to do a nibble read (using the Inspector, Nibbles Away or Locksmith) 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 COPYA. All you have to do is to add three lines to COPYA. First here is a table of locations that are to be changed 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 ($B96A):150 ($EB

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)
```

The change we want to make to COPYA is to swap the address or data field bytes that are different on the read and the writes. So we just poke the modified bytes before the read, and poke the normal bytes before the write. This is best illustrated with an example. Suppose a disk has modified address epilogue bytes of D5 AA AD (instead of the normal D5 AA 96) and modified data prologue bytes of D5 AA DA (instead of the normal D5 AA AD). So we would need to add these three lines to COPYA by typing:

```
]RUN COPYA
(after program is loaded and waiting
for you to select drives and slots):
CTRL C
]198 POKE 47446,173: POKE 47356,218
]248 POKE 47466,150: POKE 47356,173
]258 POKE 47466,150: POKE 47356,173
]70
]RUN
```

The first added line changes the address and data prologue bytes to the modified format just before reading the original disk. The last two lines added change them back to normal DOS 3.3 just before initializing and writing to the destination disk (respectively). Of course I converted the hex values of the bytes to decimal and only poked the different ones.

Instead of doing the poking by hand, I have written a program called "ADVANCED COPYA" that allows you to input any changes to the address or data markers (in hex or decimal), along with other parameters you may find handy.

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 poke 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 is doing so right to the disk so it is permanent.

It should be noted that some programs change the address and/or data marker bytes as the program runs. Examples of this is 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.

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. This is a much easier and more general method of deprotection than using "ADVANCED COPYA", but is also not as flexible. It is best to become familiar with both programs and how to use them.

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

To summarize this chapter, we had 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 suggest going through the example kraks of programs. They will provide some real-life situations that are very typical.

Chapter 6.

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.

RWTS is roughly the upper third of DOS $B700-BFFF) and does the actual loading and writing of separate sectors. 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 is 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.

So the first thing you will have to do is to get the data from the protected disk to a normal DOS 3.3 format. The first thing I ALWAYS try is to defeat the DOS error checking routine at $B942 and try and copy the disk with COPYA. 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. Usually an error can be prevented if modified epilogue bytes are used. To defeat the DOS error checking, type:

```
]CALL-151
*B942:18
*RUN COPYA
```

Many times you will only have to read a portion of the protected disk, because it does not use all tracks. Use ADVANCED COPYA or COPYB to transfer the data to a normal DOS disk in this case, or where the above mod to DOS did not allow you to copy the disk (ADVANCED COPYA and COPYB are discussed in the previous chapter, and in appendix A). Also, it would help if you knew what the address/data markers are, if different than normal DOS.

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, so 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 BYTE1-WRITE
$B858	AA	DATA PROLOGUE BYTE2-WRITE
$B85D	AD	DATA PROLOGUE BYTE3-WRITE
$B89E	DE	DATA EPILOGUE BYTE 1-READ
$B8A3	AA	DATA EPILOGUE BYTE 2-READ
$B8A8	EB	DATA EPILOGYE BYTE 1-WRITE
$B8AC	FF	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
$B92A	D9 00	LOCATION CHECKSUM COMPARE
$B935	DE	DATA EPILOGUE BYTE 1-READ
$B93F	AA	DATA EPILOGUE BYTE 2-READ
$B942	38	SET CARRY FOR I/O ERROR
$B955	D5	ADDR PROLOGUE BYTE 1-READ
$B95F	AA	ADDR PROLOGUE BYTE 2-READ
$B96A	96	ADDR PROLOGUE BYTE 3-READ
$B991	DE	ADDR EPILOGUE BYTE 1
$B99B	AA	ADDR EPILOGYE BYTE 2
$BC7A	D5	ADDR PROLOGUE BYTE1-WRITE
$BC7F	AA	ADDR PROLOGUE BYTE2-WRITE
$BC84	96	ADDR PROLOGUE BYTE3-WRITE
$BCAE	DE	ADDR EPILOGUE BYTE1-WRITE
$BCB3	AA	ADDR EPILOGUE BYTE2-WRITE
$BCB8	EB	ADDR EPILOGUE BYTE3-WRITE

If any are different, locate the different byte sequence on the COPYA version of the disk using a good disk scanning program (the Inspector, CIA, etc.) and change the locations back to normal with a sector editor.

Many times, this is all that is necessary, but not always. Maybe there was a nibble count that will prevent the COPYA copy from running. Finding this could be the trickiest part. Search through the program looking for any JMP's or JRS's to the DOS area. Beyond this I can not give much help. But after you find it, locate the routine on the disk and defeat it. You can do this by putting a $60 at the beginning of the nibble count subroutine (Return from subroutine), or by "noping" the call (no operation) to the subroutine with three "$EA"s.

For some real life examples, refer to the files "Zaxxon" and "Oil's Well". These give good examples of modified address/data markers and defeating nibble counts in modified RWTS protection.

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 9E C0    LDA   $C09E,Y
0353-   C9 D5       CMP   #$D5
0355-   D0 F8       BNE   $0350
0357-   B9 9E C0    LDA   $C09E,Y
035A-   C9 AA       CMP   #$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 be familiar with direct calls to RWTS. If you can recognize these you can usually find from where the data is being loaded from. Here the parameter list for using DOS from ASSEMBLY language:

```
$B7EA: DRIVE NUMBER TO USE
$B7EB: VOLUME NUMBER ($00=ANYTHING)
$B7EC: TRACK NUMBER TO READ
$B7ED: SECTOR NUMBER TO READ
$B7F0: LO-BYTE OF BUFFER TO READ/WRITE
$B7F1: HI-BYTE OF BUFFER TO READ/WRITE
$B7F3: PARTIAL SECTOR READ(0=WHOLE SCT)
$B7F4: COMMAND CODE(0=SEEK,1=RD,2=WRT)
$B7F5: ERROR CODE (VALID IF CARRY SET)
$B7B5: SUBROUTINE FOR ACTUAL READ/WRT.
$BD00: ALSO SUBROUTINE FOR READ/WRITE
```

Here is a sample program that uses these parameters:

```
9000-   A9 04       LDA   #$04 ;track 4
9002-   8D EC B7    STA   $B7EC;
9005-   A9 0F       LDA   #$0F ;sct F
9007-   8D ED B7    STA   $B7ED;
900A-   A9 00       LDA   #$00 ;
900C-   8D EB B7    STA   $B7EB;vol #
900F-   8D F0 B7    STA   $B7F0;page
9012-   A9 4F       LDA   #$4F ;hi-page
9014-   8D F1 B7    STA   $B7F1;
9017-   A9 01       LDA   #$01 ;read
9019-   8D F4 B7    STA   $B7F4;
901C-   A0 E8       LDY   #$E8 ;ready
901E-   A9 B7       LDA   #$B7 ;to rd
9020-   20 B5 B7    JSR   $B7B5;read
9023-   CE ED B7    DEC   $B7ED;dec sct
9026-   CE F1 B7    DEC   $B7F1;dec pg.
9029-   AD ED B7    LDA   $B7ED;compare
902C-   C9 FF       CMP   #$FF ;sct.
902E-   D0 EC       BNE   $901C;go back
9030-   A9 0F       LDA   #$0F ;else
9032-   8D ED B7    STA   $B7ED;sct $0F
9035-   CE EC B7    DEC   $B7EC;dec trk
9038-   AD EC B7    LDA   $B7EC;compare
903B-   C9 01       CMP   #$01 ;track
903D-   D0 DD       BNE   $901C;go back
903F-   60          RTS        ;return
```

Try and figure out what this does...This reads track 4, sector F down to track 2, sector 0 into $4F00 to $2000. This would be a perfect example for reading a hi-res picture in using RWTS. Be able to recognize these routines. The dead giveaway is the JSR $B7B5. All the parameters could have been loaded through other location, such as zero page location, then re-loaded into RWTS. But the JSR $B7B5 gives it all away.

Also notice how the routine read "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").

I hope this has provided some useful information in deprotecting modified RWTS programs. For further information on RWTS, refer to the chapter on using second stage DOS (RWTS). This provides information on how to use RWTS from your own programs.

Chapter 7.

Using Second Stage DOS for your own programs.

Sometimes you will find your self with a single load program that is too big for a file under normal DOS. This usually means the program uses more that $8E pages of memory. In cases like these, you can often load in the file using second stage DOS, or otherwise know as RWTS.

Another application for using second stage DOS is when you have a multi-disk access program that uses a fairly standard disk structure. Lets say you know the program loads in, and then runs for a while. Then a new level is loaded in from track $01. You could krak the program into a file, transfer track $01 to a normal format disk, and put the file on some other tracks than track $01. But DOS uses tracks $0 to $02, so how do you load the file?.. You can use second stage DOS (that lives on track $0, sectors 0 to $09 only) to load the file from track 2 and up, and this could read the levels from track1.

Being able to use second stage DOS is certainly handy, and if you write your own programs, a great way to make them load "professionally", without the dumb BASIC prompt appearing during the boot. First let's see why it's called "second stage DOS".

DOS loads in from a disk in three parts, labeled stage (or boot) 0 through 2. The first stage of DOS boot (stage 0) is loaded from track 0, sector 0, by the code in your disk controller card. This one page of memory gets loaded into $800-$8FF.

The second stage of the DOS boot (stage 1) is loaded in from track 0, sector 0 to sector 9 into $B600-$BFFF. This is loaded by the code at $800-$8FF. The code loaded into $B600 and up is RWTS.

The third stage of the DOS boot (stage 2) is loaded from track 0, sector C to track 2, sector 4, and is the rest of DOS ($9D00 to $B5FF). This is loaded by the code at $B600-$BFFF. But we can fool DOS into loading our program instead of loading in the rest of DOS!

To summarize, here are the steps:

boot 0: loads trk 0, sct 0 into $800
boot 1: loads trk 0, sct 0-9 into $B600
Boot 2: loads our program into ???

Here is how we do it:

1) Boot your DOS 3.3 System Master (you CAN NOT use a fast DOS disk here, but that won't matter cuz it will load real fast anyway).

2) Initialize a blank disk by typing:

```
]INIT HELLO
```

3) Run a sector editor on this initialized disk and change track 0, sector 0, byte $4A to $4C 80 08. Don't forget to write the sector back out.

4) Read track 0, sector 0 into $800 and write this program at $880:

```
0880- A9 XX       LDA #$XX :# of scts
0882- 8D E0 B7    STA $B7E0:
0885- A9 XX       LDA #$XX :starting trk
0887- 8D 15 B7    STA $B715
088A- A9 XX       LDA #$XX :starting sct
088C- 8D 1A B7    STA $B71A
088F- A9 XX       LDA #$XX :start pg+1
0891- 8D E7 B7    STA $B7E7
0894- 4C 00 B7    JMP $B700
```

NOTES:

$881: enter the number of pages (or sectors, they're the same, remember?) you want to read in.

$886: enter the starting track to read from. This will read in descending order, so enter the HI-TRACK.

$88B: enter the starting sector to read from. This will read in descending order, so enter the HI-SECTOR.

$890: enter the HI-PAGE PLUS 1 you want to load into. This will be the first page read into, and it will descend from there. I.E. If you want to start loading at $AFFF, the start page is $B0.

5) Write the sector back out.

6) Read in track 0, sector 1 and change byte $47 to $4C XX YY where XX YY is the address you want to jump to after the program is loaded, in lo-hi order (just like normal). Don't forget to write the sector back out.

Now you're all done! You can use track 0, sector A through track 22, sector F for any data. Transfer your program or other data using the Inspector or some other means onto the tracks and sectors you specified above.

REMEMBER, SECOND STAGE DOS WILL READ YOU PROGRAM FROM THE HIGH TRACK AND SECTOR DOWN, AND FROM THE HIGH PAGE DOWN (I.E. TRACK 9, SECTOR F INTO $8F00-$8FFF, TRACK 9, SECTOR E INTO $8E00-$8EFF, ETC.).

It should be noted that many protection schemes use this exact format to load in their programs. They just modify RWTS for their protection, as described in chapter six, modified RWTS protection. An example of this is many of Datamost's programs, including "Roundabout".

Chapter 8.

The Art of Memory Packing.

Many times when kraking a single load protection scheme and finding the starting address and what memory the program occupies, the program is just too big to fit into a normal DOS file. After examining memory, you find a large section of memory that all the bytes are the same, but is necessary for the program to run. You could use second stage DOS to read the file in, but it just seems to be a waste to have all that repetitive data on a disk.

Or maybe you have just kraked a great new game, and you have just put the finishing touches on the file, complete with a hi-res banner displaying your name and the program's name (don't break your arm patting yourself on the back). Just for reassurance, you want to make that hi-res picture so some idiot can't erase or change it (at least not easily).

Both of these cases warrant using a memory packer. Memory packing is taking some repetitive bytes and storing them into some smaller format in a file. Then, when the program is reloaded, expanding them back out to their normal size and positions. Hi-res pages and shape tables tend to have a great deal of repetitive bytes, and can sometimes be reduced to a quarter of their original size using a memory packer.

The way memory packing works is simple. Typically, a memory packer finds a byte that repeats 4 or more times (3 times is the break-even point) and stores the repetition as 3 bytes: the first byte is a "flag" byte, signifying the next 2 bytes are a packed sequence. The second byte is the number of times the byte repeats itself, and the third byte is the actual byte that is repeated.

Of course, doing this by hand would be stupid at best, so we can write a program that searches a specified range of memory for repeating bytes, and packs the range (many "picture packers" do this, but only for one of the two hi-res pages). It would be optimal to pack over the original memory, as a convenience to us (the programmer) to avoid moving surrounding memory around too much.

But before we can pack any memory, we must have a flag byte. A flag byte is a byte that signifies the next 2 bytes are a packed sequence. OF COURSE, THIS FLAG BYTE CAN NOT OCCUR ANY WHERE ELSE IN THE MEMORY RANGE YOU ARE PACKING.

So to pack a range of memory, we need to be able to do 3 things:

1) Find a Flag byte.

2) Pack the memory using a Flag byte.

3) Unpack the memory.

Of course, I have provided you with some short routines to do these three things. (NOTE: all these routines are written by "KRAKOWICZ", and are public domain). The first program is called "SHOWBIN", and finds our flag byte.

To use SHOWBIN:

1) load the memory range you want to pack (any range from $800 to $95FF).

2) Put the low page in location $00, the high page in $01.

3) BRUN SHOWBIN.

The first display will be the bytes that are never used in that memory range. Now hit RETURN, and the bytes used once are displayed. Hit RETURN again, and the bytes used twice are displayed, and so on. To exit, hit ESCAPE.

Make a note of one of the bytes that are never used in that memory range (if there is NO bytes that are never used, you will have to make the memory range you are packing smaller, sorry). Now its time to run PACKER.

To PACK the memory range, do the following:

1) Load location $05 with the flag byte, location $06 with the low page, and $07 with the hi page.

2) BRUN PACKER

After a second, you will get your prompt back. So now type:

```
*00.01
```

The returned two bytes are the address that the packed data starts at (in backassward order, of course). It ends at the last location of the hi-page specified.

An example is needed to clarify: Suppose you want to pack memory from $4000 to $95FF. To do this, here are the steps:

1) BLOAD the memory you want packed ($4000 to $95FF).

2) type:

```
]CALL-151
*00:40 95
```

(remember, $40 is the lo page, $95 is the hi-page).

3) type:

```
*BRUN SHOWBIN
```

4) Write down one of the unused bytes and hit ESCAPE to exit (let's say $91 is the unused byte that will be our flag byte).

5) type:

```
*05:91 40 95
*BRUN PACKER
*00.01
```

6) Write down the two bytes returned from locations $00 and $01 (Let's say they are $B3 and $71, respectively. This means we have packed $4000-$95FF into $71B3 to $95FF).

Now to reconstruct $4000-$95FF from $71B3-$95FF, we need to load locations $00 and $01 with $B3 and $71 (just like PACKER gave them to us), and call the UNPACKER routine.

UNPACKER is less than $50 bytes long and is relocatable (can run in any memory range). UNPACKER will unpack memory over itself, saving us a lot of work and programming. So to unpack the $71B3 to $95FF:

1) type:

```
*00:B3 71
*BRUN UNPACKER
```

That's it! Under program control, it would look like this:

```
LDA #$B3
STA $00
LDA #$71
STA $01
JSR UNPACKER
```

For more explanations on memory packing, refer to the file "SINGLE FILE ROUNDABOUT", where I show how I krak and pack a single load protection game from Datamost called "Roundabout".

Chapter 9.

Introduction to the non-maskable interrupt (NMI). JOCKEY.

INTRODUCTION:

Assuming that you have read the first several chapters of this series, you now should possess some basic information regarding the architecture of the Apple computer. Using this basic information you will go quite far down "memory lane" in your kraking efforts, but it doesn't stop there. Now we need to talk about some basic hardware you will need to make your job easier. What we will be discussing is the use of resets and "NMIs" in the art of kraking.

INTRODUCING THE NMI:

NMI is an acronym for NON MASKABLE INTERRUPT, and as the name implies, it can not be prevented (or masked) on the Apple. The NMI is the basis behind most of the "copy cards" on the market, such as the Wildcard or Replay cards. The NMI allows us to interrupt a program, and to restart it with minimal effort. Obviously this is of extreme importance to the krakist, who wants to interrupt a programs, save memory to a normal DOS disk, and restart the program upon BRUNing the file.

To use an NMI you can simply crossed pin 26 (ground) and pin 29 (NMI) of any one of the peripheral slots. You can do this with a 100 ohm resistor. This will execute an NMI.

Unfortunately, this is less than ideal since when you try to do this, you will probably execute 20 or so NMI's. This is because it happens so fast, that an NMI will interrupt an NMI, and so on for many, many times. This will put much garbage onto the stack (page one). Using a switch for this chore doesn't help since the switch actually slams (or bounces) against itself many times causing the same problem. To solve this we need to make a "de-bounced" NMI switch. This will constitute about $8 to $20 of capital resources (depending on your parts supplier), and a soldering iron. This is considerably less expensive than a store bought NMI board, but will lack some of the features the commercial ones have. A full discussion of how to make an NMI board is in the appendix "Making a NMI Board".

Assuming you have made your NMI card, I will now tell you more about how it works and its uses. If you don't fully understand the workings of the NMI, don't worry about it. Just try and follow along.

When you push the NMI switch, the 6502 processor will push the present value of the program counter on the stack along with the processor status word. Then it will jump to what ever locations are pointed to by $FFFA and $FFFB. So to restart an interrupted program, we only need to restore the registers (x, y, accum), the lower pages of memory, and the stack pointer, and do a "RTI" (return for interrupt) instruction. I have written a small program which does this all for you and works in conjunction with the NMI board and a modified F8 ROM, which I am about to describe.

Now remember our old F8 monitor ROM? Well these two locations live in the monitor ROM. It would be nice if we could change these location and after an NMI is executed, run a small program to that will save the registers, the stack pointer, and the lower pages of memory. Now this leads us back to our old friend, the F8 monitor ROM.

This is indeed what we need to do. The best thing would be to execute an NMI, and then jump to a routine that moves the lower 16K of memory into a RAM card. Then we could boot a 16K slave disk (which would only disturb the lower 16K of memory), and save all of memory to a disk. After we have saved all of memory, we could reconstruct our program and re-start, or do a "return from interrupt".

Of course to do this we need to change some of the code in the F8 monitor ROM. We can not do this directly to the F8 chip that comes with your Apple since it is Read Only Memory. But we can move the code in the ROM down to RAM, put our routines in, and burn a new "2716 EPROM". The 2716 EPROM will replace the ROM, and will have our new kraking routines in it.

Now you ask, "how do I burn a 2716 EPROM?". Well, if you don't have access to an EPROM programmer, you can take your modified F8 code (saved to a disk) to a local computer store and they should be able to burn you one for a nominal fee.

Refer to the appendix entitled "Making a F8 Krak ROM" for an explanation of how to create the code for the new EPROM and how to plug it in after it is burnt.

Lastly, we need to make a 16K slave disk and to use the program to save all of memory to a disk. To get the program type it in or download it from someone. To create a 16K slave disk, do the following: (NOTE: this only applies to the Apple II or II+)

1) Turn off your computer.

2) Open the lid, and look for the 3 rows of chips that have a white line boarder around them. These are the 48K of RAM in your Apple II+.

3) Remove any one chip from each of the two rows of RAM furthest away from the keyboard.

4) Turn the computer on and boot your DOS 3.3 System Master.

5) Put a blank disk in the drive and

type:

```
]INIT RAM 48K SAVER
```

6) When this is complete, turn the computer off and replace the two chips.

7) Run a sector editor and change the following sectors of the 16K slave disk:

TRK	SECTOR	BYTE	FROM	TO
$00	$01	$48	$03	$00
$00	$0D	$42	$06	$34

8) Write the sector back out to your 16K slave disk.

9) Delete the Hello program on the disk by typing:

```
]DELETE RAM 48K SAVER
```

Now download the "RAM 48K SAVER" file and save it to your 16K slave disk. Also down load the file "MEMORY MOVE WRITER". Save these to your 16K slave disk also, and then write protect it.

The program "RAM 48K SAVER" will save all 48k of memory to a disk and also create a restart program that will do the "return from interrupt". The "MEMORY MOVE WRITER" program will enable you to reconstruct memory back to where it belongs after you load it from your normal DOS 3.3 disk.

In the appendix "Using your NMI board", I will discuss how to use this hardware and software in a real-life application.

Chapter10.

Protected Applesoft Basic programs.

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 against 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 RUN their BASIC program. I will describe how to beat all these protection schemes, assuming you have an old style F8 monitor ROM.

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. Further more, 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 is 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 give away 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 reset into the monitor when the program is running.

Now you can try to enter the immediate BASIC mode by typing:

```
*3D0G
```

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

Assuming that didn't work, reload the program and reset into the monitor again. The next thing is to try typing 9D84G or 9DBFG.

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. If anything you type starts the program running again, the protection has changed the RUN flag at $D6. So reset into the monitor again.

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.

For debugging efforts, the RUN flag can get 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 a register with $80 or greater and storing it at $D6.

Now if 3D0G, 9D84G or 9DBFG did not produce a BASIC prompt, then the DOS being used is more elaborate. So re-load the program and reset into the monitor after it is running.

Now comes the final steps in trying to examine a BASIC program. If you are using a ROM card in slot zero with an old style F8 monitor ROM to reset into the monitor, turn on the mother board ROMs and turn off the ROM card INTEGER ROMs by typing:

```
*C081
```

Now reset the RUN flag to normal, just to be sure. Type:

```
*D6:00
```

Finally, enter APPLESOFT the sure fire way by typing:

```
*<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 it is a DOS command. 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. Very different, eh!

