The Kracker Jax Revealed

TRILOGY

Book I · Book II · Book III

TABLE OF CONTENTS

INTRODUCTION

Thank you for your purchase of the Kracker Jax Revealed Trilogy. The Trilogy is a compilation of our three separate Kracker Jax Revealed books, Volumes 1 - 3. These books were individually released over a period of two years, and represent countless hours of investigation.

We've condensed Volumes One and Two into one book, while Volume Three continues to be a separate book. All are included with your purchase of the Trilogy.

The Kracker Jax Revealed series represents the best compilation of this type of material ever assembled. The 3 Volumes were put together from a mountain of raw data that was the end result of years of hands on experience by people who were masters of the craft of unlocking the mysteries of copy protection.

Inside these pages, you'll find the knowledge that will give you the power to take complete control over your software. The step by step instructions and detailed explanations have all been designed to pass along years of practical experience to you in a very short amount of time. Once you've finished the Revealed Trilogy, you may want to read other similar material. You'll find our specific recommendations listed under "Books For Further Reading" in the Table Of Contents.

The path to knowledge is a rewarding one. We hope you enjoy the journey.

KRACKER JAX REVEALED 1 & 2

INTRO : PROTECTION SCHEME TYPE A

Owners of the 1541 disk drive may not realize it, but every time they boot their favorite program and it bangs the disk drive head, that program is using this form of protection. It is common knowledge along experienced users that this form of copy protection is hazardous to the health of the 1541 drive. Let's face it: would YOU write a program that purposely banged YOUR disk drive's read/write head against it's end stop?

This protection is still being used by many software publishers, knowing full well that the drive knock is probably the major source of alignment problems with the 1541/1571 disk drives. We at Kracker Jax can't see any purpose in the continuation of this form of protection.

Sure, you can back up your software with almost ANY nybble utility on the market. The problem is that the backup is ALSO protected and will bang the drive as well. It is this protection type that we especially urge you to learn to break, just so you can preserve the alignment of your disk drive.

The operation of this scheme is simple. The programmer writes a routine in the program (generally in the boot) to seek out a non-standard sector on the disk. If that non-standard sector is found, the drive will usually bang, and the program will continue operations. If not, the program will cease to operate or "crash". These non-standard sectors are generally write errors, and are documented in your 1541/1571 drive manual. The lost commonly used are the following:

20: Block header not found I drive banger.

21: sync character not found I sector not formatted properly I drive banger.

22: Data block not present I drive banger.

23: Checksum error in data I very common I drive banger.

26: Attempt to write with write protect on I some programs check for the write protect I no drive bang.

27: Checksum error in header I drive banger.

29: Disk ID mismatch I whole track formatted with wrong ID characters I no drive bang.

Many of the programs using this scheme are checking the protection with simple drive commands and the kernal routines in the computer ROM. Keep in mind that this check can be done with Basic programming as well as machine language. Once understood, most are fairly easy to unprotect.

Most of the time the programmer will check for the bad sector with a block read. It will look something like this: Ul:aa bb cc dd, or B-R:aa bb cc dd. The aa denotes channel, bb denotes drive number, cc denote track, and dd denotes sector. A character or two is then returned from the drive, and a comparison is made. If the comparison is satisfactory, the program continues operation. If not, the program flow is ended or set in an endless loop. Our task will be to either give the program the proper characters, or to short circuit the program flow around the protection check.

Before starting to work on any of the following programs, please do a disk log, an error scan, noting all write errors, and make a C-64 Fast Copier backup which will remove all errors. Place a write protect on the original disk.

TAPPER : BALLY NIDWAY

Procedure:

Loading the original produces a drive rattle twice. An error scan shows write errors on the original. A backup made with Three Minute Backup produces a non-working copy. Before starting to work on this program, make two backup copies.

working with your backup:

1) In order to look at the boot with our monitor, we must change its location in memory. The reason for this is because this boot cannot be stopped once it has been started. This is a simple procedure. From your utility disk, load DISK DR <> LOAD"DISK DR",8,l <>. When the cursor reappears type RUN and hit RETURN. Remove the utility disk and insert one of your Tapper backups in the drive. Hit RETURN again and you will be shown Track 18, Sector 1. Cursor over to position 3 and hit the J key. This will take you to the first sector of the Boot file. The first four bytes in this sector are the pointer bytes. Bytes 0 and 1 are the pointers denoting this as the only sector and the number of bytes used in this sector. Bytes 2 and 3 are the program address bytes in reverse order. Place the cursor over the byte in position 2 and hit the @ key. Now, type a 1 and hit RETURN. The cursor should now be on position 3. Again hit the @ key and type an 8 and hit RETURN. To make the changes on the backup, hit the

C key and hit RETURN. The sector is now changed on the disk. We have just changed the boot address to $0801, which places it in Basic memory. The boot will not run properly , but we may load and examine it.