Well, back at the ranch, if you want to save your APPLESOFT program to a normal DOS disk, do these steps:

1) Reset into the monitor after the program is running.

2) If you are using a ROM card in slot zero, Type:

```
*C081
```

3) 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 somewhere.

5) 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 spaces.

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 (such as Demuffin Plus or CopyB), but this is a quick and dirty method that will 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.

I hope this will help you learn more about the protected programs you own that are written in APPLESOFT.

Appendix A.

Kraking modified DOS's with COPYB

INTRODUCTION:

There are probably hundreds of ways to protect a program from being copied. But generally speaking, protection falls under two categories: protect the actual program (by various means), or protect a disk full of programs with some sort of DOS modification. DOS modifications are the most common since they are the easiest to deal with (from the publisher's point of view). DOS modifications are also the least successful of protection, since someone always seems to find a way to copy all the files onto a normal DOS disk, eluding all the protection. 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. While this is a powerful utility, it only works with programs that are based on DOS file structures and that have a catalog track.

INTRODUCING COPYB:

COPYB (originally written by "Krakowicz") is a highly modified version of COPYA which converts a protected disk that uses 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 parameters for COPYB:

LOCATION			NORMALLY		
HEX	DEC	DESCRIPTION	HEX	DEC	NT
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	TRK TO STOP READING+1	23	35	(4)
35F	863	TRK TO STOP READING+1	23	35	(4)

NOTES (denoted above as "NT"):

1) This is the first track that COPYB starts reading at. 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 parameter 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) Read in track 0, sector 1 through track 0 sector 9 of the protected disk into memory $8000 to $88FF with a sector editor such as 'THE INSPECTOR'. Then run COPYB.

ENTERING THE PARAMETERS AND RUNNING COPYB:

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 you have entered 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 like 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. I am sure you will find COPYB an excellent utility to have.

Appendix B.

Making a F8 Krak ROM.

In this section I will describe how to make the code for the modified F8 monitor ROM that you will find extremely useful in kraking.

The EPROM will act like a old style F8 monitor ROM with regards to resets. What I mean is that hitting reset will cause you to jump into the monitor.

The EPROM will also have a special function when an NMI is encountered. Upon NMI, this ROM will push the accumulator, the x-register,the y-register and location $00 onto the stack. The stack pointer will then be saved at location $00.

Next, the EPROM will move $00 to $4000 into a RAM card in slot zero. This clears the way for a 16K slave disk boot. Here is the code and an explanation of how it works (in 80 column format):

```
FCC9-   48          PHA            PUSH ACCUM ONTO THE STACK.
FCCA-   8A          TXA            TRANSFER X-REG TO ACCUM.
FCCB-   48          PHA            PUSH (X) ACCUM ONTO THE STACK.
FCCC-   98          TYA            TRANSFER Y-REG TO ACCUM.
FCCD-   48          PHA            PUSH (Y) ACCUM ONTO THE STACK.
FCCE-   A5 00       LDA   $00      LOAD ACCUM WITH $00.
FCD0-   48          PHA            PUSH ($00) ACCUM ONTO THE STACK.
FCD1-   BA          TSX            TRANSFER STACK POINTER TO X-REG.
FCD2-   86 00       STX   $00      STORE STACK POINTER AT $00.
FCD4-   AD 81 C0    LDA   $C081    ENABLE WRITE TO RAM BANK 1.
FCD7-   AD 81 C0    LDA   $C081    (MUST ACCESS TWICE).
FCDA-   A0 00       LDY   #$00     -----------------------------------
FCDC-   B9 00 00    LDA   $0000,Y MOVE $00 TO $FF INTO RAM CARD SO WE
FCDF-   99 00 D0    STA   $D000,Y CAN USE ZERO PAGE FOR REST OF MOVE.
FCE2-   C8          INY
FCE3-   D0 F7       BNE   $FCDC    -----------------------------------
FCE5-   84 00       STY   $00      MOVE $100-$2FFF INTO BANK 1
FCE7-   84 02       STY   $02      OF THE RAM CARD.
FCE9-   A9 01       LDA   #$01
FCEB-   85 01       STA   $01
FCED-   A9 D1       LDA   #$D1
FCEF-   85 03       STA   $03
FCF1-   B1 00       LDA   ($00),Y
FCF3-   91 02       STA   ($02),Y
FCF5-   C8          INY
FCF6-   D0 F9       BNE   $FCF1
FCF8-   E6 03       INC   $03
FCFA-   E6 01       INC   $01
FCFC-   A5 01       LDA   $01
FCFE-   C9 30       CMP   #$30
FD00-   D0 EF       BNE   $FCF1    -----------------------------------
FD02-   4C CD FE    JMP   $FECD    RAN OUT OF ROOM HERE, JMP TO $FECD.
.
```

```
.
.
FECD-   A9 D0       LDA   #$D0    RESET MOVE ROUTINE POINTERS.
FECF-   85 03       STA   $03
FED1-   AD 89 C0    LDA   $C089   ENABLE BANK 2 OF RAM CARD.
FED4-   AD 89 C0    LDA   $C089   (MUST ACCESS TWICE).
FED7-   B1 00       LDA   ($00),Y ----------------------------------
FED9-   91 02       STA   ($02),Y MOVE $3000-$3FFF INTO BANK 2
FEDB-   C8          INY           OF THE RAM CARD.
FEDC-   D0 F9       BNE   $FED7
FEDE-   E6 03       INC   $03
FEE0-   E6 01       INC   $01
FEE2-   A5 01       LDA   $01
FEE4-   C9 40       CMP   #$40
FEE6-   D0 EF       BNE   $FED7   ----------------------------------
FEE8-   AD 82 C0    LDA   $C082   TURN ON MOTHERBOARD RAM AND WRITE
FEEB-   AD 8A C0    LDA   $C08A   PROTECT BANKS 1&2 OF RAM CARD.
FEEE-   4C FD FE    JMP   $FEFD   RAN OUT OF ROOM, JUMP TO $FEFD.
.
.
.
FEFD-   A2 1C       LDX   #$1C    THIS SUBROUTINE OUPUTS THE
FEFF-   BD 0B FF    LDA   $FF0B,X MESSAGE "RAM CARD LOADED WITH
FF02-   9D D6 07    STA   $07D6,X $00-3FFF" AT THE BOTTOM OF
FF05-   CA          DEX           THE TEXT SCREEN.
FF06-   10 F7       BPL   $FEFF   ----------------------------------
FF08-   4C 59 FF    JMP   $FF59   ALL DONE, EXIT THRU NORMAL RESET.
```

To create this EPROM file, here are the steps:

1) Boot a normal DOS disk and enter the monitor by typing:

```
]CALL -151
```

2) Move your autostart F8 monitor ROM code down into RAM by typing:

```
*4800<F800.FFFFM
```

3) Now change the code as follows:

```
*4CC9:48 8A 48 98 48 A5 00
*4CD0:48 BA 86 00 AD 81 C0 AD
*4CD8:81 C0 A0 00 B9 00 00 99
*4CE0:00 D0 C8 D0 F7 84 00 84
*4CE8:02 A9 01 85 01 A9 D1 85
*4CF0:03 B1 00 91 02 C8 D0 F9
*4CF8:E6 03 E6 01 A5 01 C9 30
*4D00:D0 EF 4C CD FE
*4ECD:A9 D0 85 03 AD 89 C0 AD 89 C0 B1
*4ED8:00 91 02 C8 D0 F9 E6 03
*4EE0:E6 01 A5 01 C9 40 D0 EF
*4EE8:AD 82 C0 AD 8A C0 4C FD FE
*4EFD:A2 1C BD 0B FF 9D D6 07 CA 10 F7
```

```
*4F08:4C 59 FF
*4F0B:52 41 4D 60 43
*4F10:41 52 44 60 4C 4F 41 44
*4F18:45 44 60 57 49 54 48 60
*4F20:64 70 70 6D 73 46 46 46
*4FFA:C9 FC 59 FF
```

4) Now save the file to a disk by typing:

```
*BSAVE F8 SAVE RAM EPROM,A$4800,L$800
```

5) Finally, burn the 2716 EPROM with this code or have someone do it for you.

Now to use your new 2716 EPROM, you must make these changes directly to the chip itself (not advisable), or to a jumper socket which your new chip will plug into, and then which will be plugged into your motherboard.

You need a 24 pin low-profile socket (not wire-wrap!, they will destroy your motherboard sockets!). These are available from radio shack (part number 276-1989) or the such. Now with the socket up-side-down and the pins looking you in the face, it should look like this:

```
---------------------------------------
!  13 14 15 16 17 18 19 20 21 22 23 24!
! ./ ./ ./ ./ ./ ./ ./ ./ ./ ./ ./ ./ !
!                                     !
!                                     !
!                                    -
!                          (notch)->!
!                                    -!
!                                     !
! .  .  .  .  .  .  .  .  .  .  .  .  !
!/  /  /  /  /  /  /  /  /  /  /  /   !
!12 11 10 9  8  7  6  5  4  3  2  1   !
---------------------------------------
```

Now your soldering skills come in handy! Using some short, hi-gauge wire (wire-wrap is preferable, but anything in the 26-30 gauge will work), solder a piece between pins 21 and 24, and solder a piece between pins 12 and 18. Be extremely careful not to short out the wire or to cross solder any pins! Also, try and solder as close to the base of the socket as possible, since you have to cut off pins 18 and 21 after you have finished soldering them. Now cut of pins 18 and 21 as close to the base of the socket without cutting the freshly soldered wires! Remember, pins 18 and 21 should be short enough so that they will not touch the socket you will be plugging this one into. The socket should now look like this:

```
---------------------------------------
!  13 14 15 16 17 18 19 20 21 22 23 24!
! ./ ./ ./ ./ ./ ./ ./ ./ ./ ./ ./ ./ !
!                x        x        / !
!               /        /        /  !
!              /        /--------/  -
!    /--------/                    !
!   /                               -!
!  /                                 !
! /  .  .  .  .  .  .  .  .  .  .  . !
!/  /  /  /  /  /  /  /  /  /  /  /  !
!12 11 10 9  8  7  6  5  4  3  2  1   !
--------------------------------------
```

Double check your soldering and the connections (and notice that pin 18 and 21 are cut off!). Now carefully remove the ROM labelled F8 (it is the socket farthest on the left that has 24 pins as you face the keyboard) and plug this jumper socket into the motherboard. Now plug your modified 2716 EPROM into this jumper socket and your all done! Make sure you have the notch pointing in the same direction as the other ROMs (towards the keyboard).

When you turn on the Apple you should see a screen full of garbage with the monitor prompt at the bottom of the screen. To boot your Apple, just type "6 ctrlP", and your computer will act just as usual.

Appendix C.

Making your own NMI board.

In this article I will describe how to make your own NMI board that will work in conjunction with the modified F8 monitor EPROM you have (or will) create. Here is the parts lists for the NMI board:

(1) 7400 or 74LS00 chip. Radio Shack part #276-1801. $0.59

(1) SPDT momentary push switch. Radio Shack part #275-1549. $2.19

(1) 14 pin low profile or wire wrap socket. Radio Shack part #276-1999 or #276-1993. $0.89

(2) 3.3k ohm resisters, 1/4 watt. Radio Shack part #271-1328. $0.39

(1) Dual plug-in interface board. Radio Shack part #276-164. $4.95 NOTE: This part has been discontinued by Radio Shack, but you can sometimes still find them in the junk bin. Cut the board so it will fit inside your Apple.

ALTERNATIVELY: (1) Apple bare board number PAPGBP5001. $13.95 from Priority Electronics, 9161 Deering Ave., Chatsworth, CA 91311.

After you have obtained all the parts above, you should solder the 14 pin socket and the two resistors somewhere convenient on the bare board. Next get some hi-gauge wire and make the following connections:

1) Connect pin 25 of the bare board to one leg of each of the two resistors.

2) Connect each of the other legs of the resistors to the two outside contacts of the switch. (one resistor goes to one contact, the other resistor goes to the other contact). Use some good wire.

3) Connect pin 25 of the bare board to pin 14 of the 14 pin socket.

4) Connect the middle contact of the switch to pin 7 of the 14 pin socket.

5) Connect pin 26 of the bare board to pin 7 of the 14 pin socket.

6) Connect pin 2 of the 14 pin socket to pin 6 of the 14 pin socket.

7) Connect pin 3 of the 14 pin socket to pin 4 of the 14 pin

socket.

8) Connect pin 4 of the 14 pin socket to pin 29 of the bare board.

9) Connect the leg of one of the resistors that is connected to the switch to pin 5 of the 14 pin socket.

10) Connect the other leg of the resistor that is connected to the switch to pin 1 of the 14 pin socket.

11) Check all connections twice, and don't get confused on what pin is what on the bare board and the 14 pin socket.

You have now completed building your own NMI board. This board may be plugged into any one of the peripheral slots.

Appendix D.

Practical uses for the NMI/modified ROM hardware.

Now that you have burned your own F8 monitor ROM, constructed your own NMI board and created a 16K slave disk with the previously mentioned files, its time to put it all together and use it (also make sure you have a RAM card in slot 0). The primary use for these hardware devices is for the single load program. As a practical example, we will be putting the Locksmith 5.0 fastcopy program into a file. This program is a really fast normal DOS copy program that is worth having in a file.

First turn off your computer and install your new F8 monitor EPROM into the motherboard, and put your NMI board in any slot. Now boot your Locksmith 5.0 (an original or a copy will do) and select the "16 sctr utilities" option. Next select the "16 sector fast disk backup". Now just after the drive stops spinning, and before you see the prompt "drive- original:1", hit the NMI switch on your NMI card. You should then be in the monitor.

Now boot your 16K slave disk. The "RAM 48K SAVER" program will run and will initialize a disk and save all 48K of memory to your disk.

Finally, run the "MEMORY MOVE WRITER" program and select the number of moves as one. Next select the running address as $8000. Use a forward memory move, and enter the start page as $40, and the hi page as $80. Next select the starting page to move to as $00. Finally, select the text page, page one, and full text. Now enter $8024 as the address to jump to and save the memory move program to disk.

Now its time to put all these files together as the final product. Boot a normal 48K disk and Bload the following files by typing:

```
]BLOAD ^00-3FFF,A$4000
]BLOAD MEMORY MOVE $8000,A$8000
]BLOAD RERUN,A$8024
```

Now make the file run when you brun it by typing:

```
]CALL -151
*3FFD:4C 00 80
```

Now we can save the final product by typing:

```
*BSAVE LS 5.0 FASTCOPY,A$3FFD,L$4040
```

Congratulations! You now have deprotected the Locksmith fast copy program into a file that you may brun anytime!

This technique will work well for deprotection other single load programs too! The main advantage to this technique is that you don't have to find the starting address of the program to restart it. The program will just start up from the point where you interrupted it.

The only other thing you really must do is determine what parts of memory you must save so the program will run. REMEMBER, YOU MUST ALWAYS SAVE MEMORY FROM $00 TO $2FF FOR THIS PROCESS TO WORK CORRECTLY! Use the Memory Move Writer to rearrange memory so you can save it in a normal DOS binary file.

If you want further practice in using your NMI/F8 EPROM hardware, write a program in APPLESOFT that some some screen displaying and interrupt the program. Then try and reconstruct it using the same technique as described above and restart the program.

You can save the BASIC program in a Bfile by saving $00 to $7FF and from $800 to the end of the program, where ever that might be (zero page locations $AF and $B0 will give you the ending location of a APPLESOFT program, in backassward order). You might also have to save some of the variable storage for your BASIC program, which lives from $95FF down (depending on size). The best thing to do is to experiment, and practice makes perfect.

Appendix E.

RAM card kraking.

In order to krak most protected programs you will have to have some method of breaking out of the program and into the monitor. In order to do this you will need a old style F8 monitor F8 ROM. Of course, this means you must buy an old F8 ROM and install it. If you have a //e, this is impossible, and if you leave the old F8 in your II+, it turns out to be pretty inconvenient to reset into the monitor all the time. You could be ripping your Apple apart when you need the old style reset and switch the ROMs, but that is rather inconvenient too.

Of course, we must rectify this situation for those without the means for extra hardware (ROMs, etc.) and those with //e's who can't buy an old F8 ROM. To do this, we can use the extra 16K of memory that most every one has. In the case of the II+, this is the RAM card. In the case of the //e, this is top 16K of the 64K of on-board memory.

What we will actually be doing is using the RAM card to simulate an old style F8 monitor ROM. We can do this since the memory that the RAM card occupies is the same memory area as the BASIC and monitor ROMs occupy on the motherboard. This memory region is from $D000 to $FFFF, in which the F8 monitor ROM lives from $F800 to $FFFF.

The RAM card can be accessed via some "soft switches", much as the hi-res page is turned on. Using these soft switches we can turn off the motherboard ROMs and turn on the RAM card. Before we can do this, we must copy the image of the BASIC and F8 monitor ROMs into the RAM card, so that we can switch off the motherboard ROMs. This way your Apple will still be able to control itself via the BASIC and the monitor ROMs copied into the RAM card.

After we copy the BASIC and F8 monitor ROM images from the motherboard F8 ROM into the RAM card, we can change the F8 code so that we are dumped into the monitor upon reset. This is easy to do by changing the reset pointer at $FA62 to jump to $FF59, the old style monitor reset point.

After we change the RAM card version of the F8 monitor ROM, we must finally change the RAM card soft switches so that the motherboard RAM is active and the RAM card memory from $D000 to $FFFF is active.

After we have done all this, we can boot our protected disk and reset into the monitor.

Of course, this method has some limitations. The first is that this is limited to programs that use the lower 48K of memory. If the program uses the upper 16K of memory (RAM card) it will overwrite our work.

The second limitation is that many software publishers are wise to this trick and their programs initialize the RAM card and wipe out anything that might be hiding there. The publisher that is most famous for this is Sirius Software.

A simple solution to this problem is to move your RAM card to another slot other than zero. Almost all programs will not expect a RAM card in any other slot than zero, so our modified code remains intact. This works very well but unfortunately, is impossible for Apple //e owners. For II+ owners, this is some what inconvenient since you have to play musically slots with your RAM card because no commercial programs will recognize your RAM card in any other slot than zero. But this is the price you have to pay for owning a //e or not owning a ROM card or old style F8 monitor ROM.

With this in mind, the best thing to do is to leave your RAM card in slot zero, copy and modify the ROM code, and try and break out of the protected program. If the protected program is familiar to this trick, then move your RAM card to slot one or two. This will make your RAM card "invisible" and we can load it with our modified code, without fear of it getting overwritten. So first try your RAM card in slot zero and type in:

```
]CALL-151
*C081
*D000<D000.FFFFM
*C089
*D000<D000.FFFFM
*C083 N C083
*FA62:4C 59 FF
*C080
```

Now boot your protected program and at the desired point, hit reset. If you reset into the monitor, you are successful. If the program does its normal tricks upon reset, then you will have to move your RAM card to another slot and try again.

Assuming you put your RAM card in slot two, type:

```
]CALL -151
*C0A1
*D000<D000.FFFFM
*C0A9
*D000<D000.FFFFM
*C0A3 N C0A3
*FA62:4C 59 FF
*C0A0
```

Notice the only different between the two sets of code is the $C000 addresses. This is because these refer to what slot your RAM card is in. For example, here is a chart of the slots and their $C000 addresses:

slot	address		slot	address
0	$C080	!	4	$C0C0
1	$C090	!	5	$C0D0
2	$C0A0	!	6	$C0E0
3	$C0B0	!	7	$C0F0

With this chart you can put your RAM card in any slot and change the above code accordingly. Remember to offset the new addresses by $01, $03, or $09 as done in the examples.

Now your Apple will act as though there is an old style F8 monitor ROM installed. So every time you hit reset, you should see the "*" prompt. Your Apple will continue to act like this until you powerdown, or till something overwrites the code in the upper 16K of memory.

(Remember to move your RAM card back to slot zero when you no longer need the old style reset so your other programs can find your RAM card).

Alternatively, I have written a program which will load your RAM card for you. It will ask you what slot your RAM card is in, and make the necessary changes. After the RAM card is loaded with the new code, the program will wait for you to touch a key and then clear the lower 48K of memory and reboot. This makes using your RAM card a real treat in trying to krak protected programs.

Appendix F.

The ROM card.

INTRODUCING THE ROM CARD:

The foremost of important tools for easily snooping through memory is the ROM card. The ROM card was originally developed for those with programs written in both INTEGER and APPLESOFT BASIC. Remember that your motherboard (the big green printed circuit board inside your computer case) can house only one of the BASIC languages, either INTEGER or APPLESOFT. When the Apple was originally released, it was only available with INTEGER BASIC. So many programs were written in INTEGER, and would not run on the Apple II+ (with APPLESOFT on board) when it was introduced.

Before RAM memory was very cheap, many people bought ROM cards for their Apple II+ that could be put in slot zero (as you would a RAM card), to enable them to run programs that were written in either BASIC language. It was just as though you had loaded INTEGER BASIC into you RAM card, like the DOS 3.3 System Master's HELLO program does. When RAM cards became available at a reasonable cost, everyone started buying them because they are so much more versatile for the average folk. That is why you don't see ROM cards for sale too much any more. But for deprotecting Apple programs, the ROM card is indespensible.

Also, for the Apple II owner who wanted to run the newer APPLESOFT programs, the ROM card was available with APPLESOFT ROMs. The INTEGER and the APPLESOFT versions of the ROM card are identical, except for the actual ROMS on the card. In other words, one had INTEGER ROMs and the other had APPLESOFT ROMs, and there is no other differences.