2) Turn your computer off and insert the reset button assembly into the cartridge port. Turn on the computer and load the $8000 monitor from the utility disk <> LOAD "32768",8,l <>. Type SYS 32768 and hit RETURN. With the monitor active, place the altered backup in your drive and load the boot file <> L "BOOT",08 <>. disassemble code at $0801 (D 0801) and scroll down through the code. The code from $0838 to $0853 is a loader routine that loads in the file LOADER and then jumps to $COOO. This gives us the information we need to trace the program flow. Take this backup out of the drive and put the other backup in its place. (Remember, we altered the boot file on this backup.)

3) Load the LOADER file <> L "LOADER" ,08 <>. When the load is complete, interpret memory at $C000 (I C000). Scroll down through the code watching the left hand screen. At address $ClA3 you'll find the block read command Ul 2,0,32,8. This is the command to read Track 32 Sector 8. Our error scan has shown a 21 error in this location. Now let's disassemble and locate the protection code.

4) This protection scheme is written as a series of JSR (GOSUB in Basic). Remember each JSR ends with a RTS (RETURN). Code will be explained in segments. Try to follow the program flow.

A] Starting at $COOO in the DISASSEMBLE mode, scroll down to $C017 : JSR C116.

B] Disassemble $C116 : JSR Cl4B.

C] Disassemble $Cl4B : JSR C172.

D] Disassemble $C172 : Opens a channel to the drive then RTS.

E] Disassemble $C14E : JSR C188.

F] Disassemble $C188 : Sends Block Read command to the drive then RTS.

G] Disassemble $C151 : JSR ClAF.

H] Disassemble $ClAF : Inputs two characters from the error channel and stores them at $ClCA and $ClCB then RTS.

I] Disassemble $C154 : The accumulator is loaded with the error character placed in $ClCA and compared with a $32. The accumulator is then loaded with the error character placed in $ClCB and compared to a $31. This is the hexadecimal equivalent of a 21 error ($322, $31=l). Notice that if both comparisons ARE equal, the accumulator is loaded with a 0, if not, it's loaded with a 1, then a RTS.

J) Disassemble $C119 : The accumulator is compared with 0, and if equal a branch to $C127 occurs. To see what happens, type G C127 and hit RUN/STOP-RESTORE. Code was transferred to the $8000 area of memory and was activated by the RUN/STOP - RESTORE. You'll have to turn of f the computer and reload the monitor and the LOADER file again.

K) Disassemble $CllB : Increment $ClAB (increments the track of the Block Read to 33).

L) Disassemble $CllE : Increment $ClAB (increment the sector of the Block Read to 09).

M) Disassemble $C120   JSR Cl4B   Goes back through the error check routine once again but now the 21 error at Track 33, Sector 9 is checked (the second drive rattle). This time if the code is not branched to the message screen as before, it will return back to $COlA to resume normal loading.

N) This program can be broken in many different ways. Three will be given.

1) Place three NOPS at $C017 (EA EA EA). This will erase the code that sends the program to the protection check in the first place (our choice). The program will never do an error check.

2) Place a BNE at $C119 and $C124 (Do). This will instruct the program to operate in an opposite fashion in regards to the protection, in other words, crash if an error is found.

3) Place a $30 at $C162 and $C169. This will instruct the program to expect NO error at the Block Read locations. Again, if an error is found, the program will crash.

O) Choose one of the above methods and make your changes using the MEMORY command. After the change is made the LOADER file may be scratched and saved. Checking the disk log shows us the start address of $COOO and the end address of $C2BC. Remember to add one byte to the end address
<> S "@0:LOADER",08,COOO,C2BD <>.

Your backup is now broken and will never rattle the drive again.

Another benefit of this particular break is the fact that now you may file copy this program.

BUCKAROO BANZAI : ADVENTURE INTHRNATIONAL

Procedure:

Loading the original produces a drive rattle early in the load. An error scan shows write errors on the original. A backup made with the C-64 Fast copier produces a non-working copy.

working with your backup:

1) The disk log shows us that the boot file SAGA resides in Basic memory, so let's begin by loading the boot and examining it <> LOAD "SAGA",8: <>. List it out and notice it loads the file SAGA.OBJ and does a SYS to 4863 ($1300).

2) Turn your computer of f and insert the reset button assembly. Turn the computer on again and, from your utility disk, load the $C000 monitor <> LOAD "49152",8,l <>. When the load is complete, sys the monitor in with SYS 49152. Now load the SAGA.OBJ file from your backup, and follow the program flow <> L "SAGA.OBJ",08 <>. Start your disassembly at $1300 (D 1300). We will break the code down into sections for you. Try to follow along and inspect the code as we go through it.

A) $1300-$1323 Loads the SAGA.C64 file.

B) $1324 : Does a JSR to $137A which IS the protection check routine.

C) $137A-$l3BE Opens the error channel to the drive and sends the Block Read command to check Track 34, Sector 4. Interpret memory at $13BF to see the Ul (I 13BF). Then a jump to $13CE is taken.