THE REASONS WHY:

Their are several reasons the ROM card is so important. The least of the reasons is the need for INTEGER BASIC or the Programmer's Aid chip. If you can get a ROM card cheaply without INTEGER or the Programmer's Aid ROMs, do so. From a cost outlook, it is to your advantage. Besides, INTEGER is a dead dinosaur, and who really cares if it's faster than APPLESOFT?

The reason we want a ROM card is so we can put an old style F8 monitor ROM and THE INSPECTOR ROM (from Omega Microware) on it. These two ROM chips are really essential for learning more about protected programs. Ultimately, we would like WATSON in conjunction with THE INSPECTOR, but to do so you will also need INTEGER BASIC ROMs, since WATSON uses some routines from the INTEGER BASIC ROMs. Watson enhances the Inspector by adding even more commands and flexibility. The combination of Watson and the Inspector provides you with great power for not only snooping, but also for general purpose utility chores.

The reason we want the old style F8 ROM should be obvious by now. After reading several kraking articles and from your own experiences, you have noticed that it is impossible to break out of many programs with just an autostart F8 monitor ROM. The reason we should have the old style F8 ROM on the ROM card and not on the mother board is for convenience. The ROM card has a switch which determines which F8 monitor ROM is active when you hit reset. So you can have the convenience of the Autostart F8 monitor ROM, and when you need it, hit the switch and be able to break out of any program you want with the old style F8 monitor ROM.

OBTAINING YOUR OWN ROM CARD:

ROM cards are available used at very cheap prices. Check your local Apple users' group. Alternatively, you can get blank cards and stuff it yourself. I would suggest stuff your own since the parts are easy to get, and it is usually the least expensive route! I have also seen Japanese clone cards for sale at a very reasonable price. The best place to check for these is in The Computer Shopper, a bi-monthly newspaper of Apple and other computer bargains.

OBTAINING YOUR OWN ROMs:

You can either buy an old style F8 monitor ROM, or you can make one by changing your autostart F8 code slightly. After making the change, you can save the file to disk and have a friend or your local computer store burn the image into a 2716 EPROM. Here is the instructions for creating your own:

1) Boot a normal DOS 3.3 disk.

2) Enter the monitor by typing:

```
]CALL-151
```

3) Move the autostart F8 ROM image into RAM by typing:

```
*4800<F800.FFFFM
```

4) To enter the monitor when reset is pressed, type these changes:

```
*4FFC:59 FF
```

5) Bsave the file to a blank disk by typing:

```
*BSAVE OLD $F8,A$4800,L$800
```

6) Burn this image into a 2716 EPROM.

This new F8 EPROM will be just like the autostart version F8 ROM except when you hit reset, you will be in the monitor and not in

BASIC. Now you can reset out of any program.

Alternatively, you can use a modified F8 EPROM too, as described in other kraking articles. This will give you the advantage of being able to save memory from $00 to $8FF when you hit reset. This would certainly be helpful at times.

If you want INTEGER BASIC on your ROM board, you can either buy the ROMs from your local Apple dealer, or you can make them. When you bought your Apple disk drive and controller you also bought DOS 3.3, the DOS 3.3 System Master, and all the programs on the System Master, including INTEGER BASIC. So you can also burn INTEGER into 2716 EPROMs just like you burned your new F8 EPROM, and put them on your ROM card. Here are the steps to do this:

1) Boot your DOS 3.3 System Master.

2) Bload the file INTBASIC by typing:

```
]BLOAD INTBASIC,A$2000
```

3) Bsave the INTEGER files to a blank disk by typing:

```
]BSAVE INT $E0,A$3000,L$800
]BSAVE INT $E8,A$3800,L$800
]BSAVE INT $F0,A$4000,L$800
```

4) Burn three 2716 EPROMs from each of these files.

IMPORTANT: In order to use 2716 EPROMs on your ROM card instead of the Apple-made 9316 ROMs, you must solder two "jumpers". Notice to the right of the F8 ROM socket on the ROM board white circle with the word "2716" next to it. Inside the circle will be four solder pads, grouped into two pairs. Notice each pair has two pads real close together, but not touching. Take a soldering iron and cross each pad in each pair together with some solder. So now the circle will have two solder pads, instead of four. DO NOT CROSS ALL FOUR PADS TOGETHER! Your ROM board will now except ONLY 2716 EPROMs, so when you do this you have to use all 2716 EPROMs, and no 9316 ROMs.

While on the subject of jumpers, there is another jumper on your ROM card just below the E8 ROM. This jumper, when crossed, will ignore the position of the ROM card switch. Reset will always ignore the F8 monitor ROM on the ROM board, and just use the motherboard F8 monitor ROM. Obviously, we do not want to cross this jumper.

If you can't tell if you should cross the 2716 jumper because you don't know if you have 2716's or 9316's, it is easy to tell the difference. 2716's have a small quartz window on their face, usually beneath some label. The window is used to erase the EPROM (hence the name Erasable, Programable, Read Only Memory). They should also say "2716" somewhere on them too.

If you must mix 9316's and 2716's on the same ROM card, do not cross any of the two pairs of jumpers. Instead, refer to APPENDIX B on how to make 2716 scrambler sockets for using 2716's in 9316 applications.

9316's are the all black 24 pin ROM chips that come with your Apple, and are not erasable. They will not have a quartz window.

Now plug in your F8 monitor EPROM or ROM in the socket labeled F8, and do the same with the other E0, E8 and F0 INTEGER EPROMs or ROMs. We are ready for the next step.

THE INSPECTOR:

The next thing the ROM board enables us to do is to use THE INSPECTOR from Omega Microware. The Inspector is basically a sector editor program with some really nice features which come in handy when deprotecting programs. To use The Inspector, we just reset out of a program and into the monitor, and type C080 N D800G. Now The Inspector is running without disturbing anything in memory outside of what normally gets disturbed upon hitting reset.

Besides being a sector editor, The Inspector has a very useful FIND command which enables us to find any string of bytes in memory or to locate them on a disk. This can help us find where a particular routine is being called from, or to help find the starting address of a program, etc. Also, The Inspector has a free sector map, removes DOS from a disk, does nibble reads of protected disks, displays bytes in HEX or ASCII, reads half tracks, and compares or verifies disks. It also has unlimited uses in snooping and changing memory and disks.

The Inspector is VERY useful, especially in conjunction with its partner, WATSON (also from Omega Microware). It is the most powerful and well used utility I have. And since it is on my ROM card, it is always available without disturbing mother board memory. This is why it is so useful. If we had to load it in from disk like any other program, it would be just like any other sector editor to a large extent.

Ask around and try and find someone with the Inspector and Watson code saved in a Bfile so you can burn your own Inspector EPROM and plug it into your ROM card. If you buy the Inspector, BE SURE you tell Omega when buying The Inspector that you want it in 2716 EPROM form if you are planning on using only 2716 EPROMs on your ROM card, instead of 9316's.

WHERE Do I PUT IT?:

Now that you have a ROM board, what slot should you put it in? Generally, the conventional slot is slot zero. But, I am sure many of you have RAM cards in slot zero. It is probably best 99 percent of the time to have your RAM card in slot zero, since most programs which use RAM cards expect it in only slot zero (although it has some uses in other slots). So that leaves you with two choices, put your ROM card in another slot, or play musical slots when you need the ROM card.

I prefer to put my ROM card in slot two since the card (and The Inspector) is still always available, but that presents some problems. The main problem is that after flipping the ROM board switch up to use the old F8 monitor ROM and hitting reset, your computer cannot find APPLESOFT when you boot a disk, it can only find INTEGER BASIC (assuming you have it on the ROM card). One way out is to flip the switch back down and hit reset again before booting a disk. I do not recommend this when deprotecting a program since now your computer will jump to the reset routine that was there when you originally hit reset. Of course, there is a better way.

After reseting into the monitor and just before you boot a disk you must turn off your ROM card ROMs and turn on the motherboard ROMs. This is accomplished with a softswitch, much like turning on the hi-res page. Remember how we activated the Inspector with C080 N D800G? Well, the C080 turns on the ROM card, so those ROMs are active, much like typing INT from BASIC. If you type C081 from the monitor, this turns the ROM card ROMs off, and the motherboard ROMs on. If your ROM card is in another slot, you need to type the slot number times ten, and add it to C081. Then you can boot a disk, and APPLESOFT will be found. Here is a chart of what you would type from the monitor just prior to booting a disk (you do not have to do this if your ROM card is in slot zero):

SLOT	TURN ON ROM CARD	TURN ON MOTHERBOARD
0	C080	C081
1	C090	C091
2	C0A0	C0A1
3	C0B0	C0B1
4	C0C0	C0C1
5	C0D0	C0D1
6	C0E0	C0E1
7	C0F0	C0F1

For example, if your ROM card was in slot two, and you have reseted into the monitor, type:

```
*C0A1
```

before you boot a disk to turn on your motherboard ROMs so APPLESOFT can be found.

Likewise, if you have reset into the monitor and you want to use the Inspector, type (assuming slot two):

```
*C0A0 N D800G
```

Notice we multiply the slot number by twenty and add it to $C080 or $C081.

Another alternative is to use DAVID DOS from David Data when you boot normal DOS 3.3. This DOS is incredible in just speed savings of loading programs. It will also recognize your ROM card in any slot (and hence solves our problem), has a relocatable DOS function to put DOS in your RAM card, has a find command, and has a disassemble command. If that is not enough, it has a TLOAD and TLIST command which loads and lists text files like BASIC or binary files! This alone make DAVID DOS worth the price. The only disadvantage to David DOS is it does not have an INIT disk command. To put David DOS on another disk requires using a program that comes with it.

Of course, if you are booting a disk which does not run under normal DOS, you can not use David DOS and you must use the first alternative.

CONCLUSION:

This completes our discussion of ROM cards and what configuration is most desirable. In summary, we would like a ROM card with an old style F8 monitor ROM, The Inspector, and ultimately, INTEGER BASIC and WATSON. Next we will discuss some general methods of deprotecting single load programs. Next we will discuss some general methods of deprotecting single load programs.

Appendix G.

Memory Move writer documentation.

In hopes of making your job easier in the area of memory moves, I have written a program that will write the memory moves for you.

After BRUNing the program, you will be asked how many memory moves you wish to have. Enter the number of moves between one and eight.

The next prompt will ask at what address the memory move will be running at. Respond to the address in hex, of course.

You will be asked to confirm your above choices before moving to the next set of prompts in the program. Type "Y" to proceed, "N" to re-enter any of the above data.

Next, the program will ask if the move is a "forward" or "backward" move. Respond accordingly. A forward memory move will take memory from a originate address and move it to a destination address by taking the lowest address and moving it to the lowest destination address.

A backwards memory move takes memory from the originate address plus the length of the move and moves it to the destination page plus the length of the move. This moves memory in a backwards fashion.

Now, you will be asked what is the beginning page number to start moving memory from. Enter whole pages only and in hex. If you enter "08", then the memory move will start moving from location $0800.

The top page to move from will be prompted for next. Enter in whole page only, and in hex. This page is exclusive, so if you enter "20", memory will be moved from $0800 to $1FFF (up to page $20).

Now you will be prompted for the destination page to move the memory to. Enter a whole page only and in hex. This will be the starting page that $0800 to $1FFF is moved to. So if you enter "60", memory from $0800 to $1FFF will be moved to $6000 to $7FFF.

These prompts will be repeated for the number of memory moves that you specified.

Lastly, you will be ask what screen pages you want turned on after the memory moves are done. Your choices are text, lo-res and hi-res pages one or two, and in full or mixed graphic/text. Also asked in the address you want to jump to after the moves and the screen is set. Enter the starting location of your program, or the address of the next section of code you want executed.

You will now be asked to put a disk in the desired drive and to enter "1" or "2" for the drive you want the memory move program saved to. After you enter the prompt, the program will automatically save the memory move program to that drive.

Remember, the memory move program will only run at the address specified in the initial prompt. If you need a memory move to run somewhere other than initially specified, restart the memory move writer program. Note that the catalog name of the memory move written to disk contains the running address of the memory move program.

Appendix H.

Disabling Minor Disk Access from Single Load Programs.

Many single load programs have a small amount of disk access that you will have to defeat in order to krak the program into a single file. I will give you a couple tricks 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 krak the program into a in single file, we must defeat this. Generally, authors use RWTS directly to do any disk access, thus using RWTS.

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 kraked 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".

But 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 single file krak.

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".

The reason we use "18 60" is simple: "18" represents a "clear the carry", and 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 NEAT WAY OF DEFEATING THE 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 byte of the routine to the starting page of RWTS, and the forth from last byte to the ending page of RWTS.

For an example, I use a similar routine to defeat the disk access in "Mating Zone" from Datamost (see the file "MATING ZONE" under single load protection). In this case, the loader (RWTS) lives from $400 to $7FF, and I move 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!

Hope this helps all you novices defeat the disk access in those single load programs.

Appendix I.

Using COPYB to krak Thief.

Thief is an interesting game much like the arcade game "Beserk". Your job is to destroy robots before they destroy you. Although the graphics are very good, and the game plays well, the boot is very long and with much drive head access.

Thief is one of the more difficult applications of COPYB I have seen. For this reason I am outlining how to krak it with COPYB.

The protection used in Thief is interesting. When you boot Thief, it seems to have a fairly standard boot, and a BASIC prompt appears. This tells us that Datamost is using a modified DOS, most likely with some sort of file structure. The interesting thing about the boot is after we hear the standard DOS boot sounds (loading tracks 0 through 2 into $9D00 to $BFFF), the disk reads tracks 4 and 5 very quickly, and then shows us a BASIC prompt.

After the game is loaded, I snooped through RWTS to see if anything looked suspicious. RWTS starts at $B700, so this is the best place to start. At $B738, there is some interesting code. We see this code:

```
B738-  A9 DE    LDA #$DE
B73A-  85 48    STA $48
B73C-  A9 AE    LDA #$AE
B73E-  85 31    STA $31
```

The reason this looks suspicious is that locations $31 and $48 get destroyed when you hit reset. This means Datamost is probably trying to hide something there.

Indeed they are! The two bytes $DE and $AE are the epilogue bytes used on the Thief disk. What Datamost is doing is hiding them in volatile locations $48 and $31. The idea is that they store the epilogue bytes in these two volatile memory location, and when needed, load from these locations. Now if the inspiring krakist was to hit reset and try to use COPYB to copy the disk onto a normal DOS disk, they could not read the disk. This is because the locations that hold the epilogue bytes ($48 and $31) were changed when you hit reset. This is the secondary protection used on Thief to keep the krakist from deprotecting Thief easily.

The primary protection used is a nibble count on tracks 4 and 5. If you examine the code at $9D84, which is the start of the DOS warmstart routine, you will see a jump to $AE8E. In normal DOS (and Thief's DOS) the routine at $9D84 is the routine that gets jumped to after DOS has been loaded. Normally this runs the hello program on a disk. Instead, the code at $AE8E loads in track 3 sector $9 and sector $A into $8000 to $81FF. It then jumps to this code at $8000. The code at 8000 does an indirect jump to

$8104, which is where the nibble count actually starts.

This nibble count checks track 4 and 5 for the byte sequence $DB $AD on the disk. It starts counting the nibbles on the disk and stores them at locations $81BA and $81BB. The code looks like this (formatted in 80 columns):

```
810C-  A9 04       LDA #$04     :load accum with track #
810E-  20 55 81    JSR $8155    :jump to nibble count
8111-  A9 05       LDA #$05     :load accum with track #
810E-  20 55 81    JSR $8155    :jump to nibble count
.
.
.
8155-   A2 60      LDX   #$60   :
8157-   0A         ASL          :Seek the drive head.
8158-   20 08 BA   JSR   $BA08  :----------------------------------
815B-   A9 00      LDA   #$00   :
815D-   8D BA 81   STA   $81BA  :Clear the  nibble count bytes.
8160-   8D BB 81   STA   $81BB  :
8163-   BD 8E C0   LDA   $C08E,X:----------------------------------
8166-   BD 8C C0   LDA   $C08C,X:Turn the drive on
8169-   10 FB      BPL   $8166  :and check if the disk is present.
816B-   C9 FF      CMP   #$FF   :
816D-   D0 F7      BNE   $8166  :
816F-   EA         NOP          :
8170-   BD 8C C0   LDA   $C08C,X:
8173-   10 FB      BPL   $8170  :
8175-   C9 FF      CMP   #$FF   :
8177-   D0 ED      BNE   $8166  :
8179-   EA         NOP          :----------------------------------
817A-   BD 8C C0   LDA   $C08C,X:Read the disk
817D-   10 FB      BPL   $817A  :and check
817F-   C9 DB      CMP   #$DB   :for the byte $DB.
8181-   F0 0B      BEQ   $818E  :If found, branch to $818E.
8183-   EE BA 81   INC   $81BA  :Else increm. 1st nibble count byte
8186-   D0 F2      BNE   $817A  :If 1st nibble byte not 0 goto 817A
8188-   EE BB 81   INC   $81BB  :Increment 2nd nibble count byte.
818B-   4C 7A 81   JMP   $817A  :Jump to $817A
818E-   EA         NOP          :----------------------------------
818F-   BD 8C C0   LDA   $C08C,X:Read the disk
8192-   10 FB      BPL   $818F  :and check
8194-   C9 AD      CMP   #$AD   :for the byte $AD.
8196-   F0 03      BEQ   $819B  :If found, branch to $819B.
8198-   4C 00 C6   JMP   $C600  :Else reboot and start over.
819B-   60         RTS          :----------------------------------
.rm72
```

If the byte sequence $DB $AE is not found, the disk will reboot and try all over again. This protection is probably sufficient to make nibble copies of Thief difficult.

Now that we have determined the primary and secondary protection, how do we defeat it?

The easiest way to deprotect Thief would be to use CopyB. The problem, as I mentioned before, is that Thief's RWTS tries to load the epilogue bytes from volatile memory locations. But we can defeat this by putting the actual epilogue bytes into the RWTS instead of having it load them from other locations. To do this, after the Thief DOS is loaded type:

```
*B902:C9 DE
*B90C:C9 AE
*B947:C9 AE
*B967:C9 AE
*B971:C9 DE
*B99B:C9 DE
```

Now we may use COPYB to deprotect Thief. You will notice, when done, that the startup program is named HELLO, and running this program will load and run Thief.

It is important to note that we do not have to concern ourselves with the nibble count on tracks 4 and 5 since it is called from Thief's DOS, and not the actual program. Since we will be using normal DOS 3.3, the nibble count is no longer a problem.

To use COPYB to deprotect Thief, do the following:

1) Boot Thief and reset into the monitor after the BASIC prompt appears.

2) Make these changes to RWTS by typing:

```
*B902:C9 DE
*B99C:C9 AE
*B947:C9 AE
*B967:C9 AE
*B971:C9 DE
*B99B:C9 DE
```

3) Move RWTS down to $8000 by typing:

```
*8000<B700.BFFFM
```

4) Boot a 48K DOS 3.3 slave disk and type:

```
]RUN COPYB
```

5) Respond to the parameter questions as follows:

```
Sectors on disk: 11
Starting track to read from: 6
Ending track to read to: 34
Continue reading on error: YES
```

6) Copy the Thief disk to a blank disk.

Thief is now deprotected and may be RUN, copied to any other DOS 3.3 disk, or further examined.

Appendix J.

Deprotecting Robotron from Atarisoft.

Atari is certainly a name that everyone is familiar with when it comes to video games. Atari has successfully marketed other company's games (after buying the rights, of course) for many home and personal computers. And finally they have started marketing games for the Apple.

This is good for us, the Apple user, since now we can enjoy many of the favorite arcade games on our Apple. Atari has also blessed us with weak copy protection, making most of the new Atari releases easily to krak.

A case in point is Robotron. Robotron 2084 is the best implementation of the William's arcade game I have seen for the Apple. My hat is off to the author of the Apple version, who ever it is (for some reason Atari leaves the author's name out of the game!?). But even though the game is well done, not much time was put into protecting it from prying eyes, especially since copy protection has evolved so far on the Apple....

Atari uses a slightly modified DOS. This is evident from the conventional boot sounds and the appearance of an APPLESOFT cursor at the bottom left of the screen when booting the disk.

Just for fun, after booting the disk, type CTRL C. This will prevent the basic HELLO program from running after DOS is loaded on an conventional DOS 3.3 disk. If you try this on Robotron, you will find the same thing...the computer beeps, the drive stops spinning and you are placed in APPLESOFT. You may now list the BASIC program with the command LIST. This reveals a one line program that reads:

```
10 HOME: CLEAR: PRINT CHR$(4);
   "BNROBOTRON"
```

From this program we now know there must be a catalog track since we can see that some DOS command is used (called BN) to run the file ROBOTRON. We know that BN is some kind of DOS command since preceding it is a CHR$(4) which tell APPLESOFT that the next command is a DOS command.

So naturally the next thing to do is to type "CATALOG". Well, we get disappointed with a SYNTAX ERROR. The conclusion we can draw from this is that someone at Atari was thinking enough to change the DOS commands from the normal ones (they probably used DOS BOSS from Beagle Brothers, no doubt).

So the next thing to do is to boot a normal DOS 3.3 disk and then put the Robotron disk in a drive and now try typing "CATALOG". This exercise provides us with the rewarding message "I/O ERROR", but this is to be expected. Atari has made the disk uncopyable by changing the epilogue bytes on the disk from DE AA EB to a perverted DE AB FE. This can be seen by using the nibble read commands from either THE INSPECTOR from Omega Software or NIBBLES AWAY II (if you don't have either of these fine utilities, don't worry about it though).

For those of you who don't know what "epilogue bytes" are, I will discuss it here for you....