D) $13CE-$l3FE : Two bytes are received from the error channel and stored at $1556 and $1557. Then a check of these two addresses for the proper error bytes is done. The bytes are compared to $32 (2 in decimal) and a $31 (1 in decimal). These bytes correspond to a 21 error in decimal. If the comparison is incorrect, the program branches to $l3FF. Do a GO 13FF (G 13FF) to see what happens. (You'll have to reload your monitor and SAGA.OBJ file again). If the comparison is correct, the program continues along until it encounters the RTS at $13FE. This will branch the code back to $1327, and the program load will continue.

3) This protection scheme is fairly simple, and extremely easy to defeat. Four different methods will be given to break this title. Choose one and make your changes with the MEMORY command.

A) Place three NOPs at $1324. This will erase the JSR to the protection routine. The program will never even look for protection now (our choice).

B) Place an $FO at $13E9 and $13F0. This will tell the program to fail if an error IS found.

C) Replace the code at $13E3 with A9 32 EA (LDA 32 EA) and the code at $13EA with A9 31 EA (LDA 31 EA). This loads the accumulator with the correct bytes the protection check is looking for.

D) Change $13E6 from a $32 to a $30 and $l3ED from a $31 to a $30. This tells the program to look for NO error ($30=0 in decimal). The program will crash if an error is found.

4) After your changes are made, all that is left is to save the code back to your backup. The disk log tells us the file resides from $1300 to $1575. Be sure to add one byte to the end address <> S "@0:SAGA.OBJ",08,l300,1576 <>.

Your backup is now free from the restrictions of copy protection. It will no longer bang your drive head and can even be file copied. This scheme can be found in approximately this form in many different programs. Don't be surprised if you see it again.

SARGON III CHESS : HAYDEN SOFTWARE

Procedure:

Loading the original produces a drive rattle twice at the end of the load. An error scan shows write errors on the original. A backup made with the C-64 Fast Copier produces a non-working copy.

working with your backup:

1) Turn the computer off and insert the reset button assembly into the cartridge port. Turn the computer back on and from your utility disk, load the $8000 monitor <> LOAD "32768",8,l <>. Sys the monitor in with SYS 32768. Place your backup in the drive and load the boot file <> L "SARGON III",08 <>. Start your disassembly of code at $02A7 (D 02A7). The code from $02A7 to $02F2 loads the COPYRIGHT 1984 file in and jumps to $COOO.

2) Load the file COPYRIGHT 1984 <> L "COPY*",08 <>. We will explain the code a section at a time, so try to follow as we go through it. Using the DISASSEMBLE command, disassemble memory beginning at $COOO (D C000).

A) Disassemble $COOO : $C000-$C091 sets up a loader routine that loads HAYDEN SOFTWARE and JUMPS to $C311.

B) Disassemble $C311 : $C311-C336 opens an error channel to the drive and sets the Y register to 0.

C) Disassemble $C337 : JSR $C376.

D) Disassemble $C376 : $C376-$C389 sends Block Read command to Drive to check Track 2, Sector 15. The address $C2F7,Y is accessed. Since Y has been set to 0, the true address IS $C2F7. Interpret memory at $C2F7 to see the B-R (I C2F7). This subroutine returns when an RTS is encountered.

E) Disassemble $C33A : JSR $C38A.

F) Disassemble $C3BA : $C38A-$C3AO inputs two bytes from the error channel and compares it to a $30 (0 or no error in decimal). If NO error is found, a branch to $C373 is taken. This in turn jumps to a reset vector and the program Crashes. If errors are found, the program flows until the RTS is encountered.

G) Disassemble $C33A   Loads the Y register with 0D (13 in decimal).

H) Disassemble $C33F : JSR $C376 : Same as step D, except this time the address $C2F7,OD ($C2F7+0D) is sent to the drive. This address is the same as $C304 and is the B-R command for Track 3, Sector 16 (I C304).

I) Disassemble $C342 : JSR $C3BA : Same as step f. Checks for error and RTS if found.

J) Disassemble $C345 : Close all channels and files; continue setup and jump to start of program.

3) This protection scheme is fairly simple and can be defeated in many ways. Four will be given. Choose one, and make your changes with the MEMORY command. When the change has been made, all that is left is to save the file back to the disk. The disk log tells us the file resides in memory from $COOO to $C3A2. Remember to add one byte to the end address when you save it
<> S "@0:COPYRIGHT 1984",08,C000,C3A3 <>.

A) Change the address $CO8F from 4C 11 C3 (JHP C311) to 4C 45 C3 (JMP C345). This will jump the program flow completely around the protection check (our choice).

B) Change $C33A and $C342 from 20 8A C3 (JSR C38A) to EA EA EA. This will erase the JSR to the error check.

C) Change $C397 from FO 0A (BEQ reset address) to EA BA. This will erase the branch to the crash and the program flow will be forced to continue on.