First we must discuss the formatting of a DOS 3.3 disk. Every normal DOS 3.3 disk has 35 tracks (0 through 34) and 16 sectors (0 through 15). How the tracks are located on the disk is hardware dependent, but how the sectors are located is software dependent, hence the name "soft sectored". Since software determines the sectoring, it was easy for Apple to change from 13 sector format to 16 sector format back in 1981. This convenience is also why it is so easy to protect the Apple disk format. For DOS to find the sector it is looking for, it must rely on some road markers. Every sector has what is called an "Address field". The address field is a unique set of bytes on every sector that lets DOS know the current disk volume, track, and sector number. It also has a checksum byte to determine if some damage has occurred to the sector making it unreadable. The unique set of bytes that represent the address field are formatted as such:

```
prologue  vol trk  sctr chksm  epil.
------------------------------------
D5 AA 96!XXYY!XXYY!XXYY!XXYY!DE AA EB
------------------------------------
```

When ever DOS sees the unique set of bytes D5 AA 96, it knows the above information follows. Similarly, there is a "Data field". It has its own set of unique bytes to alert DOS to its where abouts:

```
prologue                chksm  epil.
-------------------------------------
D5 AA AD!PROGRAM,DATA,ETC.!XX!DE AA EB
-------------------------------------
```

When ever DOS sees the unique set of bytes D5 AA AD it knows that a program or some kind of data follows. This information is on every sector of a normal DOS 3.3 disk.

If any one of the prologue bytes are change, normal DOS would not be able to locate the Address or Data fields and an I/O error would result.

If the epilogue bytes are not what they should be, an I/O error will result. This is not due to their uniqueness, since DOS will read whatever two bytes follow the information fields and use them for verification.

Therefore, two easy things to do in protection is to alter the Address field and/or the Data field prologue bytes, and alter the protected DOS accordingly to locate these unique bytes. Now normal DOS can not find the Address field, so it does not know what sector it is trying to read, or it cannot read the data field because it cannot find the unique set of bytes that designates the data. So COPYA will not copy the protected disk.

What Atari has done is changed the epilogue bytes. This is really a minor change since normal DOS can still find the address field (so it knows what sector it is looking at) and the data field (so it knows where the data is), but you still get an I/O error since the computation of the checksum will not be zero.

Well, for us to read the disk from normal DOS, all we must do it to defeat the routine that detects a checksum error. If we do this, we will not get an I/O error and we will be able to read the disk from normal DOS 3.3. The routine that does this lives at $B942. If an error exists, the carry byte is set and DOS say "bad boy" and scolds you with an I/O error. So changing location $B942 from $38 to $18 to cure this problem!

Now we may catalog the Robotron disk. Upon doing this we find two file, RUNNER and ROBOTRON. RUNNER is the hello program and is unneeded (and also unusable without modification since Atari has change the DOS commands as follows: CATALOG has been removed. BLOAD is BD. BRUN is BN.). So get out FID from your DOS 3.3 System Master and BRUN FID. Now transfer the file ROBOTRON to a normal DOS 3.3 disk and you're all done!

To re-cap the instructions used to deprotect Robotron:

1) Boot normal DOS 3.3

2) Initialize a disk with normal DOS 3.3 by typing:

```
]FP
]INIT HELLO
```

3) To enter the monitor, type:

```
]CALL-151
```

4) Change the error detection routine by typing:

```
*B942:18
```

5) Insert your DOS 3.3 System Master into a drive and type:

```
*BRUN FID
```

6) Use FID to transfer the file ROBOTRON from the original Robotron disk to your freshly initialized DOS 3.3 disk.

7) BRUN the file ROBOTRON on your normal DOS 3.3 disk to play Robotron.

NOTE: This technique also works for some other Atari releases like PAC-MAN, CENTIPEDE and some others.

Appendix K.

Kraking PANDORA'S BOX from Datamost.

Generally when a publisher buys a protection scheme, they use it for as many programs as possible to get the most for their money (yes, protection schemes are just programs that people write and sell). The advantage to this is once you learn what they are doing, it is easy to back up many of their programs. Datamost and Infocom are two examples of this. If you can backup ZORK I, then you can backup ZORK II and ZORK III and DEADLINE and.....

Datamost used a modified DOS for many of their programs until about April 1983. After this point, many of their programs use different protection. But Datamost published at least seven good games before April 1983, and PANDORA'S BOX is one of them. The method I am about to describe will apply to many of them, but we will be using PANDORA'S BOX as an example.

Datamost uses a modified DOS for its protection. Normally, this is apparent from the BASIC prompt that appears on the screen after a few seconds into the boot. But Datamost turns on the hi-res screen right when the boot starts to hide this. But we still know there is a modified DOS because of the way the boot sounds. Listen to your normal DOS disks boot. You will here the same sound every time: First the drive spins for a half second or so. This is track 0 sector 0 through sector 9 getting loaded into $B600 to $BFFF. Then you here the drive's read-write head slide up to track 2 to load in the rest of DOS. Track 2 and 1 are read in quickly and the read-write head slides up to the catalog track, 17, and the hello program is located and run. Now listen to Pandora's Box load in. You will here the same sounds. This is a dead give-away that a modified DOS is being used.

Whenever a modified DOS is used, the first thing you should do is to boot a normal DOS disk and defeat the DOS error checking. DOS checks the carry bit to determine if any errors have occurred in a disk access. If the carry bit is clear, DOS assumes that everything is ok and just keeps on going. The routine that gets jumped to if an error is suspected is at $B942. This simply sets the carry bit and returns to the calling routine. To defeat the error checking, we only have to change $B942 to $18, instead of $38. This simple mod will allow us to copy previously uncopyable disks with COPYA.

All that is left to do is to change their DOS just slightly to live in a normal DOS 3.3 environment. At track 0, sector 3, change byte $91 from $DF to $DE. What Datamost has done to make their disk "uncopyable" is to change the epilogue byte from the normal $DE to $DF. This will sufficiently confuse copy program, preventing easy copies. (If you do not know what is meant by an "epilogue" byte, please refer to BENEATH APPLE DOS by Don Worth and Pieter Lechner. This manual is indispensable for further understanding of DOS.) Also, byte $42 must be changed from $38 to

$18 on the same track and sector. This is an insurance policy, more or less, that everything will work correctly in the normal DOS 3.3 environment. What you are actually doing is changing byte $B942 in DOS like we did before to make the COPYA copy, but you are doing it directly to the disk for permanence. These two mods are all we need to do to make Pandora's Box a straight COPYA disk.

In cookbook fashion, here are the steps:

1) Boot normal DOS 3.3

2) Type:

```
]CALL -151
```

3) Change byte $B942 from $38 to $18 by typing:

```
*B942:18
```

4) Type:

```
]RUN COPYA
```

5) Copy Pandora's Box to a blank disk.

6) Re-boot normal DOS 3.3 and run your sector editor and change the following bytes:

TRACK	SECTOR	BYTE	FROM	CHANGE TO
00	03	42	38	18
00	03	91	DF	DE

7) Write the sector back out to your copya disk version of Pandora's Box.

Appendix L.

Kraking Jungle Hunt from Atarisoft.

Jungle Hunt is a classic example of protection using modified address and data epilogue bytes. Of course, we know that it is easy to copy a disk with modified epilogue bytes by defeating the DOS error checking routine in normal DOS (at $B942) and copying the disk with COPYA.

Now you may ask how I figure this out (actually, a very good and common question for novice krakers). Notice how Jungle Hunt boots, you hear the normal DOS boot sounds and a BASIC prompt appears on the screen. This is a dead giveaway that a modified DOS is used. So the next step I always try is the easiest: defeat the DOS error checking routine at $B942 and try copying the disk with COPYA.

With Jungle Hunt, it worked, as you will find it works with many protected programs that use a modified DOS. To do this, type:

```
]CALL-151
*B942:18
*RUN COPYA
```

The whole Jungle Hunt disk copies with no problems. So now, after the copy is complete, CATALOG the disk and see if it uses a normal DOS file structure. You will find it does to a certain extent, and you will see a text file and a binary file. The text file is the "Hello" program, which EXECs the binary file named "A". (This is easy to see if you load track 1, sector 9 with a sector editor and examine bytes $75 to $92. These bytes hold the "hello" program's name.) This binary file "A" uses second stage DOS to read in the game (bload the file and try and figure it out).

Now all we must do is to normalize the present DOS on the disk to work in our normal DOS environment (our copy now has standard epilogue bytes). Not being much of a glutton for punishment, instead of fooling with their DOS, I choose to try and use normal DOS 3.3, with some small modifications.

So whip out the DOS 3.3 System Master and BRUN MASTER CREATE. Now put DOS on the copy of Jungle Hunt, and use "A" as the hello program name. Now all we must do is to change the new DOS so it can have a binary program for a "Hello" program.

So now get your sector editor running and change track 0, sector D, byte $42 from $06 to $32 and your all done!

In summary, what we have done is to make a COPYA copy of the disk by defeating the normal DOS error checking, put normal DOS on the disk and made "A" the hello program.

In cookbook fashion, here are the steps:

1) Boot your DOS 3.3 System Master.

2) Enter the monitor by typing:

```
]CALL-151
```

3) Disable the DOS error checking routine at $B942 by typing:

```
*B942:18
```

4) Run COPYA and copy the original Jungle Hunt disk by typing:

```
*RUN COPYA
```

5) After the copy is complete, put normal DOS on the copy of Jungle Hunt. To do this, boot the DOS 3.3 System Master and type:

```
]BRUN MASTER CREATE
```

6) Use "A" as the Hello program name and copy DOS to the Jungle Hunt disk.

7) Run your sector editor and change track 0, sector D, byte $42 from $06 to $32 of the Jungle Hunt copy.

8) Write the sector back out to the copy of Jungle Hunt.

And your all done!

Appendix M.

Kraking Zaxxon from Datasoft.

(Please note: There has been a new release of Zaxxon from Datasoft, and this is the one I will be primarily discussing, although the same protection was used on both versions. You can tell if you have the new version because it has an option to use the Mockingboard with it. If your Zaxxon boots and asks 'MOCKINGBOARD IN SLOT 4 Y/N?', you have the new version. In addition, I have seen two different protections used on the newer versions of Zaxxon. I will give details on the deprotections of all versions, including the older version of Zaxxon.)

The first thing to notice is the boot of Zaxxon. Listen to your disk drive as the game boots and you can hear the drive arm swing out to an outside track and then swing back in and read in the game. This is what is loosely know as a 'nibble count' or 'checksum' routine. If when the outside track is read, a byte doesn't match a benchmark it should, the game will clear memory and reboot. Usually, the only thing involved in deprotecting a program that isn't a single load and that has a 'nibble count', is to find the routine that does the check and jump around it... usually about the only way to find this routine is to (yech!) trace the boot.

Boot code tracing is a method of tracing how a program gets from your disk, to memory. It does not magically happen all at once, but in stages, which we can trace and examine, and hopefully understand (hence the name 'boot code tracing'). The theory of boot-code tracing is to follow the boot process one step at a time to see where is takes you by altering the code to prevent it from running away from you. Yes, it is advisable that you understand assembly language, since the code that boots the disk is in many cases, intentionally misleading.

This process is based upon the law that track zero, sector zero must always be read by the disk controller card into page 8 ($800-8ff) of memory. after this, depending on the complexity of the protection, it is sometimes difficult to understand what goes on in the rest of the load.

In a normal slave disk boot, there are three stages of a boot starting with the code at $C600-C6FF in the disk controller card. This codes loads in track $0, sector $0 into $800-$8FF, which in turns loads in track $0 sector $0 through sector $9 into $B600-BFFF. This code loads in track $0, sector $c through track $2 sector $4 into $9D00-$B5FF and finally your hello program is run. In summary:

```
STAGE    CODE       DESTINATION  JUMPS
NUMBER   LOCATION    LOCATION     TO
--------------------------------------
  0     $C600-C6FF  $0800-08FF   $0801
  1     $0801-08FF   $B600-BFFF   $B700
  2     $B600-BFFF   $9D00-B5FF   RUN
                                 PROGRAM
```

Now, in order to change the code in the boot so it doesn't run away from us, we can either alter memory or alter (a copy of) the disk. If we defeat the dos error checking routines, we can copy the Zaxxon disk with COPYA. Of course it wont' run, because of the 'check' routine and some other incompatibilities with normal DOS 3.3, but we can read and write to the copy easily. To defeat the error checking, enter the monitor with CALL-151 and enter:

```
*B942:18
```

Whenever DOS encounters an error it jumps to a routine at $B942 that sets the carry bit and returns. The carry bit is a flag to DOS that there was an error, to stop whatever it was doing, and print a worthless message out to the user. If we defeat this routine, we can fool DOS and read the entire Zaxxon disk. We can now copy the Zaxxon disk and examine the data on it. So put your DOS 3.3 System Master in the drive and run COPYA. When the prompt for source drive is asked, hit ctrl C. This will break you into BASIC. Now type the following:

```
CALL-151

*3A1:18
*302:17
*35F:17
*3D0G

70
RUN
```

This will make a change to COPYA.OBJ0 and delete line 70 of the Applesoft part of COPYA (to prevent loading in COPYA.OBJ0 and writting over the change just made). Now select desired drives and copy Zaxxon. Note: Zaxxon only lives from track $0 to track $16. When track $17 is encountered, so we can change COPYA so it only reads up to track $17. After the copy is made, make another copy of the duplicate of Zaxxon just made. Label this one 'WORK ZAX' and the other copy 'COPYA ZAX'.

Now put these two copies aside for a moment. The next step is to trace the boot. First enter the monitor with CALL-151 and clear out memory with this command:

```
*800:0 N 801<800.95FFM
```

Now, to start the boot we need to run the code in our disk controller card, but stop it before it runs away. We can not change the code in ROM (Read Only Memory), but we can copy it down to RAM (Random Access Memory) and change it. Use this command from monitor to do this:

```
*9600<C600.C6FFM
```

Now we can change the boot code so that is loads in track $0, sector $0 but does not execute it. At location $96F8 you will see a JMP $801. This starts the next boot process (refer to table). Now we can change this to jmp $FF59, which will jump to the monitor. So, from monitor type:

```
*96F8:4C 59 FF
```

Now execute the code with:

```
*9600G
```

And the drive will spin then the apple will beep and you will see the '*' cursor. From our table we know this code loaded into $801. (To stop the drive from spinning, enter *C0E8 from the monitor. Also, to turn off the high res page enter *C054 and *C051.). So now examine the code at $801 with the command:

```
*801L
```

Upon examining the code, we find it is fairly normal with some exceptions:

```
0801-   A5 27       LDA   $27      !-------
0803-   C9 09       CMP   #$09     !
0805-   D0 18       BNE   $081F    !
0807-   A5 2B       LDA   $2B      !    N
0809-   4A          LSR            !
080A-   4A          LSR            !    O
080B-   4A          LSR            !
080C-   4A          LSR            !    R
080D-   09 C0       ORA   #$C0     !
080F-   85 3F       STA   $3F      !    M
0811-   A9 5C       LDA   #$5C     !
0813-   85 3E       STA   $3E      !    A
0815-   18          CLC            !
0816-   AD FE 08    LDA   $08FE    !    L
0819-   6D FF 08    ADC   $08FF    !
081C-   8D FE 08    STA   $08FE    !
081F-   AE FF 08    LDX   $08FF    !
0822-   30 15       BMI   $0839    !    D
0824-   BD 4D 08    LDA   $084D,X  !
0827-   85 3D       STA   $3D      !    O
0829-   CE FF 08    DEC   $08FF    !
082C-   AD FE 08    LDA   $08FE    !    S
082F-   85 27       STA   $27      !
0831-   CE FE 08    DEC   $08FE    !
0834-   A6 2B       LDX   $2B      !
0836-   6C 3E 00    JMP   ($003E)  !
0839-   EE FE 08    INC   $08FE    !-------
083C-   AD 55 C0    LDA   $C055    !TURN ON
```

```
083F-   AD 50 C0    LDA   $C050     !HI-RES2
0842-   AD 57 C0    LDA   $C057     !-------
0845-   A2 7D       LDX   #$7D      !???????
0847-   9A          TXS             !???????
0848-   4C B4 08    JMP   $08B4     !-------
         :  :  :
         :  :  :
         :  :  :
08B4-   A9 20       LDA   #$20      !
08B6-   85 1B       STA   $1B       !
08B8-   AD 52 C0    LDA   $C052     !
08BB-   A9 60       LDA   #$60      !
08BD-   8D 06 07    STA   $706      !-------
08C0-   6C F8 08    JMP   ($08FD)   !JUMP TO
                                    !$8000
                                    !-------
```

The first part of the code from $801 to $83B is lifted verbatim from a DOS 3.3 slave disk. This code loads in track $0 sector $0 through track $0 sector $09 into $7F00-$88FF. (This is revealed from location $8FE, which is one higher than the first page loaded into. The byte at $8FF is one less than the number of sectors to be loaded). The next piece of code turns on the hi-res screen. The last part of code, before the jump to $8B4, looks innocent, but really isn't. They load the X-register with $7D and transfer it to the stack pointer. To understand the complications of this, you must understand how the computer keeps track of where to return to after a RTS (Return From Sub-routine). When the 6502 encounters a JSR (Jump Sub-Routine), it store the present address on the stack so it knows where to return to when a RTS is encountered. This can be used to obscure code from unwanted eyes. For example, say we want to go to $9600. We can load the stack with $95 and then $FF by using the PHA op-code (Push Accumulator on Stack). When a RTS is encountered, the two last bytes are pulled from the stack, incremented by one (to $9600), and jumped to. Alternatively, we can change where the pointer on the stack is pointing to and make it point to where we want to go. Keep this in mind when you find a RTS.

At $8B4 a few zero page locations are loaded, and then there is an indirect jump to $8000 (through $8FD). To see the code at $8000, we need to load the next stage of the boot, but stop it before it can execute. to do this, run your sector editor and change track $0, sector $0,byte $C0 to 4C 59 FF on the disk labeled 'WORK.ZAX'. This will jump us to the monitor before it can execute the code at $8000. Now write the sector back out to the disk and boot the disk. After a moment, we will beep into the monitor and we can examine the code at $8000. Do this with the command:

```
*8000L
```

and the code will look like this:

```
8000-   A0 09       LDY   #$09      !-------
8002-   A2 00       LDX   #$00      !
8004-   8A          TXA             ! M
8005-   EE 0D 80    INC   $800D     ! E  M
8008-   EE 10 80    INC   $8010     ! M  O
800B-   BD 18 7F    LDA   $7F18,X   ! O  V
800E-   9D 7E 00    STA   $007E,X   ! R  E
8011-   CA          DEX             ! Y
8012-   D0 F9       BNE   $800B     !
8014-   88          DEY             !
8015-   DO EE       BNE   $8005     !-------
8017-   60          RTS             !JUMP
                                    !THRU
                                    !STACK
                                    !-------
```

Notice the RTS at $8017. Remember, we jumped, not jumped sub-routine, to everywhere we got to, so there is nothing on the stack to return to! Well, yes there is! The memory move moves memory from $7F19 to $8718 down to $7F to $087E. This moves memory across page $1, which is the stack! Remember, the stack pointer is set to $7D in the BOOT1. So after the memory move we do a RTS. Well the stack is pointing at $17D which is now $07 and $65 after the memory move. That is our next jump (plus one) for the finally stage of the boot!

Datasoft has added a finally bit of protection in that the next jump is across the text page, which of course changes when we exit the program in any manner. But we can merely move memory to, say $107E, and examine it to see what the next load does. To do this change $8010 to 10 and execute the memory move. Do this as follows:

```
*8010:10 N 8000G
```

NOW $1766 IS EQUIVALENT TO $766. SO
TYPE:

```
*1766L
```

And examine the code. It should look as follows:

```
1766-   A5 2B       LDA   $2B       !
1768-   8D 0E 02    STA   $020E     !  G  C
176B-   A9 20       LDA   #$20      !  O  O
176D-   EA          NOP             !  O  D
176E-   A6 2B       LDX   $2B       !  D  E
1770-   20 1F 02    JSR   $021F     !-------
1773-   A0 20       LDY   #$20      !
         :  :  :                    !NIBBLE
         :  :  :                    !COUNT/
         :  :  :                    !CHECK
         :  :  :                    !-------
17D4-   A9 16       LDA   #$16      !  G  C
```

```
17D6-  8D 11 02     STA  $0211    !  O  O
17D9-  A9 D0        LDA  #$D0     !  O  D
17DB-  8D 16 02     STA  $0216    !  D  E
17DE-  4C 9A 01     JMP  $019A    !
                                  !-------
```

If you sit and look at this code you will find the offending 'nibble count' or 'checksum' starts at $1773 and goes to $17D3! This is what we need to know to defeat it. All we need to do is to jump around this. We can do this by:

```
*1773:4C D4 07
```

The other loads and stores of the accumulator are parameters for their loader. For further understanding, here are the parameters:

$211 = High track number to start loading from.

$212 = High sector number to start loading from.

$21E = Number of pages (sectors) to load.

$21B = Starting page to load at.

JSR $1E9 = Start the load.

The last thing to do is to find where on the disk this code is and change it. Most good sector editors (like 'THE INSPECTOR') have locate routines so you can find a pattern of bytes on a disk.

One final note: Now that we know where the check routine lives we can defeat it. But we also must change the epilogue bytes to normal DOS 3.3. Datasoft uses $CC for their epilogue bytes where normal DOS uses $DE. If you are familar with loaders and RWTS you can find it in their loader. The table is in page $4 (in page $14 for us).

With this information in hand, here is a step-by-step procedure for deprotection Zaxxon (Mockingboard version only):

1) Run COPYA from your DOS 3.3 System Master. After it is loaded and asking what drives you want to use, press ctrl c to exit to BASIC.

2) CALL-151 to enter the monitor.

3)
```
*B942:18
*3A1:18
*302:17
*35F:17
*3D0G
```

4) 70 <return> To delete line 70 of COPYA.

5) Type run to run COPYA. Choose appropriate drives and slots.

6) Run your sector editor and make the following changes: (note, you may have to try all of the following changes depending on when your Zaxxon was released. One of them will work.)

TRACK	SECTOR	BYTE	CHANGE FROM	CHANGE TO
$00	$04	$4F	$CC	$DE
$00	$04	$50	$D0	$EA
$00	$04	$51	$AE	$EA
$00	$07	$0D	$A0	$4C
$00	$07	$0E	$20	$D4
$00	$07	$0F	$84	$07

7) Write the sector back out to disk.

A final note: the Zaxxon in the arcades has five planes and our Zaxxon only gave us three! If you want more planes, change byte $17 on track $09, sector $08 with your sector editor to the number of planes you want (between $00 and $FF). I choose to change this byte to $03. This gives you four planes which is a nice compromise between the five you get in the arcades and the three that Datasoft gave us. This modification applies to all versions of Zaxxon.

For older versions of Zaxxon without the Mockingboard:

1) Follow the same instructions as above but instead change these bytes with your sector editor:

TRACK	SECTOR	BYTE	CHANGE FROM	CHANGE TO
$00	$07	$00	$A9	$4C
$00	$07	$01	$01	$C0
$00	$07	$02	$48	$08
$00	$04	$4F	$CC	$DE

2) Write the sector back out to disk.

For mockingboard versions of Zaxxon in which the previous method did not work:

1) Follow the same instructions as above but instead change these bytes with your sector editor:

TRACK	SECTOR	BYTE	CHANGE FROM	CHANGE TO

```
$00      $07      $1F     $20     $4C
$00      $07      $20     $3E     $C0
$00      $07      $21     $02     $08

$00      $04      $4F     $CC     $DE
```

2) Write the sector back out.

Appendix N.

Kraking Electronic Arts' Cut and Paste, One on One, and the Last Gladiator.

Electronic Arts' recent releases have all used about the same protection. The protection has been to change the data field prologue bytes from a normal "D5 AA AD" to a modified "D5 BB CF" on tracks $03 to $20. Tracks $00 to $02 contain the modified RWTS to read the new data field bytes, and hi-res title page. These three tracks are unprotected normal DOS 3.3.

In addition, Electronic Arts uses a nibble count on track $22, and track $21 is unused.

Remember that the prologue data field bytes tell DOS where the data starts on a sector. This is usually identified by the unique sequence of bytes "D5 AA AD". Electronic Arts has modified these to "D5 BB CF" so normal DOS can not tell where the data starts, and hence an I/O error occurs when copying with COPYA (or many bit copiers for that matter).

So to deprotect the new Electronic Arts releases, we must:

1) Read the original disk with a data field prologue bytes of "D5 BB CF".

2) Write to a normal DOS 3.3 disk with a data field prologue bytes of "D5 AA AD".

3) Change the modified Electronic Arts' RWTS to read a normal "D5 AA AD" data field prologue bytes.

4) Disable the nibble count.

Easy enough, right? So we can use ADVANCED COPYA to read and write the modified data field bytes, and then use a sector editor to disable the nibble count and change their RWTS. Here is the procedure:

1) Boot normal DOS 3.3 and run ADVANCED COPYA by typing:

```
]RUN ADVANCED COPYA
```

2) Use the default parameters to copy tracks 0 to 2 to a blank disk (just press RETURN for all the prompts except ending track;

enter "2").

5) This will copy track 0 to 2 of the original Electronic Arts' disk to your blank disk. When done, hit "S" to start over and change these parameter:

Use Foreign RWTS: N
Starting track: 3
Ending track: 32
Read Half Tracks: N
Continue Reading on Errors: Y
Disable DOS error checking: Y
Change address/data headers: Y
Data Prologue Bytes: $D5 $BB $CF

(Don't forget to enter a "$" before any HEX inputs).

6) Copy the orginal disk (don't format the destination disk or you'll destroy tracks 0-2).

7) After the copy is done, reboot DOS 3.3 and run your sector editor. Make these changes to your COPYA copy:

Cut and Paste

trk	sct	byte	from	to
2	3	$47	$BB	$AA
2	3	$51	$CF	$AD
1	C	$05	$A0	$18
1	C	$06	$20	$60
1	C	$68	$20	$18
1	C	$69	$A2	$60
1	F	$68	$20	$18
1	F	$69	$A2	$60
1	F	$6A	$A1	$EB

The Last Gladiator

trk	sct	byte	from	to
1	F	$68	$20	$18
1	F	$69	$A2	$60
1	F	$6A	$A1	$EB
1	C	$05	$A0	$18
1	C	$06	$20	$60
1	C	$68	$20	$18
1	C	$69	$A2	$60
2	3	$47	$BB	$AA
2	3	$51	$CF	$AD
1F	E	$05	$A0	$18
1F	E	$06	$20	$60
1F	E	$68	$20	$18
1F	E	$69	$A2	$60
1F	F	$05	$A0	$18
1F	F	$06	$20	$60

1F	F	$68	$20	$18
1F	F	$69	$A2	$60

One On One

trk	sct	byte	from	to
1	F	$68	$20	$18
1	F	$69	$A2	$60
1	F	$6A	$A1	$EB
1	C	$05	$A0	$18
1	C	$06	$20	$60
1	C	$68	$20	$18
1	C	$69	$A2	$60
2	3	$47	$BB	$AA
2	3	$51	$CF	$AD
9	2	$1F	$01	$FD
C	4	$05	$A0	$18
C	4	$06	$18	$60
C	4	$07	$88	$C8
C	4	$DC	$A0	$18
C	4	$DD	$FF	$60

Now your all done! Don't forget to write the sectors back out to your copya copy as you change them.

Appendix O.

Kraking Roundabout from Datamost in a full disk format.

Roundabout is one of those games that uses a modified DOS for protection, but borderlines on being a single load protection program. This is because after the game is loaded, there is only some minimal disk access that can be defeated, allowing use to krak Roundabout into a single file.

But to have all levels of the game, the program must be in a whole disk format. Also, the protection is interesting and well worth discussion for educational purposes.

Roundabout uses second stage DOS to load in the game almost exactly as described in chapter 7. Datamost has copied track 0, sectors 0 to 9 from a 48K DOS 3.3 slave disk and has slightly modified this RWTS to add some copy protection.

The bulk of the protection is to use modified data and address epilogue bytes. But to be really tricky, they use different epilogue bytes for different tracks!

If you compare track 0, sectors 0 to 9 of Roundabout and a normal DOS 3.3 slave disk, you can find this routine that determines which epilogue bytes to use for which track:

```
BEAF-   85 2A       STA   $2A
BEB1-   86 2B       STX   $2B
BEB3-   A9 DE       LDA   #$DE  (1)
BEB5-   8D 9E B8    STA   $B89E
BEB8-   8D AE BC    STA   $BCAE
BEBB-   8D 35 B9    STA   $B935
BEBE-   8D 91 B9    STA   $B991
BEC1-   A5 2A       LDA   $2A
BEC3-   C9 22       CMP   #$22
BEC5-   B0 15       BCS   $BEDC
BEC7-   A9 DF       LDA   #$DF  (2)
BEC9-   8D 9E B8    STA   $B89E
BECC-   8D AE BC    STA   $BCAE
BECF-   8D 35 B9    STA   $B935
BED2-   8D 91 B9    STA   $B991
BED5-   A6 2B       LDX   $2B
BED7-   A5 2A       LDA   $2A
BED9-   4C A4 B9    JMP   $B9A4
BEDC-   18          CLC
BEDD-   69 01       ADC   #$01
```

```
BEDF-   85 2A       STA   $2A
BEE1-   4C A4 B9    JMP   $B9A4
```

Notice at (1) the first byte in the epilogue bytes is $DE and is stored in RWTS so it can read sectors with epilogue bytes of DE AA. Then at (2) the first byte in the epilogue bytes is $DF so it can read sectors with epilogue bytes of DF AA.

Datamost has taken a very standard protection of modifying the epilogue bytes a step further. But unfortunately for them, not far enough. If they had used this protection for either the data or address PROLOGUE bytes, their protection would have been much harder to krak.

The reason for this is since the prologue bytes are still normal, normal DOS can find where the address and data information starts, but we get an I/O error since the ending epilogue bytes don't match. This is very easy to go around by defeating the normal DOS error checking routine at $B942 and copying the disk with COPYA. (NOTE: we only need to copy from tracks $00 to $1E of the original disk. The rest of the disk is not even formatted.)

Then all that is left to do is to defeat the above routine and we're all done. We can search the disk with The Inspector for the byte sequence of "20 AF BE" or "4C AF BE". This sequence or "4C AF BE" is found at track 0, sector 3, and if you look at the code, we have to actually defeat 4 bytes of "4C AF BE 2A" by NOPing them to "EA EA EA EA".

So in cookbook form, here are the steps:

1) Boot your DOS 3.3 System Master and type:

```
]RUN COPYA
```

2) After the program is running and the drive stops, type:

```
CTRL C
```

3) You should now be in BASIC. Now enter the monitor and make these changes to COPYA so it only copies tracks 0 to $1E by typing:

```
]CALL-151
*302:1E
*35F:1E
*3A1:18
*B942:18
*3D0G
```

4) Now delete line 70 of COPYA so not to reload COPY.OBJ0 and restart COPYA by typing:

```
]70
```

```
]RUN
```

5) Copy Roundabout to a blank disk.

6) After the copy is done, run your sector editor and make these changes to your copy of Roundabout:

```
Track 0, sector 3, byte $A0
from $4C AF BE 2A
 TO  $EA EA EA EA
```

7) Write the sector back out.

and you're all done! Roundabout is now krak into a full disk format.

Appendix P.

Kraking Mouskattack from On-Line Systems.

Although Mouskattack is a rather old and not so thrilling maze game, it warrants discussion on de-protection methods and the usage of DOS from protected programs.

Upon booting the Mouskattack disk, the cursor appears in the lower left of the screen, indicating a somewhat normal DOS is used by the program. In fact, if you boot a normal DOS 3.3 disk and put your Mouskattack disk in the drive and type 'catalog', a director does appear. You will not see any files but just copyright notices and authors involved. But in fact these are files in the directory.

The disk seems almost unprotected, and to confirm this, I made a copy with COPYA from the DOS 3.3 System Master. It reads the non-DOS tracks slowly, but this is due to the sector skewing used by On-Line in hopes of a faster loading game (they were not too successful, and we will see why in a moment), and not due to the protection. The first three tracks (the DOS tracks) read at the normal speed because they are normal DOS just like from you DOS 3.3 System Master.

After making my 'COPYA' copy, I used a disk editor to read in track 11, sector F of the disk. This is the first sector of the catalog track. The way that On- Line got the directory to appear without file types and sector lengths was simple; the first seven (or six) characters in the directory name are back spaces. So if we change, starting with byte $0E, the first seven character to anything but control character, numbers or spaces, we can load and examine the files like any other DOS 3.3 file. Don't bother with the other file names, they are all blank files. The only one we are interested in is the first directory entry, 'MOUSKATTCK'.

Now get back into BASIC and catalog the copy of Mouskattack. Your catalog should have as a first entry, a binary file four sectors in length called 'XXXXXXXMOUSKATTACK'. We can now BLOAD this file and snoop through it. (At this point you might ask how I knew to do this. Well, since Mouskattack has a normal DOS on it, by loading in track one, sector 9 with a sector editor we can see what file is the boot file or 'HELLO' program. Sure enough, we see Mouskattack preceded by seven control H's (backspaces). Therefore we know that Mouskattack is the first file we should snoop through.)

So BLOAD the file XXXXXXXMOUSKATTACK, enter the monitor and examine locations AA72.AA73. This will tell us the loading location of the last BLOADED file. It will appear 'backasswards' with the low byte first and the high byte second. i.e.,00 08 is equivalent to $0800. So do a 800L to list the first screen full of the program.

Upon examining, we see the accumulator gets loaded and then saved to a location in DOS. Finally, after this happens a few times, you will see a Jump To Subroutine (JSR) at $B7B5. Now you ask, 'What is all this stuff?' What On-Line is doing is using the second stage of DOS to load in the game Mouskattack. You are looking at the first stage (of three stages) of the load. (We are actually tracing the boot, as describe in earlier issues of Hardcore, but using normal DOS.) The listing will appear as this:

```
800:LDA #$00
    STA $B7EB; VOLUME NUMBER, 0 MATCHES
               ANYTHING.
    LDA #$01
    STA $B7F4; COMMAND CODE, 1=READ
    LDA #$18
    STA $B7EC; TRACK NUMBER TO START
               LOADING FROM.
    LDA #$03
    STA $B7ED; SECTOR NUMBER TO START
               LOADING FROM.
    LDA #$95
    STA $B7F1; HIGH BYTE OF PAGE TO
               START LOADING TO.
    LDA #$00
    STA $B7F0; LOW BYTE OF PAGE TO
               START LOADING TO.
    LDA #$03 ; SECTOR NUMBER TO LOAD
822:STA  $14 ; SECTOR COUNTER IN 0PAGE
    LDA #$B7
    JSR $B7B5; SUBROUTINE IN RWTS TO
               READ A SECTOR.
    BCS $0822; IF ERROR, GOTO $822
    INC $B7F1; INCREMENT PAGE TO LOAD
    INC $B7ED; INCREMENT SECTOR TO LOAD
    DEC  $14 ; DECREMENT SECTOR COUNTER
    BNE $0822; IF SECTOR COUNTER <> 0
               THEN GOTO LOCATION $822
835:JMP $9500; JUMP TO LOCATION $9500
```

Now, what is happening is that starting with track 18, sector 3, is getting loaded into location $9500. The sector number and page to load gets incremented, and the process continues till sector F of track 18 is reached. Then we jump to location $9500 to start the next stage of the load. If we jump to the monitor at location $835 instead of $9500, we can examine the next stage of the load (Do this by doing 835:4C 59 FF N 800G).

At $9500 the same thing happens again but in a larger propective. The codes gets more obscure and difficult to follow, so i won't list it here.

Now you ask, what was the purpose of this exercise? By examining the code at $9500, We can find the starting page Mouskattack loads at, and even the starting location! (Can you see where?)

The starting location of the load is at $A00 and the starting location is at $5300. This is the reward for all this work. WE know that since the load code lives at $9500 and DOS occupies $9D00 and up, that Mouskattack must live from $A00 to $94FF with a starting location at $5300. (not mentioned is the third load that overwrites the demo code. This load is the only part of the disk that is really protected. Our COPYA copy will work up till this load, but not past it for the game.)

Now, remember that code from $900 to $95FF does not get destroyed by a slave boot? Well, since Mouskattack lives within this area, it becomes an easy crack. Here is the procedure:

1) Boot your original copy of Mouskattack.

2) After the demo and when the text appears 'HOW MANY PLAYERS?', insert a blank initialize slave disk.

3) Hit reset. It does not matter if you have an old style monitor or not. With an AUTOSTART monitor you slave disk will boot.

4) Enter the monitor by typing:

```
]CALL -151
```

5) Change DOS to allow you to save larger binary files to disk by typing:

```
*A964:FF
```

6) Make the file run when BRUN by typing:

```
*9FD:4C 00 53
```

7) Save the file to disk by typing:

```
]BSAVE MOUSKATTACK,A$9FD,L$8B03
```

A finally note: remember how I mention that the load of Mouskattack was rather slow? This is due to the way the sectors are read in from the disk. Recall that the sector number gets incremented along with the page number loaded to. If On-Line had started with the high page number, instead of the low page number and read the sectors in decrementing order instead of incrementing order, the load would have taken a forth of the time. (Providing they use normal DOS 3.3 skewing). This is the logic that the fast loading dos's have taken to increase loading and saving time.

Appendix Q.

Deprotecting Photar from Softape.

Photar first seemed unprotected and on normal DOS 3.3. After running COPYA on the disk, I got an I/O error on track 4.

Using "The Inspector" from Omega, I nibble read track 4 and discovered that the address field epilogue bytes had been modified on the original disk from the normal $DE AA EB to $DE AA E8. This is why COPYA died on track 4. Track 4 seemed to be the only track with this modification.

To understand what Softape is doing you must know the structure of a DOS 3.3 disk. Each sector of a DOS disk has an address field which tells DOS what volume, track, sector and checksum it is reading. The address field starts with the unique bytes $D5 AA 96 (called the prologue bytes). Next comes the volume number, track, sector and checksum. Finally, there is a set of three unique byte (called the epilogue bytes) that end the address field. These are normally $DE AA EB. If these are changed, normal DOS will not be able to verify the address field and hence you get a I/O error.

Another interesting thing about Photar is that it is a 16K slave disk! What this means is that DOS is loaded at $1600 to $4000 instead of $9600 to $BFFF as a 48K slave disk. It then relocates itself to the normal $9600-$BFFF area. (Regardless of what your DOS manual says, a slave disk will self-relocate its DOS to the highest available memory, just like a Master disk does. So there is never any need to use a Master disk.)

After the DOS relocation, there is a JMP to $1A00. The code looks like this (formatted in 80 columns):

```
1A00-   A0 01      LDY   #$01   :change epilogue bytes to
1A02-   20 93 1A   JSR   $1A93  :$DE AA E8 to read trk 4.
1A05-   A2 10      LDX   #$10   :number of sectors to load.
1A07-   A0 08      LDY   #$08   :page to start loading to.
1A09-   A9 04      LDA   #$04   :track to start reading at.
1A0B-   20 4A 1A   JSR   $1A4A  :do the read.
1A0E-   B0 F5      BCS   $1A05  :branch back if error occurred.
1A10-   A0 00      LDY   #$00   :change epilogue bytes back
1A12-   20 93 1A   JSR   $1A93  :to normal $DE AA EB.
1A15-   A2 70      LDX   #$70   :number of sectors to load.
1A17-   A0 20      LDY   #$20   :page to start loading to.
1A19-   A9 05      LDA   #$05   :track to start reading at.
1A1B-   20 4A 1A   JSR   $1A4A  :do the read.
1A1E-   B0 F5      BCS   $1A15  :branch back if error occurred.
1A20-   AD 50 C0   LDA   $C050  :turn on hi-res page 1
1A23-   AD 52 C0   LDA   $C052  :and start game.
1A26-   AD 57 C0   LDA   $C057
1A29-   AD 54 C0   LDA   $C054
.
.
```

.

This code at $1A00 was where DOS was originally loaded, and if you read track $00, sector $0A with a sector editor, you will find this code. The code reads track 4 into $800 to $18FF and then reads track 5 and up into $2000 to $8FFF.

What we need to do it to let Photar read in the game for us, and stop it before it runs away from us and starts the game. This way we will have all of Photar in memory and we can save it to a normal DOS 3.3 disk. (We cannot just let the game start and use $1A20 as a starting address since Photar spreads memory out and uses the text page for valid routines).

So all we need to do is boot Photar, and after we see the BASIC prompt, hit reset. Now just enter the monitor with CALL -151 and change $1A20 to jump to monitor and break. Now we just execute the code at $1A00 and let photar load in the game for us. After the load is done and before the game starts, we will be in the monitor. Now just boot a 16K slave disk and Bsave Photar!

In cook book fashion, here are the steps:

1) Boot Photar.

2) After the BASIC prompt appears and before the game starts, hit reset.

3) Enter the monitor by typing:

```
]CALL -151
```

4) Set the break point so the game loads but does not start. Type:

```
*1A20:4C 59 FF N 1A00G
```

5) Reset the changed bytes to what they were. type:

```
*1A20:AD 50 C0
```

6) Move $800 out of the way for a 48K slave disk boot. Type:

```
*9000<800.A00M
```

7) Boot a 48k slave disk.

8) Enter the monitor by typing:

```
]CALL -151
```

9) Move $800 back down to its proper place. Type:

```
*800<9000.91FFM
```

10) So the game runs when BLOADed type:

```
*7FD:4C 20 1A
```

11) BSAVE the game by typing:

```
*A964:FF
*BSAVE PHOTAR,A$7FD,L$7F1F
```

Appendix R.

Deprotecting Bug Attack from Cavalier Computer.

The protection used in Bug Attack is well done and challenging to remove. Boot code tracing the disk will give you a headache real fast because of extensive use of page $02 (the keyboard buffer). Also, upon reseting from the game, screen memory gets destroyed, which is necessary memory for the game to run correctly. (The text page code at $400 to $7FF is shape tables for all the characters and explosions in Bug Attack.)

What we need is some way to reset out of the game and to move memory from $200 to $7FF up to $2000 to $25FF, or somewhere else safe. This can be done if you have some sort of a modified F8 monitor ROM, but not everyone has this tool.

Also, after we have solved this first problem of saving $200 to $7FF, we must be able to move it back to its original location when we re-run the game. We really have no easy way of re-loading $200 to $7FF with the original memory after saving it to a normal DOS disk as a binary file. This is because $200 to $2FF is the keyboard buffer, and anything that is typed, or DOS command executed from BASIC, will destroy some of $200 to $2FF.

This makes Bug Attack a little more challenging than many kraks. And to get around these problems, some ASSEMBLY language programming will have to be used. But even if you don't know ASSEMBLY language, don't be scared off. I will hold your hand through this one.

Since boot code tracing seems to be a futile effort (or more timely than I am willing to deal with), I decided to look at the loader after the game is loaded. Cavalier uses a fairly short loader to bring their game to life, and it lives at $A200. It was not hard to find it using the Inspector's Find command to look for the string "AD 50 C0", which turns on the hi-res page (remember, the hi-res page gets turned on immediately upon booting the disk). Also, if you search for the string "2C 50 AD", which is another way to turn on the hi-res page, the starting address of Bug Attack is given away at $4D36.

Now that we have found the loader, we want to modify it to read in Bug Attack at $900, instead of $200. This way memory from $200 to $7FF is loaded in a safe region allowing us to save it to a normal DOS disk.

The byte to change in the loader is at $A255, which normally contains a $02, to start loading at $200. We want to change this byte to $09, to start loading at $900. The command $A200G will start the drive up and will load Bug Attack in starting at $900. Obviously, the game will not run, so after the drive stops, hit reset.

NOTE: To get the loader set up with the right parameters and to start loading from the correct track (track $03), you have to be careful when you press reset. First, boot the game and after it is loaded and running, press reset. If you have an old style F8 monitor, type 6 CTRL P to start the drive again. Now, after the drive recalibrates, it will wait a second and then turn on the hi-res page and the drive head will seek to track $03 and start reading in the game. EJust before the hi-res page turns on and the head seeks to track $03, press reset (twice if you have a new style F8 monitor)F. Now you have captured the loader with the correct parameters at $A200. This will take a little practice, but just keep hitting reset until you feel you have caught the loader at the right time.