D) Change $C395 from C9 30 (CMP 30) to C9 32. This will force the program to crash if an error IS found.

After your changes are made1 you will have a completely broken copy that can be fast copied and even file copied.

THE SLUGGER : NASTERTRONICS

Procedure:

Loading the original produces a drive rattle. An error scan shows write errors on the original. A backup made with the C-64 Fast Copier produces a non-working copy.

working with your backup:

1) Checking the disk log shows us the boot file is in Basic memory so let's start by loading it <> LOAD "THE SLUGGER",8: <>. List it and examine the loader. It loads various files and then does a SYS 514 ($0202). The disk log again tells us the address $0202 is the start of the GOFILE file.

2) Turn the computer off and install the reset assembly into the cartridge port. Turn the computer back on, and from your utility disk, load the $2000 monitor <> LOAD "8192",8,l <>. Sys it in with SYS 8192. Now from your backup, load the GOFILE file <> L "GOFILE",08 <>. Start disassembly at $0202 (D 0202). Scroll down through the code and notice that this file loads the CODE file and Jumps to $0340.

3) From the backup, load the CODE file <> L "CODE",08 <>. Start disassembly at $0340 (D 0340). The disassembly is given in the sections below. Try to follow along as we go through it.

A) $0340-$036A : Opens the error channel to the drive.

B) $036B-$037D : Sends Ul (Block Read) command to the drive to read Track 6, Sector 7. Use the INTERPRET command to see the Ul (I 03E3).

C) $037E-$0385   Set up to read two bytes from the error channel.

D) $0389-$0396 : Inputs a byte from the error channel and compares it to a $32 (2 in decimal). Another byte is retrieved and compared to a $33 (3 in decimal). Each compare results in a branch to a crash address if not satisfied. Otherwise the program flow continues on to a Jump to $03A1. These compares are the 2 and the 3 of a number 23 error. The error scan confirms a 23 error at Track 6, Sector 7.

E) $03Al-$03A8 : Close error channel and normal program flow continues.

4) The break in this program is fairly simple. Four different methods will be given. Choose one and make your changes with the MEMORY command.

A) Change $0340 to 4C AB 03 (JUMP $03AB). This will cause the program to jump completely around the protection check (our choice).

B) Change $038D and $0394 to $30. This will instruct the protection check to look for NO error ($30=0 in decimal).

C) Change $038E and $0395 to $FO. This will cause the protection to branch to the crash if an error IS found.

D) change $0389 to A9 32 EA (LDA 32) and $0390 to A9 33 EA (LDA 32). This will load the accumulator with the bytes it wants in the compares. The bytes will not be input from the error channel.

5) When your changes are made, all that's left is to save them to the backup. The disk log supplies the start and end addresses. Be sure to add one byte to the end address
<> S "@0:CODE",08,0340,0401 <>.

Your backup is completely broken and may be file copied to another disk.

ROGUE TROOPER : UXB

Kracker Jax Revealed Book one dealt with this scheme in four different programs. We have included this one title because this exact protection is a litte tricky and has been found on quite a few programs.

Procedure:

Loading the original produces a drive rattle. An error scan shows massive write errors on the original. A backup made with the C-64 Fast Copier produces a non working copy. Before starting to work on this program, do a disk log and an error scan to determine error type and location.

working vith your backup:

1)   Let's start by plugging Hesmon in the cartridge port and powering on. Insert your backup in the drive and load the boot file < LOAD"UXB",8,l > . From the disk log we can determine that this file begins at memory location $032C. Start disassembly at $032C < D 032C > . Cursor down through the code. This code opens channels to the drive and loads a one character file name at $035B. If you Interpret memory at $035B < I 035B > you'll find the file name X. After the load, a jump to $08B0 is taken.

2) Load the X file into memory < LOAD"X" ,8,l > . Begin disassembly at $OBBO. The following is an explanation of the program flow.

```
D $08B0 : JSR 081E
D $081E : $081E-$0841 opens an error channel to the drive
          < I E260 > and does a JSR back because the JUMP to
          $FFCO is a kernal routine and always ends with a
          JSR.
D $08B3 : JSR 0844
D $0844 : Sends a Ul (Block Read) command to the drive from
          an encrypted form. The code from $084E-$085D decrypts
          and sends the Ul.
D $0868 : JSR FFA5 : Imputs a byte from the serial port.
D $0872 : CMP 081A ($32 or the 2 in a 23 error).
D $0874   BNE crash.
D $0877 : JSR FFA5 : Imputs a byte from the serial port.
D $087A : CMP 081D ($33 or the 3 in a 23 error).
D $087C : BEQ to a JSR which closes channels and RTS back to
          $08B6. Otherwise the program flow falls through to a
          crash.
```

3) There are many ways to break this title. Three will be given. Make all your changes using the Memory command and then resave the file to the backup as < S "@0:X" OB 0801 0977 >

A) Place 3 NOPs at $08B3 over the JSR to $0844. This will cause the program to not even check protection.