Now that we have $200 to $7FF captured at $900 to $EFF, boot a 48K DOS slave disk and save $200 to $8FF to disk. Type:

```
]BSAVE ^02-08,A$900,L$600
]BSAVE ^08-09,A$F00,L$100
```

Now we can re-boot Bug Attack and after the drive stops, put your 48k slave disk in the drive. After the initial explosion is done, press reset and boot your slave disk. Now type:

```
]BLOAD ^08-09,A$800
]BLOAD ^02-08,A$8000.
```

You now have all of Bug Attack in memory. The problem is how do we get $8000 to $85FF back to $200 to $7FF? Also, we need an image of the hi-res title picture at $8000 to $9FFF. This is because the picture gets re-drawn after each game, and is stored at $8000 to $9FFF. This is getting complicated, isn't it?

We don't want a duplicate of the memory at $8000 to $9FFF saved to disk when we already have it at $2000 to $3FFF, and we need memory at $8000 to $85FF moved to $200 to $7FF. This is where the ASSEMBLY language comes in. We can write a small routine to first move $8000 to $85FF down to $200 to $7FF, and then move $2000 to $3FFF up to $8000 to $9FFF. Finally, we can jump to the staring address of Bug Attack at $4D36. A good place to put this routine is at $1A00, since there is an empty hole in the Bug Attack code. Here is the code we need (formatted in 80 columns):

```
1A00-   A2 00       LDX   #$00    :load X with $00.
1A02-   BD 00 80    LDA   $8000,X:load accum with $8000 indexed by X.
1A05-   9D 00 02    STA   $0200,X:store accum at $200 indexed by X.
1A08-   E8          INX           :increment X.
1A09-   D0 F7       BNE   $1A02   :if X not $FF, go back to $1A02.
1A0B-   EE 04 1A    INC   $1A04   :else increment page to load at.
1A0E-   EE 07 1A    INC   $1A07   :increment page to store at.
1A11-   AD 07 1A    LDA   $1A07   :load accum with page to store at.
1A14-   C9 08       CMP   #$08    :compare accum to $08.
1A16-   D0 E8       BNE   $1A00   :if less than $08, goto $1A00.
1A18-   A2 00       LDX   #$00    :-----------------------------------
```

```
1A1A-   BD 00 20    LDA   $2000,X: do the same thing as above
1A1D-   9D 00 80    STA   $8000,X: but move memory from $2000
1A20-   E8          INX          : to $3FFF up to $8000 to
1A21-   D0 F7       BNE   $1A1A  : $9FFF.
1A23-   EE 1C 1A    INC   $1A1C  :
1A26-   EE 1F 1A    INC   $1A1F  :
1A29-   AD 1F 1A    LDA   $1A1F  :
1A2C-   C9 A0       CMP   #$A0   :
1A2E-   D0 E7       BNE   $1A17  :---------------------------------
1A30-   4C 36 4D    JMP   $4D36  :jump to the beginning of the game.
```

The code here is what you call "self modifying" code because its modifies itself as it runs by changing the pages to load and to save at. It is generally not good practice to write self modifying code, but in this case, it is the shortest way to write these memory moves. Besides, only real men write self-modifying code.

Before we save the game to our disk, we must remove the disk access during the game. This disk access code checks for your original disk by reading the present disk in drive one and checking the address field epilogue bytes to make sure they are the (modified) ones on the original Bug Attack disk. This "nibble count" routine lives at $4A33. The logical way to defeat this would be to skip across the routine, or to put a RTS (return to subroutine) at the beginning of it. The problem with this is that the program later check to see if the nibble count code has been altered. If it has, the program dies. To solve this, we can void any calls to the nibble count routine, thus not changing any of the actual code in the routine.

So to defeat the nibble count routine, type:

```
*4E05:4C A4 49
```

Also, to make the game run when it is BRUN, type:

```
*7FD:4C 00 1A.
```

Last, type:

```
*A964:FF
```

so you can save more than 120 sectors in one file. Now you may save the game to your DOS disk by typing:

```
*BSAVE BUG ATTACK,A$7FD,L$7E03.
```

In cookbook form, here are the steps to deprotecting Bug Attack:

1) Boot Bug Attack and after it is loaded, hit reset. (If you have a old style F8 monitor, type 6 CTRL P after reset).

2) Just after the drive head recalibrates and before the hi-res

page is turned on, press reset again (twice if you have a new style F8 monitor).

3) Enter the monitor (if you are not already in it) by typing:

```
]CALL -151
```

4) Type:

```
*A255:09 N A200G
```

5) After the drive stops, press reset.

6) Boot a 48K normal DOS 3.3 slave disk.

7) Type:

```
]BSAVE ^02-08,A$900,L$600
]BSAVE ^08-09,A$F00,L$100
```

8) Boot Bug Attack and press reset after the initial explosion is done.

9) Boot your 48K normal DOS slave disk.

10) Type:

```
]BLOAD ^08-09,A$800
]BLOAD ^02-08,A$8000
```

11) Enter the monitor by typing:

```
]CALL -151
```

12) Enter these bytes from the monitor by typing:

```
*A964:FF
*7FD:4C 00 1A
*4E05:4C A4 49
```

13) Enter the memory moves at $1A00 by typing:

```
*1A00:A2 00 BD 00 80 9D 00 02
*1A08:E8 D0 F7 EE 04 1A EE 07
*1A10:1A AD 07 1A C9 08 D0 E8
*1A18:A2 00 BD 00 20 9D 00 80
*1A20:E8 D0 F7 EE 1C 1A EE 1F
*1A28:1A AD 1F 1A C9 A0 D0 E8 4C 36 4D
```

14) Carefully check that you have entered the code correctly by typing:

```
*1A00L
```

15) BSAVE the game by typing:

```
]BSAVE BUG ATTACK,A$7FD,L$7E03
```

To recap, we have solved two problems that have not really been address before: How do we save "volatile" memory areas (such as the keyboard buffer and the text page), and, how do we move these sections of code back to where they belong. I hope I have given you some idea how these two things can be done.

Appendix S.

Deprotecting Flip out.

After booting Flip-out, the first key to press is reset. But something unusually happened: I did not see the usually monitor prompt (I have an old style F8 monitor ROM on the motherboard). My computer rebooted as if I had a new style F8 monitor ROM! This intrigued me into investigating this strange phenomenon.

What I discovered was that in the main program (not the boot code) Sirius had turned on my 16K RAM card and copied an image of the new style F8 monitor ROM into my RAM card. Of course, they left the RAM card turned on instead of the motherboard ROMs. It is easy to understand what they are doing if you just remember what memory the RAM card occupies.

The RAM card encompasses memory from $D000 to $FFFF. This may seem strange since motherboard ROM (APPLESOFT and the monitor) also ranges from $D000 to $FFFF. But there is a set of soft switches that will turn on motherboard ROM, or turn on the RAM card. A demonstration of this is when you boot your 48k DOS 3.3 system master and it loads INTEGER basic into the RAM card. Now you have two languages available that occupy the same logical memory space, $D000 to $FFFF. You can switch between the two with INT and FP. What these commands are doing is switching between the motherboard ROMs and the RAM card's memory.

Well, if you read your RAM card manual you can see that the softswith $C080 (assuming your RAM card is in slot zero) will allow you to look at your RAM card's memory. But have you ever tried typing $C080 from monitor? It will lock your Apple up so only powering off and on will recover!

The reason is that you switch to the RAM card's memory $D000-$FFFF and turn off the motherboard's ROMs. When you do this, the computer loses control since there is no longer a monitor ROM available from $F800-$FFFF to overlook your Apple's operations!

The solution is to type C081 C081 N F800<F800.FFFFM from monitor. This reads the motherboard's ROM but allows you to write to bank 2 of the RAM card. The memory move F800<F800.FFFFM moves the F8 motherboard ROM into memory $F800-FFFF in bank 2 of the RAM card! Now you may type C080 to turn on the RAM card and look at its memory, since a copy of the $F8 monitor ROM is in the RAM card from $F800 to $FFFF.

Sirius has done exactly this so that no matter what $F8 ROM you have in the motherboard, your Apple will only look at the new style $F8 ROM image in your RAM card. Thus the Apple clears memory and reboots when reset is pushed! This is an easy problem to fix, now that we have identified it. Just take your RAM card out of your computer and you may reset into the monitor as usual.

Well anyway, we may now move onward in the quest for deprotection of Flip out.

Flip out is a single load program. To deprotect single load programs, you must be able to determine three basic things:

1) What memory is used by the program.

2) The starting address of the program.

3) How to get the memory saved to a normal DOS 3.3 disk and reloaded back into memory in the proper place(s).

Keep these in mind as we trace through Flip out (or any other single load game).

Sirius has changed their protection schemes a lot in the last few years. The height of their hard-core protection was demonstrated in games like "Bandits" and "Fly-wars". The problem with hi-tech protection is that it might not boot on a Rana drive, or on a //e or someother flavor of Apple. Also, protection costs money, and the less of it you use, the less money it costs to publish a game. It seems that Sirius has gotten smart and figured out that they really weren't losing as much money as they thought to "pirates", at least not as much as a sophisticated protection schemes costs!

In light of this, Sirius has chosen a much simpler, but still effective, copy protection scheme for Flip out. Just try and copy it with Nibbles away and you'll see just how effective it is!

Keeping in mind the three things we must figure out to deprotect a single load game, the first thing I generally do is to find what memory is required to run the program. To do this we can flip through memory (keep in mind that shape tables, etc. don't disassemble into meaniful code), or we can trace the boot and see where the program gets loaded to: I prefer to trace the boot when the boot is fairly simple, which Flip out is.

So boot up normal DOS 3.3 (so we can save piece of code for later examination, if you wish) and enter the monitor with CALL-151. Now we must copy the code in the disk controller ROM down to RAM so we can modify it to our liking (remember you can't change code in ROM, that's why they call it Read Only Memory). Do this with 8600<C600.C6FFM from the monitor. Now we have the disk controller ROM code where we can modify it, and start to trace the boot.

What the disk controller ROM does it to read track zero, sector zero into $800 to $8FF of memory. Then it JMPs to $801 and executes the code as $801 (which continues by loading in a little more code, which then loads in more code, which then.....well, you get the idea).

So at the end of the code at $8600 we see a JMP $0801. We must change this to JMP $FF59, which we'll exit us in the monitor after it is done loading track zero, sector zero into $800 to $8FF. So put in Flip out in drive one and type 86F9:59 FF N 8600G to execute the code. The drive will recalibrate and a second later beep into the monitor, just like we told it. To turn off the drive motor, type C0E8 from the monitor prompt. Now, if you want you can save this hunk of code to your normal DOS 3.3 disk with BSAVE BOOT0,A$800,L$100 (since this process did not disturb DOS which lives from $9600 to $BFFF).

Now type 801L to flip through the code just loaded in and we will find this:

```
801-   A5 2B      LDA    $2B
803-   AA         TAX
804-   85 FB      STA    $FB
806-   4A         LSR
807-   4A         LSR
808-   4A         LSR
809-   4A         LSR
80A-   09 C0      ORA    #$C0
80C-   8D 00 30   STA    $3000   ----------
80F-   A0 00      LDY    #$00    :
811-   84 00      STY    $00     :
813-   A9 D0      LDA    #$D0    :
815-   85 01      STA    $01     ;Destroy
817-   A2 30      LDX    #$30    ;anything
819-   AD 81 C0   LDA    $C081   ;in the
81C-   AD 81 C0   LDA    $C081   ;RAM card.
81F-   B1 00      LDA    ($00),Y:
821-   91 00      STA    ($00),Y:
823-   C8         INY            :
824-   D0 F9      BNE    $081F   :
826-   E6 01      INC    $01     :
828-   CA         DEX            :
829-   D0 F4      BNE    $081F   :
82B-   A6 FB      LDX    $FB     :---------
82D-   84 F7      STY    $F7     :
82F-   A9 04      LDA    #$04    ;Page to
831-   85 F8      STA    $F8     ;load at
833-   85 FA      STA    $FA     :
835-   BD 8C 8C   LDA    $C08C,X:read disk
838-   10 FB      BPL    $0835   :
83A-   C9 AD      CMP    #$AD    :
83C-   D0 F7      BNE    $0835   :
83E-   BD 8C C0   LDA    $C08C,X;Make sure
841-   10 FB      BPL    $083E   ;datafield
843-   C9 DA      CMP    #$DA    ;epilogue
845-   D0 F3      BNE    $083A   ;bytes are
847-   BD 8C C0   LDA    $C08C,X;$AD DA DD
84A-   10 FB      BPL    $0847   :
84C-   C9 DD      CMP    #$DD    :
84E-   D0 EA      BNE    $083A   ----------
```

```
.
.
.
88A-  4C 29 04  JMP   $0429  :JMP boot2
```

This code loads in the final loader (boot2) over the text screen memory ($400 to $7FF) and JMPs to $429 to load in the actually game. Now we need to see boot2 (the game loader) to see where it is actually loading in the game.

Well, as you can see this is slightly difficult since boot2 gets loaded over the text page, and when we hit reset, this memory pretty much hits the bit-bucket. But boot1 (the code that loads in boot2 which we are now looking at in $801-$88C) can be changed to load boot2 somewhere else and gracefully reset into monitor. To do this, change the load byte from page $04 to page $14, and change the JMP $429 to jump into the monitor. Then we can examine the boot2 loader. To do this enter 830:14 N 88B:59 FF from the monitor.

The next thing we must change is the disk controller ROM code at $8600. We need it to execute but not write over the modified code at $801. To do this we can tell it to load track zero sector zero at $6000 (instead of $800) and jump to our modified code at $801. Of course our code will load in boot2 into $1400 so we can look at it. Are you confused yet? Go back if you are and don't come back till you understand what is going on.

OK, put Flip out in drive one and type

```
8659:60 N 86F9:01 08 N 8600G.
```

The drive will recalibrate and and boot zero will read track zero sector zero into $6000 (thus not overwriting our code at $801). It will then jump to $801 (boot1) and load boot2 into $1400 to $17FF. Unfortunately, it will keep reading boot2 into $1400 to $17FF because we haven't change enough code, so after a few seconds, hit reset.

Now if you want you can put your normal DOS 3.3 disk in a drive and save boot2 with BSAVE BOOT2,A$1400,L$400.

Next type 1429L and examine boot2. You will notice that memory from $800 to $BFF gets wiped clean and that a reset error routine gets moved to $8F00-$8F80. This is a good indication that Flip-out lives from $C00 to $8F80! I'll let you sort through the boot2 code to find out for sure, or you can take my word for it.

The last tidbit of information that the boot2 loader reveals is the starting location of Flip out. Look at the code at $17CC to $17E4 and you'll see how $800 to $BFF gets wiped and then JMPs to $7800 to start the game.

Now we have filled requirements one and two. All that is left is to save the memory from $C00 to $8F80 on a normal DOS 3.3 disk. This is easy since a 48K slave disk does not destroy memory from $900 to $96FF. So just boot Flip out, and when the drive stops, reset into the monitor. Now boot your 48k slave disk and save Flip out to disk!

In cook-book fashion:

0) TURN YOUR APPLE OFF and remove your RAM card.

1) Boot Flip out.

2) After the drive stops and Flip out is loaded into memory, reset into the monitor.

3) Boot a 48k slave disk.

4) Enter the monitor by typing:

```
]CALL-151
```

5) So the program executes when brun, type:

```
*BFD:4C 00 78
```

6) Bsave flip out by typing:

```
*A964:FF
*BSAVE FLIP-OUT,A$BFD,L$8383
```

If you want the title page also, do the above and:

1) Reboot the flip-out disk and reset into the monitor when you see the title page.

2) Boot your 48k slave disk and save the picture by typing:

```
]BSAVE PIC,A$2000,L$1FFB.
```

3) Bload the Flip out file by typing:

```
]BLOAD FLIP-OUT
```

4) Bload the picture file by typing:

```
]BLOAD PIC,A$2000
```

5) Enter the monitor by typing:

```
]CALL-151
```

6) Change these bytes by typing:

```
*BE0:AD 10 C0 AD 50 C0 AD 54
```

```
*BE8:C0 AD 57 C0 AD 52 C0 AD
*BF0:00 C0 10 FB 4C 00 78
```

7) Bsave the file by typing:

```
*A964:FF
*BSAVE FLIP-OUT,A$BE0,L$83A0
```

-THE DISK JOCKEY-

7) Bsave the file by typing:

```
*A964:FF
*BSAVE FLIP-OUT,A$BE0,L$83A0
```

Appendix T.

Deprotecting Guardian from Continental Software.

Continental Software uses a fast loader that lives over hi-res page one (memory from $2000-$4000) to load Guardian. Because of this, we do not have to save memory across hi-res page one, but we have to catch the game just before it starts to draw the hi-res title page.

Because of this, it is important to hit reset at the correct time. (It is comforting to know that we do not need an old style monitor in deprotecting Guardian, since they do not use a clear memory reset routine. This is convenient for those with //e's.). The technique to use is to boot the game and allow it to load and run. Next hit reset and allow it to re-load. Now we must hit reset at the correct time. Listen to the game load a few times and time it so you can hit reset just before the hi-res message starts drawing on the screen. If you have a Apple //e or II+ with the auto start F8 monitor, hit reset twice to break out of Guardian.

After you have mastered this, we just boot a 48K slave disk and save memory from $4000 to $8E00 to disk. To start the game, we just do a JSR $7C19 (to erase hi-res page one), turn hi-res page one on, and do a JMP $592E to start the game. With this is mind, here are the steps to deprotecting Guardian:

1) Boot Guardian and let the game load and run. Hit reset and boot the game again. Just before the hi-res page starts drawing, hit reset again to break out of the program (if you have an autostart F8 monitor, hit reset twice).

2) Boot a 48K slave disk.

3) Enter the monitor by typing:

```
]CALL -151
```

4) Enter these bytes to start the game when BRUN by typing:

```
*3FFD:4C 00 8E
*8E00:20 19 7C AD 10 C0 AD 50
*8E08:C0 AD 54 C0 AD 57 C0 AD
*8E10:52 C0 4C 2E 59
```

5) Save the game to disk by typing:

```
*BSAVE GUARDIAN,A$3FFD,L$4E19
```

Appendix U.

Kraking Mating Zone from Datamost: an example of defeating minor disk access from a single load program.

Kraking Mating Zone into a file is a challenging job. The reason is that Mating Zone uses some disk access that reads in some shapes for display between levels. The hard part about defeating this disk access is that we must have everything just right for it to work, which means we have to do some ASSEMBLY language programming to make it work.

Mating Zone's primary protection is using modified address and data prologue bytes. In addition, Mating Zone uses half tracks on tracks $10.5 - $13.5. But we really don't care too much about that since we are kraking it into a single file.

The starting address was not too hard to find at $C71. After reseting into the monitor I just searched for the bytes "AD 50 C0" with the Inspector. This revealed three possible locations. The one at $C71 was a dead giveaway though because it referenced all the hi-res commands (turn on graphics, turn on hi-res graphics, use page 1, use full screen graphics). Plus, it was right before a huge "jump table", which really gave it away. And for the final test, the game works fine if you type "C71G" to restart it.

The memory needed in kraking Mating Zone is from $800-$1FFF and from $6000-$AFFF. We do not need the two hi-res pages ($2000-$5FFF) since the game uses them to draw on and for a shape buffer. I figured this out by wiping out portions of memory and then re-starting the game. If the game did not work, I knew I needed the memory range I wiped out (use the monitor's move command to wipe out a memory range. I.E. to wipe out $B000-BFFF you would type: *B000:0 N B001<B000.BFFFM).

The problem with Mating Zone is the disk access used for the shapes between levels. If you look carefully, you can find the loader lives at $400-$7FF. A very inconvenient place for the krakist, since it gets wiped out when we hit reset (notice all the screen "garbage" when you hit reset).

Upon further examination and experimentation I found we could move a series of 60's across $400-$7FF ($60 represents a return from subroutine) and this would disable the disk access between levels. This was considerably easier than trying to find all the locations in main memory that called the loader (you probably won't be successful anyways).

In addition if hi-res page 1 and 2 are not clear at the start of the game, much hi-res crap is encountered when the disk access should occur. So for an added chore, we must clear $2000-$5FFF before starting the game for our disabled disk access to work properly.

So with this in mind, it is best we get started saving all the bits of memory needed to a 48k slave disk. So first, let's get the hi-res picture that is first displayed when the game is booted. We can display this before starting the game for added appeal (yes, we must take pride in our work).

To do this, boot Mating Zone and reset into the monitor when the initial hi-res pages are flashing, Now boot your 48K slave disk and save the pic by typing:

```
]BSAVE PIC,A$4000,L$2000
```

Now reboot Mating Zone and reset into the monitor after the disk stops and the game is running.

Next we have to move memory around so we can boot a slave disk and save it. Remember we need $800-$1FFF and $6000-$AFFF, and $9600-$AFFF and $800-$8FF will get destroyed by a slave disk boot. Type:

```
*4000<9600.AFFFM
$5A00<800.900M
```

Now we can boot a 48K slave disk and re-arrange memory and save it to disk. Do this after booting the slave disk by typing:

```
]BLOAD PIC,A$2000
]CALL-151
*800<5A00.5AFFM
```

Finally, we have to write a memory move at $5A00 that will:

1) Move $4000-$5AFF to $9600-$B0FF (notice we move the memory move itself too! This is because we have to wipe out $2000-$5FFF eventually, which is where the memory move initially lives).

2) JMP to $B01B and continue.

3) Turn on hi-res page 1 (to display our hi-res title page).

4) Clear the text screen with $60's (to disable the disk access).

5) Wait for a keypress to start the game.

6) Clear hi-res pages 1 and 2.

7) Clear the keyboard buffer.

8) JMP to the start address at $C71 to start the game.

The routine should look like this:

```
5A00-   A2 00       LDX   #$00     :-----
5A02-   BD 00 40    LDA   $4000,X:move
5A05-   9D 00 96    STA   $9600,X:$4000
5A08-   E8          INX            :-5AFF
5A09-   D0 F7       BNE   $5A02    :to
5A0B-   EE 04 5A    INC   $5A04    :$9600
5A0E-   EE 07 5A    INC   $5A07    :
5A11-   AD 04 5A    LDA   $5A04    :
5A14-   C9 5B       CMP   #$5B     :
5A16-   D0 E8       BNE   $5A00    :-----
5A18-   4C 1B B0    JMP   $B01B    :JMP
5A1B-   AD 57 C0    LDA   $C057    :-----
5A1E-   AD 54 C0    LDA   $C054    :turn
5A21-   AD 52 C0    LDA   $C052    :on
5A24-   AD 50 C0    LDA   $C050    :hires
5A27-   A2 00       LDX   #$00     :-----
5A29-   A9 60       LDA   #$60     :move
5A2B-   9D 00 04    STA   $0400,X:$60's
5A2E-   E8          INX            :over
5A2F-   D0 F8       BNE   $5A29    :text
5A31-   EE 2D B0    INC   $B02D    :page.
5A34-   AD 2D B0    LDA   $B02D    :
5A37-   C9 08       CMP   #$08     :
5A39-   D0 EC       BNE   $5A27    :-----
5A3B-   AD 10 C0    LDA   $C010    :wait
5A3E-   AD 00 C0    LDA   $C000    :for
5A41-   10 FB       BPL   $5A3E    :key
5A43-   A2 00       LDX   #$00     :-----
5A45-   A9 00       LDA   #$00     :clear
5A47-   9D 00 20    STA   $2000,X:$2000
5A4A-   E8          INX            :-5FFF
5A4B-   D0 F8       BNE   $5A45    :
5A4D-   EE 49 B0    INC   $B049    :
5A50-   AD 49 B0    LDA   $B049    :
5A53-   C9 60       CMP   #$60     :
5A55-   D0 EC       BNE   $5A43    :-----
5A57-   AD 10 C0    LDA   $C010    :JMP
5A5A-   4C 71 0C    JMP   $0C71    :start
```

After entering this routine at $5A00, we can save the entire file to disk by typing:

```
*7FD:4C 00 5A
*A964:FF
*BSAVE MATING ZONE,A$7FD,L$8E04
```

In cookbook form, here are the steps:

1) Boot Mating Zone and reset into the monitor when the hi-res pages are flashing.

2) Boot a 48K slave disk and save the hi-res picture by typing:

```
]BSAVE PIC,A$4000,L$2000
```

3) Reboot Mating Zone and after the game is loaded and running, reset into the monitor.

4) Move memory for a slave disk boot by typing:

```
*4000<9600.AFFFM
*5A00<800.900M
```

5) Boot a 48K slave disk and rearrange memory by typing:

```
]CALL-151
*800<5A00.5AFFM
```

6) Enter the above memory move at $5A00.

7) Make the file execute when brun and let DOS save large files by typing:

```
*7FD:4C 00 5A
*A964:FF
```

8) Save the file to your disk by typing:

```
*BSAVE MATING ZONE,A$7FD,L$8E03
```

And your all done! Mating Zone is now kraked into a file with all levels.

Appendix V.

Kraking Crime Wave from Penguin Software.

Crime Wave is a typical single load protection game that is a bit difficult to krak without additional hardware. The reason for this is because Penguin uses an old trick of using the text page (memory from $400 to $7FF) for memory storage. This means when we hit reset, the whole text page memory is destroyed and shifted, making it impossible to save and reload the text page memory area to restart the game.

For a demonstration of this, boot the game and after it is loaded, reset into the monitor and type 803G to restart the game. Notice how the hi-res shapes and text are destroyed. Therefore we must find a way to save memory from $400 to $7FF in order to krak the game.

Usually, this means using additional hardware like a "krak ROM" or some other similar devise to save the text page. But in Crime Wave's case, there is another way!

If you page through memory you will find Crime Wave uses both of the hi-res pages (turn on the 2 pages and you can see the hi-res drawings). Now watch the game play, and you can see that the hi-res scenes are drawn. This is hint that we don't need to save $2000-$5FFF in our kraked version, so don't even bother looking at this memory range.

But At $A8A5, you will find this routine:

```
A8A5-   A9 20       LDA   #$20
A8A7-   85 7D       STA   $7D
A8A9-   A9 05       LDA   #$05
A8AB-   85 7F       STA   $7F
A8AD-   A0 00       LDY   #$00
A8AF-   84 7C       STY   $7C
A8B1-   84 7E       STY   $7E
A8B3-   A2 03       LDX   #$03
A8B5-   B1 7C       LDA   ($7C),Y
A8B7-   91 7E       STA   ($7E),Y
A8B9-   C8          INY
A8BA-   D0 F9       BNE   $A8B5
A8BC-   E6 7D       INC   $7D
A8BE-   E6 7F       INC   $7F
A8C0-   CA          DEX
A8C1-   D0 F2       BNE   $A8B5
A8C3-   60          RTS
```

This routine moves memory from $2000-$22FF to $500-$7FF. This must mean that before the hi-res pictures are drawn at $2000-$5FFF and the game is started that $2000-$22FF is moved across the text page.

So we need to stop the program after is has loaded the text page data into $2000-$2FFF and save this memory portion, before the game starts. To do this just boot the game and listen to the disk drive read the tracks. After you hear the third "click" or track, hit reset. Now boot a normal DOS slave disk and save $2000-$22FF.

To get the rest of the program, reboot Crime Wave and after the program is done loading and is running, reset into the monitor. Now move memory for a slave disk boot and save $800-$1FFF and $6000-BFFF.

Since Penguin uses the memory above $9600, we need to use a memory move routine to put this memory back after we load our normal DOS bfile. We can arrange memory so we have $500-$7FF at $2000-$22FF, and have $8400-$BFFF at $2300-$5EFF. Then we can have our memory move routine run at $5F00. This will make the bfile as small as possible.

So after you have saved $500-$7FF (at $2000-$22FF), reboot Crime Wave and reset into the monitor. Now we must move memory for a slave disk boot (that will not disturb $900-$95FF, remember?). So, move $8400-$BFFF down to $2300 and move $800-$8FF to $8400 by typing:

```
*EB5:60
```

```
*9BAB:EA EA EA
*2300<8400.BFFFM
*8400<800.900M
```

(NOTE: the changes at $EB5 and $9BAB disable Penguin's minor disk access).

Now boot your 48K slave disk and re-arrange memory and save it to disk:

```
]CALL-151
*800<8400.84FFM
*BLOAD ^$2000-22FF,A$2000
*BSAVE ^800-8400,A$800,L$7C00
```

Now run the memory writer program and create a memory move that will run at $5F00 and will move $2300-$6000 to $8400. Set the display pages for hi-res page one and jump to $0803. Now to complete our krak, do the following:

```
]BLOAD ^800-8400
]BLOAD MEMORY MOVE$5F00,A$5F00
]CALL-151
*7FD:4C 00 5F
*5F18:20 A5 A8 4C 03 08
*BSAVE CRIME WAVE,A$7FD,L$7C03
```

Notice we add 3 bytes at $7FD so the file jumps to our memory move routine at $5F00 when BRUN. Also notice at the end of the memory move we add a call to the routine that moves $2000-$2300 to $500-$7FF and then we jump to the beginning of the game at $803.

So to summarize, here are the steps:

1) Boot Crime Wave and after 3 "click" of the disk drive, reset into the monitor.

2) Boot a 48K slave disk and save $2000-$22FF by typing:

```
]BSAVE ^$2000-$22FF,A$2000,L$300
```

3) Now reboot Crime Wave and reset into the monitor after the game is done loading. Now type:

```
*EB5:60
*9BAB:EA EA EA
*2300<8400.BFFFM
*8400<800.900m
```

4) Boot your 48K slave disk and rearrange memory and save it to disk. Type:

```
]CALL-151
```

```
*800<8400.84FFM
*BLOAD ^$2000-$22FF
*BSAVE ^$800-$8400,A$800,L$7C00
```

5) Now brun the memory move writer program and enter the following:

```
NUMBER OF MEMORY MOVES: 1
RUNNING ADDRESS: $5F00
START PAGE: $23
END PAGE: $5F
DESTINATION PAGE: $84
HI-RES PAGE 1, FULL MODE
JUMP AFTER MOVE TO: $0803
```

6) Now reload the main memory and the memory move and save the file by typing:

```
]BLOAD ^$800-$8400
]BLOAD MEMORY MOVE$5F00,A$5F00
]CALL-151
*7FD:4C 00 5F
*5F18:20 A5 A8 4C 03 08
*BSAVE CRIME WAVE,A$7FD,L$7C03
```

And your all done!

Appendix W.

Kraking Roundabout into a single file: Using a memory packer.

Assuming that you have read the file on kraking Roundabout into a full disk version, here is a method you could use to krak it into a file using a memory packer. Without using a memory packer, there is too much memory for us to save it into a single file. With a memory packer, Roundabout can be put into a single 125 sector file.

But keep in mind that Roundabout can be kraked into a single file only if we give up all the different shapes used in the game for each level. If we put it into a single file, the game will now have 6 different shapes used for each one of the levels. But the real reason I am kraking it into a single file is to show how a memory packer could be used in a single file krak.

Roundabout uses memory from $800 to $B2FF, for a total of $AB pages, which is greater than $8E pages that we can save a file under normal DOS with (at least without using some sort of second stage DOS loader). Therefore we must find some way to make the memory we are saving to disk smaller. We can do this by using a memory packer, as described in chapter 8.

First we need to find a starting address for Roundabout. This is not too difficult. I started checking the even address pages, such as $800, 900, etc. and found the starting address at $7000 (of course don't look for a starting address in the hi-res memory area if the program uses one or both of the hi-res pages).

After doing this, I found that Roundabout needed memory from $800 to $B2FF to run. To find this out I erased a memory area and tried running the game to see if it worked (if you were smart you would just look at the loader and see what it loaded in, but I didn't). I also found we needed to save hi-res page 2 since that was loaded in from disk and not drawn from within the game.

In addition, I had to remove the disk access that loads in the new shapes for each level. The routine that does this uses second stage DOS to load the shapes, so we only need to disable RWTS ($B700-$BFFF). We could fill this memory area with the string "18 60", as described in the chapter on disabling disk access, but I managed to find the calling routine and found it calls RWTS at $BD00. So all we need to do is to put a "60" at location $BD00, and the program will return to the calling routine without doing any disk access.

Now we have to save $800-$B2FF to a slave disk in parts. The best way to do this is to save the hi-res picture in one pass, and try and save the rest of memory in another pass. The reason is simple: saving the actual routines in one pass is a good idea since you don't want to save two pieces that were not running together at one time. The hi-res page doesn't have any routines on it ever, so we can save it separate of the running routines.

So with this in mind, boot Roundabout and when the disk drive has stopped and the game is running, reset into the monitor. Now move memory around for a slave disk boot by typing:

```
*4000<9600.B300M
*5D00<800.900M
```

Now we can boot a slave disk, move $800-$8FF back and save these memory portions by typing:

```
]CALL-151
*800<5D00.5DFFM
*BSAVE ^96-B3,A$4000,L$1D00
*BSAVE ^08-40,A$800,L$3800
*BSAVE ^60-96,A$6000,L$3600
```

The only memory portion left is hi-res page two ($4000-$5FFF). This is easy to get. Just boot the Roundabout disk, and watch as the game loads. You will see the hi-res page get loaded in. So hit reset just after the hi-res page is loaded and before the drive stops and the game starts. This way you catch the hi-res page before the game starts and any shapes are drawn on it.

After you hit reset with the hi-res page in memory, boot your 48k slave disk and type:

```
]BSAVE ^40-60,A$4000,L$2000
```

Now we must put some of the memory pieces together so we can get ready to pack them. To do this type:

```
]BLOAD ^08-40
]BLOAD ^40-60
]BLOAD ^60-96
]BLOAD SHOWBIN
```

Using SHOWBIN, we can find the unused flag byte for the memory range we are going to pack (refer to chapter 8 for a description of the packing routines and how they work). But first we must figure out what memory range we want to pack.

I decided memory from $800 to $8FFF would be good to pack, since the memory above that was not repetitive, and therefore would not benefit from packing anyways. So now we must enter the parameters for SHOWBIN:

```
]CALL-151
*00:08 8F
*300G
```

NOTE that I entered a "08" and a "8F" so SHOWBIN would look at memory from $800 to $8FFF. The "300G" starts the SHOWBIN program.

The first set of bytes are the bytes that are never used in that memory range. So pick one of them for your flag byte and write it down. I choose $6A. Hit ESCAPE to exit SHOWBIN

Now bload PACKER so we can pack our memory range. Do this by typing:

```
*BLOAD PACKER
*05:6A 08 8F
*300G
```

Note that we enter the flag byte at $05, the low page at $06 and the hi page at $07. The "300G" executes PACKER and packs our memory range. Now lets see how small we packed it. Type:

```
*00.01
```

And the computer responds with:

```
0000 - E8 39
```

This means that $800 to $8FFF packed down to $39E8 to $8FFF. So we should save the packed memory to disk by typing:

```
*BSAVE ^08-8F.PAC,A$39E8,L$5C19
```

This means we now have room to put $9600 to $AFFF at $1C00, beneath the packed memory, and save Roundabout to disk in a 125 sector file. Then to reconstruct Roundabout, we need to write a memory move routine that will run at $3900 (just below the packed memory to conserve the most memory, of course) to do the following:

1) Move $1C00-$39FF up to $9600-$B3FF (NOTE that we use the memory move routine to move itself too out of the way so it doesn't get overwritten).

2) Jump to $B31B to continue.

3) Store a $60 at $BD00 (this will disable the disk access within the game).

4) Unpack $800-$8FFF from $39E8-$8FFF.

5) Jump to $7000 to start the game.

To do the above, we can use the memory move writer to write the memory move to run at $3900 and to move $1C00-$39FF to $9600, add some bytes to disable DOS and to set up for UNPACKER at $3918, and then bload UNPACKER at $3928 to unpack our packed memory. Here is the routines including the memory move:

```
3900-    A2 00        LDX    #$00
```

```
3902-   BD 00 1C    LDA   $1C00,X
3905-   9D 00 96    STA   $9600,X
3908-   E8          INX
3909-   D0 F7       BNE   $3902
390B-   EE 04 39    INC   $3904
390E-   EE 07 39    INC   $3907
3911-   AD 04 39    LDA   $3904
3914-   C9 3A       CMP   #$3A
3916-   D0 E8       BNE   $3900
3918-   4C 1B B3    JMP   $B31B
391B-   A9 60       LDA   #$60
391D-   8D 00 BD    STA   $BD00
3920-   A9 E8       LDA   #$E8
3922-   85 00       STA   $00
3924-   A9 39       LDA   #$39
3926-   85 01       STA   $01
3928- UNPACKER
.
.
.
3983-   4C 00 70    JMP   $7000
```

Notice right before unpacker we set the parameters for it by putting the low packed byte of the packing address at $00, and the hi byte at $01.

Now that we have all of the Roundabout memory in memory and ready to save to disk. type:

```
*1BFD:4C 00 39
*BSAVE ROUNDABOUT,A$1BFD,L$7A04
```

And your all done!

NOTE: If the hi-res page is changed at all from the original in the game, the amount of memory used after packing will be different.

The Annotated Mr. Xerox

In my previous note I distinguised between syntactic and semantic attacks on protected Apple disks. Syntactic attacks, epitomized by the "nibble copiers" such as Locksmith, attempt to duplicate the format of information on the disk without regard to its meaning. Semantic attacks, epitomized by the "boot trace" method, attack the code itself. The design of the Apple is such that boot trace attacks can be made tedious but not impossible.

Syntactic attacks proceed in the following fashion: most nibble copiers are highly paramaterized. An attacker examines the format of the target disk (using the "nibble editor" feature of the program) and then devises a set of parameters which permits the copier to duplicate that format. Most of the parameterization deals with the various techniques used to detect gaps and track starts in the completely "soft" Apple disk format. Once an attacker finds a set of parameters that works, they are quickly published (in some cases, by the marketer of Locksmith, who as a consequence attends few software vendor's conferences) and pirated copies of the given disk rapidly becomes available in the halls of Junior High Schools all across the country.

A new technique used to defeat syntactic attacks takes into account the entire piracy process. In particular, it recognizes the fact that only a miniscule minority of people who own (usually pirated) copies of Locksmith can actually devise a set of parameters; instead they rely on the published ones. What the software vendors are doing now is releasing the same program in a multiplicity of different formats. Discovering a set of parameters for one instance of the program does not help you copy another instance. Copies of copies still propagate, but the process is much slower.

The process of semantic attack was outlined in my previous note. I have been fortunate in being able to obtain the seminal essays in the field, written by the pseudonymous "Mr. Xerox." These essays were stolen (what else?) from a ripoff of a for-profit pirate bulletin board in the Boston area. I have converted them to 80-column upper/lower case (the original shouts a lot), deleted some expletives, and worked over the language a little bit. I have also interspersed notes in brackets to clarify certain points and make the discussion a little more intelligible to persons other than Apple hackers. I went to the trouble because this stuff is too good to be lost to posterity. Enjoy.

[Start of first article.]

Note: I chose Apple Galaxian here because it is a widely distributed program, and it encompasses the basic ideas in "boot trace" cracking. For all those interested pirates out there, yes, there is another way to crack programs. You don't need any ram-cards, prom burners, or foreign to regular DOS programs. Anybody who is not a clown, with some machine language programming ability can trace a boot. This method of cracking, tracing the boot, is, in a true sense, cracking the code. You see, for all disks, they must first boot up to start running. After the first stage boot (at location $C600), they jump to the second stage boot program (at $800), and then to a third, and some even a fourth, but there comes a point where the loading of the program from disk stops, and the running of the program begins. If you can trace this, and stop it after it has finished loading, and save all the memory locations that contain the program onto a normal 3.3 disk, you have cracked the program. This method is most useful for cracking the "single-shot" booting programs such as Apple Panic, Raster Blaster, and Gorgon. These disks don't contain any standard DOS, but rather their own.

This DOS has just one purpose, and that is to load the program into the computer, from the disk, and start its execution. Now, this is not as simple as it sounds, as the software protectors are not dumb, they try to make it tough for you to trace. However, it is not impossible, since the disk must boot up, and since it must have some booting process, that process is traceable.

[Because of the ease of boot tracing, and the use of the "Snapshot" type boards that dump memory to disk, all new arcade games make disk accesses in the midst of play. These accesses are made even though the entire game possibly fits into memory. The purpose of the accesses is to raise the work factor in cracking the disk.]

Let me try and show you an example of how to trace a boot of a program. Let me show you how to trace Apple Galaxian. The first stage boot starts at $C600. If you turn your Apple on, and type "CALL-151 (RETURN)" AND "C600G (RETURN)", the disk will proceed to start and boot the disk in the drive. This is because $C600 containing the program for the disk to boot first.

[That is to say, $C600-$C6FA contains a ROM program which is invoked on power up or a PR 6. It is the fact that this ROM program is executed by the main 6502 processor, and not the controller chip, that makes boot tracing possible. If the controller chip executed it with hard-wired addresses then it would not be possible to redirect the JMP $801 and start the boot tracing process.

For non-Apple types, CALL -151 is the entry into the "system monitor," a set of ROM utility routines. The ones referenced in these articles are invoked by console commands <addr>: <byte string>, which alters memory, <destination addr> < <start addr><end addr>M, which relocates, <addr>L, which disassembles a screen worth, and <addr>G which executes an uncondtional branch to <addr>.]

If you examine this program by typing "CALL-151 (RETURN)", and "C600LLLLLLL (RETURN)", you will soon come across a JMP $801, Near the end, specifically, at $C6F8. This is the link to the next stage of the boot. What we must do is allow the first stage to load in at $800, but instead of letting it run (continue to boot, and go to $800), we stop the computer, and examine what is at $800.

To do this let's move $C600 down to $ 9600. Type "CALL-151 (RETURN)" AND "9600<C600.C700M (RETURN)." This moves C600 down for you. Then type "96F8:4C 59 FF (RETURN)." Instead of having the boot go to $800, this will make it jump to $FF59 (the reset location). Then type "9600G".

[The effect of these steps is to move the ROM boot program down into RAM, and modify it so that the JMP to the next stage boot is replaced with an entry into the system monitor.]

Your disk should boot up for a second or so, and then you should hear a bell, and the monitor cursor will appear at the bottom of the screen. The next step is to examine the cod which starts at location $800. If you look at this by typing "800L (RETURN)" You will see the second stage boot of Apple Galaxian.

[Or, of course, whatever it is you are tracing, since the JMP $801 is normally in ROM and must be encountered by any bootable disk.]
By typing "800LLLLLLL (RETURN)", you can see what goes on next in the boot step. What happens next is that it takes the memory that is stored at $800, and moves it down to $200, and some other stuff, like loading the next stage of the boot. Then, if you look at location $841, you will see a JMP to $301. This is the next stage in the boot. So, we must move what is in memory up out of $800, because the next time we read something from the disk the locations at $800 will be changed. So type "9800<800.900M (RETURN)", and that will do the move. The next thing is to change what is at $9800, the stuff we just moved up, so that it will run at $9800, instead of its normal location of $800. To do this, type " 9803:BD 0 98 (RETURN)" and "9841: 4C 01 93 (RETURN)".

[$9841 contains the relocated jump out. This is changed to a JMP to $9301, which is loaded with a monitor entry in the next step.]
Then type "9301:4C 59 FF", because we changed it to run at $9800, and also changed it to stop after running instead of jumping to the next boot stage, at $300. We told it to jump to $9300, and at $9300, we put a JMP $FF59 (jump to RESET). And finally, change the JMP at $96F8 from $FF59 to $9801 by typing "96F8:4C 01 98".

[This last change relinks the relocated ROM boot routine to the relocated next stage boot routine.]

Now again type $9600G. This time, we are one stage farther. If you now move the stuff at $300 up to $9300, and change it to work at $9300 by typing "9300<300.400M (RETURN)" and "9313:AD CC 93 (RETURN), and "933C:AD CC 93 (RETURN)", this will be completed.