B) Place a 30 at $OBlA and at $081D. This will allow the drive to send back an OK condition and pass protection because we will now be comparing to NO error.

C) Place a 60 (RTS) at $0844 which will cause the routine that checks protection to be short circuited.

When your changes have been made, this title may be file copied.

INTRO : PROTECTION SCHEME TYPE B

This protection scheme has allowed software publishers a means of protecting their programs from the finest nybblers on today's market. It employs a loader that resides in RAM at $COOO. This loader does the protection check and then proceeds to gather a Basic boot from the program disk. This boot is placed in RAM at the beginning of Basic ($0801-). Our task in each of these schemes will be to let the original disk pass protection and then place the boot in memory. At this point we can retrieve the boot and from then on use it to load our back-up, leaving the protection check completely behind.

Before starting, you must understand the way a Basic program is placed in memory and how the pointers affect it. The reason for this is that most of the time upon reset, the beginning pointers will be destroyed and we will have to repair them ourselves.

The pointers used by Basic are very specific, and if not correct, the Basic program will fail to operate properly. To show you how a Basic program looks in memory, let's inspect the example on your work disk.

First, load the $C000 monitor from your Utility disk <> Load "49152",8,l <> and sys it in by typing SYS49152 and hitting RETURN. You should be in the monitor now so load again from your work disk the file called BASIC EXAMPLE <> L "BASIC EXAMPLE",08 <>. After the load, examine memory from $0801-$0890 (M 0801) Scroll up and down through the code. You should be looking at the same code as shown below. Please note that the example below has all pointer bytes underlined for ease of viewing.

```
     :0801 0E 08 05 00 99 22 93 11
     :0809 11 11 O5 22 00 35 08 0A
     :0811 00 99 22 20 20 20 54 4B
     :0819 49 53 20 49 53 20 41 4E
     :0821 20 45 5B 41 4D 50 4C 45
     :0829 20 4F 46 20 46 4F 57 2D
     :0831 41 20 22 00 5A 08 14 00
     :0839 99 22 20 20 20 42 41 53
     :0841 49 43 20 50 52 4F 47 52
     :0849 41 4D 20 49 53 20 46 4F
     :0851 52 4D 41 54 45 44 20 22
     :0859 00 84 08 1E OO 99 22 2D
     :0861 20 20 49 4E 20 54 48 45
     :0869 20 4D 45 4D 4F 52 59 2D
     :0871 4F 46 20 54 48 45 20 43
     :0879 4F 4D 4D 4F 44 4F 52 45
     :0881 2E 22 00 00 00 FD BD FF
     :0889 DO FF FF E6 FF FE 00 00
```

The format of Basic is as follows. Starting at $0801, the bytes OE 08 denote the placement of the next line number in memory in reverse order ($080E). The next two bytes, OS 00 denote the current line number in reverse order ($0005=5).

Follow the bytes from here until you get to the next 00. This byte (residing at address $080D) denotes the end of the first line in this program. The next four bytes are again the pointers for the second line in our Basic program. The address $080E and $080F contain the bytes 35 08. These are the address of the next line number in our program, again in reverse order ($0835). The next two bytes starting at $0810 are OA 00 which is the current line number of our program, again in reverse order (OOOA=OA in hex or line 10 in decimal). This format is followed all through any normal Basic program and ends only when three hex zeros are encountered (00 00 00). This tells Basic that the programs end has been found. You'll find these bytes in our example starting at $0883.

This means that this program could be saved with your monitor using the addresses from $0801-$0885. The $0801 being the beginning of Basic and the $0885 the last of the three zero bytes. The actual save command would be <> S "FILENAHE",08,0801,0886 <>. We used the end address $0886 because all monitor saves need one extra byte added to the actual ending address ($0885+l=$0886).

By understanding the structure of Basic, we can now repair any damage done to our pointers when we reset out of our program loads. Now let's move on to our example programs.

TITLE BOUT : AVALON HILL

Loading the original produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A backup made with a nybbler also produces a non-working copy. Before starting to work on this program, please make a non-working backup of the original.

Working with your backup:

1) Start by validating the BAM <> OPENl5,8,15,"V":CLOSEl5 <> to make room for a new file we will be adding later. Scratch The first file from your backup <> OPENl5,8,15,"S0:AH":CLOSE15 <>.

Working with your original:

2) Place a write protect tab on the original to ensure its safety during the breaking process.

3) Turn of f your computer and insert your reset assembly into the cartridge port. Turn the computer on again and load the boot file and start the load process <> LOAD"AH",8,1 <> . Allow the program to load until the screen turns black and the words LOADING DATA appear in the middle of the screen. At this point, reset the computer.

4) Remove the original disk from your drive and insert the utility disk. Load the $COOO monitor <> LOAD"49152",8,l <>. When the load is complete, sys the monitor in with sys 49152. The monitor should be active now. Remove the utility disk from the drive and replace it with the backup work disk.

5) Interpret memory starting at $0801 (I 0801). Scroll through memory and notice the Basic program. Our task is to repair the pointers and save the program to your backup (see Scheme B Intro). Using the memory command (N 0801) inspect code at 0801. Notice that the first two bytes are 00 00. These two bytes represent the start of the flan line in this Basic program. Obviously, these bytes have been destroyed by the reset because the next line couldn't be zero. To find the correct bytes to replace the two zeros, follow this procedure. We know the first four bytes are pointer bytes ($080l-$0804). we also know that the next time a zero byte appears in memory ($0811), it signals a new line. The next address is the address that the pointer will point to ($0812). Therefore, the first two bytes in this program should be 12 08 because all addresses are read in

reverse order. Now we can scroll to the two zeros at $0801 and type over them 12 08 and hit RETURN. The first four bytes starting at $0801 should now be 12 08 Ol 00 (the Ol 00 bytes represent the current line number in reverse Ol 00=00 01). Our Basic program is now repaired and all that's left is to locate the end of the program and save it to your backup disk. To find the program end, use the HUNT command in your monitor. We'll hunt for the three zero bytes that signal the end of Basic. <>H 0801 8000 00 00 00 <>. As the first bytes begin to be reported, hit the number 1 key to stop the hunt. We are only interested in the first address reported. In this case, it should be $1C15. Using the MEMORY command, inspect memory around the address $1C15. You will notice that the third zero is at the location $1C15. We now have all the information needed to save the new boot to your backup. The start address is $0801 (beginning of Basic) and the end address is $1C16 (all monitors require us to save the actual address plus one:$1C15+l=$1C16). Make sure your backup is in the drive and save the ~emory from $0801-$1C15 <> S "AH",08,0801,1C16 <>.

When the save is complete, you will have a broken copy that will no longer do a protection check. We have essentially replaced the auto boot and the protection check with the result, a Basic boot.

SUPERBOWL SUNDAY : AVALON HILL

Procedure:

Loading the original produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A backup made with a nybbler produces the same non-working copy. Before starting to work on this program, please make a non-working backup of the original.

working with your backup:

1) Start by validating the BAM <> OPENl5,8,15,"V":CLOSEl5 <> to make room for a new file we will be adding later. Scratch The first file from the backup <> OPENl5,8,15,"S0:START":CLOSEl5 <>.

working with your original:

2} Place a write protect tab on the original to ensure its safety during the breaking process.

3) Turn your computer off and insert the reset assembly into the cartridge port. Turn the computer on again and load the boot file and start the load process <> LOAD"START",8,l <>. Allow the

program to load until the game menu is on the screen. At this point, reset the computer.

4) Remove the original disk from your drive and insert the utility disk. Load the $COOO monitor <> LOAD"49l52",8,1 <> . When the load is complete, sys the monitor in with SYS49152. The monitor should be active now. Remove the utility disk from the drive and replace it with the backup work disk.

5) Interpret memory starting at $0801 (I 0801). Scroll through memory and notice the Basic program. Our task is to repair the pointers and save the program to your backup (see scheme B intro). Using the memory command (H 0801) inspect code at 0801. Notice that the first two bytes are 00 00. These two bytes represent the start of the next line in this Basic program. Obviously, these bytes have been destroyed by the reset because the next line couldn't be zero. To find the correct bytes to replace the two zeros, follow this procedure. We know that the first four bytes are pointer bytes ($0801-$0804). We also know that the next time a zero byte appears in memory ($0811), it signals a new line. The next address is the address that the pointer will point to ($0812). Therefore, the first two bytes in this program should be 12 OB because all addresses are read in reverse order. Now we can scroll to the two zeros at $0801 and type over them 12 08 and hit RETURN. The first four bytes starting at $0801 should now be 12 08 00 00 (the 00 DO bytes represent the current line number in reverse DO 00=00 DO; yes, we CAN have a line number 0!). Our BASIC program is now repaired and all that is left is to locate the end of the program and save it to our backup disk. To find the program end, use the HUNT command in your monitor. We'll hunt for the three zero bytes that signal the end of Basic. <>H 0801 8000 DO DO DO <>. As the first bytes begin to be reported, hit the number 1 key to stop the hunt. We are only interested in the first address reported. In this case it should be $OA6E. Using the memory command, inspect memory around the address $OA6E. You will notice that the third zero is at the location $OA7O. We now have all the information needed to save the new boot to our backup. The start address is $0801 (beginning of Basic) and the end address is $0A71 (all monitors require us to save the actual address plus one: $0A70+l=$0A71). Make sure your backup is in the drive and save the memory from $0801-$0A70 <> S"START",08,0801, 0A71 <>.

When the save is complete, you will have a broken copy that will no longer do a protection check. We have essentially replaced the auto boot and the protection check with the result, a Basic boot.