[Note what has happened here: the relocated "shadow" boot sequence in high memory has been linked together and run to load low memory but not execute it. The last two changes are Apple Galaxian-specific.]
But now, there is a problem. The jump out is at $9343, and it jumps not to the next stage immediately, but to a certain number of

subroutines. Then, through the same jump, it jumps to the next stage. How do we get around that, you ask?

The answer is to write a program that checks to see where it it jumping to, and if it is not jumping to where it normally jumps to, then stop, because we know that the next jump is not to a subroutine, but to the next stage of the boot. This may sound complicated, but just type this routine in at $9400: "9400:A5 3E C9 5D D0 03 6C 3E 00 4C 59 FF", AND "9343:4C 00 94 (RETURN)". That will take care of this stage.

[What is being described in the above paragraph is a test for a specific page zero pointer value. Protected disks now attempt to defeat boot tracing my establishing a variety of subtle dependencies between boot stages, including self-modifying code and the use of stack residues. The 6502 is such a simpleminded processor that these tricks raise the work factor by very little, if any.]

Now check to see that you have typed in everything correctly, and then type "9600G" to restart the boot. Now the disk spins for a little while longer, and when it stops we will have come to the last step of this boot process. This step loads the main program in from disk, and then jumps to the beginning of it.

By typing "93CC (RETURN)", the computer will display the page-1 of the next stage boot.

[This again is Apple Galaxian-dependent. This step examines the high-order byte (as we, not the 6502, read addresses) of a base address. The base address is evidently that used by a relocation routine.]

It will display "B6", and you add one to it, and get $B7, so type "B700L", and presto, we have the next stage of this boot. This stage does the program loading, along with turning on the graphics, and jumps to the beginning of the program. If you can see it, the beginning of the program is at $600, and there is a JMP to $600 at location $B759.

So, all we have to do is to have it do all the loading, and instead of having it jump to $600, stop it there.

[This again is a description of having the "shadow" loader load the code and then stop.]

But there is a problem connected with this (aren't there always!). The problem is that if we stop it here, location $600 is in text video memory, so we must not have it jump to $FF59 (STOP), but jump to a routine that relocates everything from $0000-$0800, and then stop. [This is a description of a standard situation in the Apple. Low memory is used for the stack, bit-mapped text displays, and console input. Many of these locations change when RESET or other console keys are hit. Such changeable memory is of course prime real estate for anybody trying to make code hard to analyze.]

I will provide you with this. Just type "B500:A2 00 B5 00 9D 00 20 BD 00 01 9D 00 21 BD 00 02 9D 00 22 BD 00 03 9D 00 23 BD 00 04 9D 00 24 BD 00 05 9D 00 25 BD 00 06 9D 00 26 BD 00 07 9D 0 0 27 E8 D0 CE 4C 59 FF (RETURN)."

[The above is just a bunch of LDA/STA loops.]

This will take care of moving everything from $0-$800 to $2000-$2800. But now change $B759 to jump to this small program by typing "B759:4C 00 B5." We also have to change some other locations. Location $93CC must be changed to $D6, so type "93CC:D6 (RETURN), and instead of jumping to $FF59 at $8409, and stopping at that stage of the boot, JMP to the beginning of this boot at $B700, by typing "9409:4C 00 B7 (RETURN)".

[This is a similar set of steps to those done before, relinking the "shadow" boot sequence back together.]

That takes care of most all preparations for the final crack. Now check to see that you have typed in everything correctly, and if you are ready, type "9600G".

[Thus entering the duplicated ROM code (often called "boot zero" and duplicating the steps that occur on a power on or PR 6.]

If everything worked correctly, it should boot up for about 10 seconds, and you should see the Hi-res picture loading in, and then your speaker should beep, and you should see on the screen a bunch of letters. If this didn't happen, check all the above steps, and repeat the process. If it has, then you are just about finished. If you want to check to see if it has worked, assemble this program, and type it in at $B560. If not, go on to the next step.

```
      OBJ $B560
BEGIN LDX  $0
AGAIN LDA $2000,X
      STA $00,X
      LDA $2100,X
      STA $100,X
      LDA $2200,X
      STA $200,X
      LDA $2300,X
      STA $300,X
      LDA $2400,X
      STA $400,X
      LDA $2500,X
      STA $500,X
      LDA $2600,X
      STA $600,X
      LDA $2700,X
      STA $700,X
      INX
      BNE AGAIN    ;LOOP
```

```
        JMP $0600     ;BEGINNING OF PGM NOW
```

[The above program restores low memory to enable testing of the loaded game.]

Boot up a normal DOS disk, and save everything from $2000-$2800, which represent locations $0-$8 moved up by $2000. You should then repeat the whole boot trace, and proceed to the next step.
[In other words, save low memory, then reload the rest of memory using the "shadow" boot sequence.]

Examine the memory of your Apple. You should save all the information from $800-$A000 on a normal DOS disk, then link the files that you have saved on the dos disk together, and make the file a BRUNable file that loads everything in, and moves the $00-$800 image back down in memory, and then jumps to location $600, the beginning of the program.
[Note how easily the loading of critical information in low memory, considered a sophisticated technique by unsophisticated programmers, is defeated.]

If any of you are unfamiliar with how to save everything, and need some help, here is how to do it: Follow the directions for traceing the boot, and type "2800<9600.A000M (RETURN)" and "3200<800.900M (RETURN)." Also, we need a program to move everything that we just relocated back into their original locations.

This time, I will assemble it for you. All you have to do is type "3400:A2 0 BD 00 20 95 00 BD 00 21 9D 00 01 BD 00 22 9D 00 02 BD 00 23 9D 0 03 BD 00 24 9D 0 4 BD 0 25 9D 0 5 BD 0 26 9D 0 6 BD 0 27 9D 0 7 EA (RETURN)" and
"3432:BD 0 32 9D 0 8 BD 0 33 9D 0 9 EA (RETURN)" and
"343F:BD 0 28 9D 0 96 BD 0 29 9D 0 97 BD 0 2 A 9D 0 98 BD 0 2B 9D 0 99 BD 00 2C 9D 0 9A BD 0 2D 9D 0 9B BD 0 2E 9D 0 9C BD 0 2F 9D 0 9D BD 0 30 9D 0 9E BD 0 31 9D 0 9F (RETURN)" and
"347B:E8 D0 84 EA AD 57 C0 AD 54 C0 AD 52 C0 AD 50 C0 EA 4C 00 06 (RETURN)".

[If you don't like hand-entering long hex programs you should stay out of the cracking game. Certain prehistoric types who entered 7090 patches through console binary switches are right at home.]

This will take care of the small program that we need to move everthing back. But we also need to put a JMP $3400 in the beginning, because when it BRUNs, it must jump to this small program first. Now you can boot up your 3.3 disk and type "CALL-151 (RETURN)", "9FD: 4C 00 34 (RETURN)","A964:FF (RETURN)", AND "BSAVE GALAXIAN,A$9FD,L$8C03 (RETURN)." And now you are finished.

[End of first article.]

Space Raiders, by Paul Lutus of USA, is a pretty crummy game in my opinion, but it is very easy to crack. Its boot contains only one stage, and the protection against cracking it is minimal. It should give you another basic example of how to "boot trace" crack programs. If you remember from the last cracking tips article, the first stage boot is at $C600. At $C6F8, the boot JMPs to $801, the next stage of the boot. So, what we must do is have it load the second stage boot in, stop, and then examine the code for the jump to the next stage, or the start of the program.

Let's start by moving the boot from $C600 down in memory to $9600. To do this type "9600<C 600.C700M (RETURN)", This will do the move. Now we must have it stop there instead of going onto $801, so type "96F8:4C 59 FF (RETURN)". Now we are ready to to initiate the first stage of the boot, and we do so by typing "9600G (RETURN)". The drive will go for a split second, and then the monitor cursor should appear in the lower left corner of the screen. If this has not happened, repeat these steps. Now we can examine the next stage of the boot. Type "801LLL" to see the next stage of the boot. If you examine it, and trace it in your brain (remember you have one, not like some Bozos), soon you will see a JMP $4000, and that is the the end of this boot.

After it loads everything in, it then jumps to the stuff it has just loaded in, which is at $4000. $4000 just happens to be the beginning of the program. So now that we have this stage in, we must move it up in memory, and change its JMP from $4000 to $FF59, to stop it there and allow us to save everything onto a normal 3.3 disk. You can do that by typing "9800<800.900M (RETURN)", and "9885:4C 59 FF (RETURN)" and "96F8:4C 01 98 (RETURN)." Then reboot the disk by typing "9600G." Now, when the monitor cursor appears at the bottom of the screen again, we know that the boot is finished. You can check to see if the program runs by now typing "4000G". But wait, what happened?

. The 's. This is their
protection from letting you stop the program and then typing $4000G. You see, at location $9885 where we had the jump to $FF59 (the RESET location, the boot proceeded to jump to that location in ROM. But at that program in ROM [i.e., at the Apple Monitor entry point] the value of certain zero page locations were changed. One of the locations that changes was location $21. If you look at the second stage boot again, and look at the two commands just before the jump to $FF59, you will see something like: LDA $26 STA $21 JMP $FF59. Can you see that if you repeat the whole boot that I just explained, and instead of testing it immediately by typing "4000G (RETURN)", type "21:26 (RETURN)", and then "4000G", it will run?

If you have not tested it, then you have my guarantee that it will do what I describe. You see, somewhere in the program that starts at $4000, it checks to see if there is a $26 in location $21 If there is not, then it will crap out, if there is then it will run.

[The above instance is a not atypical example of what simpleminded techniques are used in attempts to complicate use of the boot trace method. Once the code is made visible, then all these little games become pathetically transparent. There seems to be a parallel here with people who develop cryptologic systems in the absence of any strong cryptanalytic ability, and as a consequence make fatal assumptions about the attack method.]

Now we are just about finished. We just need a small program that will go before the program at $4000 and put a $26 into location $21. So type " 3FF0:A9 26 85 21 4C 00 40 (RETURN)". This small program looks like:

```
     LDA  $26
     STA $21
     JMP $4000
```

Then boot up a normal disk, and do a BSAVE like this - "BSAVE SPACE RAIDERS,A$3FF0,L$4100", and you will be finished.

[End of second article. Few games are protected in such a simpleminded fashion anymore.]

If you have read my last two articles, you should be familiar with how to "boot-trace" crack programs. If you have not read either then you will be completely lost in this one, so get a hold of one of them, and study them before proceeding with this one. Apple Galaxian and Space Raiders were fairly easy to crack, but now come the tougher ones. [This paragraph was left in to make the reader familiar with Mr. Xerox's locution, which is strangely reminiscent of that of a Zen master. You should meditate on this.]

The protectors can make things complicated by adding nibble counts to the software. If any of you are not familiar with what a nibble count is, it is a certain track that contains a specific amount of a certain byte. The program that is protected on the disk, sometime during its run, goes back to disk to reads all of these nibbles back. If the program doesn't find these bytes, or the right number of them, it will crap out.

[Another reason for otherwise unneeded disk accesses in the middle of arcade games. The idea here is that all Apple disk drives run at a slightly different speed. If you copy the track using a nibble copier, the speed variation will cause the copy to have a different number of the bytes being counted. The code can thereby detect it is dealing with a copy and not an orginal. If you have access to the Locksmith technical notes, you can observe what a tedious business it is to defeat nibble counts using that program. Then note how easy it is using a semantic attack described below.

Some protectors get it only half right, and the program craps out in a manner that permits entry into the Apple monitor. That is, the program defeats nibble copiers like Locksmith but opens itself up to semantic attacks. Such occurences are the occasion for great merriment in the cracking community.]

Bug Attack is one of these programs. After the title page comes on, and there is the explosion of dots, it waits for you to press a key, or push a button. after you press a key, it goes back to disk, and does a nibble count. If it can read all these nibbles, and everything checks out, it will continue with the game. But if it doesn't, well, you're in trouble. So, if this is to be cracked, and put onto a normal 3.3 disk that contains no track dedicated to containing those bytes that should be read, we will need to defeat the nibble count. Let's first crack the program. Then I will show you how to defeat this specific count.

To crack this, first turn your apple on, and press "RESET" to stop the drive from booting the disk. Get into the monitor by typing "CALL-151 (RETURN)" if you have an Apple][+. If you have an older Apple][, then by pressing reset you will automatically be placed into the machine language monitor.

[The Apple][+ ROM is called the "Autostart ROM" because it goes to the disk boot routine on power up. As mentioned above, it also takes different action from the original Apple ROM when RESET is pressed.] Then type "8600 <C600.C700M (RETURN)", to move the phase zero boot from ROM into RAM, and "86F8:4C 01 88 (RETURN)", to make the boot continue at location $8801 instead of $801, and "8801:4C 59 FF

(RETURN)", to force the boot to stop instead of continuing to the next phase. Then start it up by typing "8600G (RETURN)".

Now move the second stage that is at $800 up to $8800 by typing "8800<800.900M (RETURN)", and modify it so it will run at $8800 by typing "8803:BD 00 88 (RETURN) and "8841:4C 01 83 (RETURN)." Then type "8301:4C 59 FF", and finally reboot by typing "8600G (RETURN)" again.

Now we are at the third stage that is at $300, so move that stuff up to $8300 by typing "8300<300.400M (RETURN)" and modify this stuff to run at $8300 by typing "8313:AD CC 83 (RETURN)" and "833C:AD CC 83 (RETURN) and "8343:4C 0 84 (RETURN)."

Now we will run into the same trouble that we had in Galaxian in that the jump out of this stage is not immediate, but only after many jumps to a certain subroutine. So we need that program at $8400 again that checks to see if it is going to the subroutine, or to the beginning of the program. If it is going to the subroutine, then let it continue, if not then stop. So type "8400:A5 3E C9 D5 D0 03 6C 3E 00 4C 59 FF (RETURN)", and reboot again by typing "8600G (RETURN)."

[Interestingly enough, the same pointer values are being checked here as in the Galaxian case. These two games may be from the same outfit. Accomplished crackers learn the idiosyncracies of the various companies. Sirius Software, whose president uses every possible forum to denounce pirates, uses some the most baroque techniques imaginable. As with cryptography, however, mere complexity does not give strength.]

Now to find out where the next stage loaded in, type "83CC (RETURN)." You will see an $A1, so add one to that, and you get $A2, so type "A200L (RETURN)". We are now at the final stage of the boot.

In this stage, the boot turns on the graphics, loads the program, and jumps to the beginning of it. If you type "L" a few times you will come across a point where this stage ends, and the jump to the beginning of the program is located. The jump is at location $A2F8, and it is an indirect one to $1FF. If you don't know, an indirect jump to $1FF doesn't jump to the locations that $1FF and $200 point to, but to the locations that $1FF and $100 point to.

[The protector here is attempting to confuse novice crackers by exploiting an anomaly of 6502 addressing: pointers do not cross page boundaries but instead wrap around to the start of the page.]

So, to find out where this jump goes to, type "A2F8:4C 59 FF (RETURN)" and "8409:4C 00 A2(RETURN)" and "83CC:D2 (RETURN)", and finally reboot by typing "8600G (RETURN)".

[This should be old hat by now.]

Now we can examine location $100 by typing "100 (RETURN)" and location $1FF by typing "1FF (RETURN)". From this information you now know that the jump is to location $4D 36.

You have now cracked the program, but one more major obstacle remains in our way. The program contains a nibble count. If you boot the original, and press button (0), You will see that it goes back to disk for a second and does the count. So the way to get rid of the nibble count is to find where it is in memory, and just avoid it when the program is run.

I have examined the program and found that after the title page is displayed, and the dot graphics explosion takes place, there is a jump at $4E24 that goes to the nibble count routine at $4A33. After the nibble count is done, there is a jump out of it at $4A88. This jump is to the begining of the game, location $494A. Now, we can modify the whole nibble routine at $4A33 just to skip turning on the drive, and jump directly to the beginning of the program. But like always, they (the protectors) have stealthily hid a routine in the middle of the game that checks to see if the nibble count routine has been changed in any way. If it has than the program will crapl out, if not then it will continue with the game. Pretty sneaky of the protectors, huh?

[It is difficult to see what the protectors had in mind when they came up with this baroque touch. If they are going to the trouble of checking the integrity of the code, then they must be assuming that someone is mounting a semantic attack. If so, the checking routine is just as visible as the routine being checked.

Incidentally, the easiest way to locate nibble count routines is to search for references to locations $C080-$C08F, X-modified (the usual format) or $C0E0-C0EF (absolute references to slot 6). These locations hold the "soft switches" which must be referenced to perform physical disk operations.]

So to get around this problem, we must simply take the JMP at $4E24 goes to the nibble count part at $4A33, and change it to jump to the beginning of the program at $49A4. So make the change by typing "4E24:4C 49 A4 (RETURN)".

[Note how trivially the "check the checking routine" strategy is defeated. Some poor souls go so far as to checksum memory. The attack for that is left as an exercise for the reader.]

After this change has been made, the program is in a form which can be saved to a normal 3.3 disk. Don't forget to save pages $0-$8 with the rest of the file , and load them back into memory when you BLOAD the file back in normal 3.3 DOS. If you have just read all this, and you don't believe that it will all work, try this, here is a program that will do the nibble count changes and will show you that the nibble count was really defeated.

```
      ORG $A800
START STA $AF00  ;STA TEMP
      LDA  $4C   ;JMP BYTE
      STA $4E24  ;JMP LOCATION
      LDA  $A4   ;LOW BYTE
      STA $4E25  ;JMP LOCATION+1
      LDA  $49   ;HIGH BYTE
      STA $4E26  ;JMP LOCATION+2
      LDA $AF00  ;GET OLD A VAL BACK
      JMP $4D36  ;BEGINNING OF THE PROG
                 ;THIS WILL DO THE JMP
                 ;TO PROVE THE DEFEAT
                 ;OF THE NIBBLE COUNT.
```

The assembled version is "A800:8D 00 AF A9 4C 8D 24 4E A9 A4 8D 25 4E A9 49 8D 2 6 4E AD 00 AF 4C 36 4D (RETURN)", and we need to jump to this location instead of ($1FF), so type "A2F8:4C 00 A8 (RETURN)". This must be done in the boot trace instead of entering "A2F8:4C 59 FF". When you run the boot, the game will proceed normally, but the disk will never be accessed, and thus we have defeated the nibble count!

[End of third and last article.]

Regarding Tiger Teams
Off the record comment on Apple piracy:

1. General

It is not theoretically possible to protect a runnable Apple disk from a determined attack; the only thing a defender can do is raise the work factor. This circumstance arises from the nature of the Apple hardware. The autostart facility requires that the first bootstrap algorithm be in the disk controller ROM and is therefore unmodifiable by would-be protectors. As a consequence track 0, sector 0 of an Apple disk **must** be in a known format.

As a consequence, there is a standard pirating methodology which is to simply trace the bootstrap sequence by hand, noting and dismantling the funny stuff as you go. This always works; it just may take more or less time. The process is simplified by the extreme simplicity of the 6502 architecture and the corresponding limit on low-level trickery. A knowledge of the basic Apple DOS hardware and software, and 6502 machine language is all that is required. The DOS knowledge can come from a variety of sources; the most efficient is the book "Beneath Apple DOS," supplemented by the disassembled DOS listings available from the same people.

2. Objectives

There are basically two possible objectives for an attack on an Apple disk. One is to make it "nibble-copyable" by something like Locksmith. The other, and more elaborate, is to put it into standard DOS format so that COPYA works.

3. Protection Technology

Protection method rely on two basic factors: the extreme "softness" of the Apple disk technology and the rigid expactations of DOS with regard to format. First generation protection methods aimed at protecting the software from DOS technology; second generation aims at protecting it against the nibble copiers.

Basically, the Apple hardware provides only two things: the ability to recognize the start of a byte on disk and the ability to find track 0, sector 0. The first makes a wide variety of protection schemes possible and the second makes a perfect one impossible.

Protecting against DOS utilities involves writing information out in nonstandard format and using a non-DOS to read it. Almost everybody (Infocom is the only example to the contrary) constructs a non-DOS by patching DOS. This makes the changes trivially easy to find and rip out.

Protection against nibble copiers involves playing games with the algorithm that the hardware uses to recognize the start of a byte and the distinguished bytes which indicate the start of a track. Schemes also use what is called "nibble counting," which exploits the speed variability of the drives, "synchronization," which is the alignment of track starts in a time-dependent fashion, and "quarter tracks" or "half tracks" which exploit the fact that the head is positioned by software and not hardware (that's right, in software you have to turn the stepper motors on and off in the proper sequence. Ever wonder how Wozniak got the controller board cost down?).

4. Attack Technology

The general detailed algorithm for pirating an Apple disk is:

4.1 Trace the bootstrap sequence and see what they are doing.

4.2 Find out what funny stuff you need to bypass. Usually the really exotic techniques are applied to just one or two tracks in the bootstrap sequence because they impact performance so much.

4.3 Kludge together a utility which uses their code to read a track and a copy of standard DOS to write it out. Unpatch their code so that they look for standard DOS as well.

4.4 Treat yourself to a beer, you sly devil.

Stupid pirates fool around with just the data formats on the disk. Smart pirates go after the code. This is the same basic methodology as used on mainframes: don't attack the data, attack the operating system.h

5. Tools

You need a nible editor (a pirated copy of Locksmith works fine), a good set of disk utilities (a pirated copy of Bag of Tricks is good here), and a good disassembler (there is a decent one available from Call-APPLE). In addition, you will probable write a "DOS-ifier" which converts the class of funny formats obtainable by doing easy patches to DOS back to standard. And the usual program development tools.

6. Work Factor

Most senior pirates are semiretired and just crack things far enough to see and sneer at the scheme being used. This takes at most a week of evenings. Another week of evenings and you can have the thing in COPYA format, unless the source format is so wildly nonstandard that it isn't worth the trouble and you just take it to the point where a nibble copier can handle it.

With regard to BB's, call 933-1771 to get a list of Twin Cities boards. Most pirates communicate via Pirate's Cove: (516) 698-4008. Be patient; it is heavily overloaded.

Earl