GULFSTRIKE : AVALON HILL

Procedure:

Loading the original produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast copier or a nybbler produces a non-working copy. Before starting, please make a non-working backup of the original.

working with your backup:

1) Start by validating the BAM <> OPENl5,8,15,"V":CLOSEl5 <> to make room for a new file we will be adding later. Scratch the first file from your backup <> OPENl5,8,15,"S0:BOOT":CLOSEl5 <>.

2) Turn off your computer and insert the reset assembly into the cartridge port. Turn the computer on again and remove your backup from the drive. Insert the utility disk and load the $COOO monitor <> LOAD"49152",8,l <>. After the load, sys in your monitor with SYS 49152 and hit RETURN. We want to fill memory from $0801-$2000 with EA so, use the FILL command <> F 0801 2000 EA <>. We now have a marked work space to load our program into. Use the reset button to reset the computer and clear the screen.

working with your original:

3) Place a write protect on the original to ensure its safety during the breaking process.

4) Load the boot file and start the load process <> LOAD"BOOT"18,l <>. Allow the program to load until the screen clears and then turns blue. At this point, reset the computer.

5) Remove the original disk from your drive and insert the utility disk again. Load The $COOO monitor <> LOAD"49152",8,l <>. When the load is complete, sys the monitor in with 5YS49152. The monitor should be active now. Remove the utility disk from the drive and replace it with the backup work disk.

6) Interpret memory starting at $0801 (I 0801). Scroll through memory and notice the Basic program. Our task is to repair the pointers and save the program to our backup (see Scheme B Intro). Using the MEMORY command (M 0801), inspect code at $0801. Notice that the first two bytes are 00 00. These two bytes represent the start of the next line in this Basic program. obviously, these bytes have been destroyed by the reset because the next line couldn't be zero. To find the correct

bytes to replace the two zeros, follow this procedure. We know the first four bytes are pointer bytes ($0801-$0804). We also know that the next time a zero byte appears in memory ($080D), it signals a new line. The next address is the address that the pointer will point to ($080E). Therefore, the first two bytes in this program should be OE 08 because all addresses are read in reverse order. Now we can scroll to the two zeros at $0801 and type over them OE 08 and hit RETURN. The first four bytes starting at $0801 should now be OE 08 OA 00 (the OA 00 bytes represent the line number in reverse OA 00=OOOA =10 in decimal). Our Basic program is now repaired and all that is left is to locate the end of the program and save it to our backup disk. To find the program end, use the HUNT command in your monitor. We'll hunt for the first three EA bytes that signal the end of the program that we loaded in <> H 0801 2000 EA EA EA <>. This search will bring back the address $08A5. Disassembly of code around this address reveals a small machine language program placed under a Basic program. To properly capture all the necessary code, we must save the code from the beginning of Basic ($0801) to the beginning of our EA bytes ($08A5). Because all monitors require us to add one extra byte to the end address, use this command: <> S"BOOT",08,0801,08A6 <>.

When the save is complete, you will have a broken copy that will no longer do a protection check. We have essentially replaced the auto boot and the protection check with the result, a small program consisting of a Basic loader with a machine language routine placed under it.

CREATIVE CONTRAPTIONS : BANTAM

Procedure:

Loading the original produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A backup made with a nybbler produces the same non-working copy. Before starting to work on this program, please make a non-working backup of the original.

Working with your backup:

l) Start by scratching the first file from your backup
   <> OPENl5,8,15, "S0:CREATIVE":CLOSEl5 <>.

Working vith your original:

2) Place a write protect on the original to ensure its safety
   during the breaking process.

3) Turn your computer off and insert the reset assembly into the cartridge port. Turn the computer on again and load the boot file and start the load process <> LOAD"CREATIVE",8 <>. When the cursor appears, type RUN and hit RETURN. Let the program load for about 15 seconds and reset the computer.

4) Remove the original disk from your drive and insert the utility disk. Load the $COOO monitor <> L0AD"49152",8,1 <>. When the load is complete, sys the monitor in with SYS49152. The monitor should be active now. Remove the utility disk from the drive and replace it with the backup work disk.

5) Interpret memory starting at $0801 (I 0801). Scroll through memory and notice the Basic program. Our task is to repair the pointers and save the program to our backup (see Scheme B Intro). Using the MEMORY command (M 0801), inspect code at 0801. Notice that the first two bytes are 00 00. These two bytes represent the start of the next line in this Basic program. Obviously, these bytes have been destroyed by the reset because the next line couldn't be zero. To find the correct bytes to replace the two zeros, follow this procedure. We know the first four bytes are pointer bytes ($0801-$0804). We also know that the next time a zero byte appears in memory ($0818), it signals a new line. The next address is the address that the pointer will point to ($0819). Therefore, the first two bytes in this program should be 19 OB because all addresses are read in reverse order. Now we can scroll to the two zeros at $0801 and type over them 19 08 and hit RETURN. The first four bytes starting at $0801 should now be 19 08 OA 00 (the OA 00 bytes represent the current line number in reverse OA 00=OOOA =10 in decimal). Our Basic program is now repaired and all that is left is to locate the end of the program and save it to your backup disk. To find the program end, use the HUNT command in your monitor. We'll hunt for the three zero bytes that signal the end of Basic. <>H 0801 8000 00 00 00 <>. As the first bytes begin to be reported, hit the number 1 key to stop the hunt. We are only interested in the first address reported. In this case it should be $0879. Using the MEMORY command, inspect memory around the address $0879. You'll notice that the third zero is at the location $087B. We now have all the information needed to save the new boot to our backup. The start address is $0801 (beginning of Basic) and the end address is $087C (all monitors require us to save the actual address plus one: $087B+l=$087C). Make sure your backup is in the drive and save the memory from $0801-$087B <> S"CREATIVE",08,0801,0B7C <>.

When the save is complete, you will have a broken copy that will no longer do a protection check, and will even load faster than the original. We have essentially replaced the auto boot and the protection check with the result, a Basic boot.

INTRO : PROTECTION SCHEME TYPE C

This protection scheme employs the use of a "fat track" to prevent the user from making his backup. To make matters worse, the fat track is placed on the outer (36-40) tracks.

Most of the examples covered in this manual work approximately the same. The following general loading procedure is taken with each.

1) The boot is loaded and autostarts the program.

2) A fast loader is set up and activated.

3) The logo screen is loaded in and activated.

4) The protection routine is decrypted.

5) The files pertaining to the program are loaded in. These are generally encrypted.

6) The protection is checked, which places a numeric value ($FF) in the disk drive's memory.

7) The value is checked using a memory read.

8) The value is used as a part of a decryption routine to decrypt the main program. Proper decryption takes place ONLY if the correct value is returned.

9) The code then jumps to the start of the program.

The Activision examples in this manual represent this protection scheme in it's most difficult form to un-protect. You'll find this same scheme being used by other software publishers, but generally not encrypted. They usually check for the value in the same way and start the program if found. One example of this will be given, and will be unprotected by a different method. Understanding this routine is imperative, because this scheme has been improved, and will be covered in it's expanded form in updates to this manual.

COUNTDOWN TO SHUTDOWN : ACTIVISION

Procedure:

Loading the original produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A nybbled backup produces the same non-working copy. Before starting to work on this program, please make a (non-working) backup of the original, and a disk log to log the file addresses.

1) Turn off your computer and insert your reset button assembly into the cartridge port. Turn the computer on again. Load the $COOO monitor from your utility disk <> LOAD"49152",8,l <>. At the completion of the load, type SYS 49152 and hit RETURN. The monitor should be active now.

Working with your backup:

2) With your backup in the drive and the monitor active, load the boot file <> L "1C0P*",08 <>. When the load is complete, disassemble memory at $02A0. You'll find a loader routine that loads in the 1985 file and jumps to $0B79.

3) Load the 1985 file into memory <> L " 19*",08 <>. After the load, start disassembly of code at $0B79 (D 0B79). The code is as follows: $0B79-$OBCB sets up a fast loader and loads in the logo screen. $OBCC is a JSR (GOSUB in BASIC) to the logo screen. $OBCF is the start of the main program load. It is this code that is of interest to us.

4) The code at $0C40-$0C61 is a decryption routine. Examine it because it is the key to the de-protection. This routine allows decryption and examination of the protection code. At the end of this decryption routine is a RTS ($0C61). Using the Memory command (M 0C61), change the 60 to a 00. This will allow a normal operation of code until the 00 (Break or Stop) is encountered. The program, once started, will stop right after the decryption, allowing us to examine the protection routine.

5) For our purposes, we will skip over the fast loader and logo screens. Let's start the program after the logo screen is run ($OBCF). Type G OBCF and hit RETURN. The screen should turn black. Wait for about five seconds and reset the computer. Return to the monitor with SYS 49152. Using the INTERPRET command, examine code from $OAOO on (I OAOO). Code at $OAEF reveals a Block Execute (executes a protection check routine placed in drive memory) and code at $0B72 reveals a Memory Read

that reads the value placed in the drive by the protection check. This value, in this scheme, is always an $FF. Examine code at $0B42. The value is being returned to the computer by a Memory Read with a kernal routine. The $FFCF routine brings back the value $FF. It is then EORed with $AA which turns it into a $55 and then stores it at location $0B65. Our job is to place the correct value in $0B65 and disable the routine overwriting it. This can be accomplished by placing three NOPs at $0B47 which will allow the routine to Memory Read the value but not place it in computer RAN. All that is left is to place the correct value of $55 at $0B65.

6) Now we have the correct values to plug into the code to disable and give the protection check what it wants. The last step is to place the changes on the disk. This is best done with a sector editor because to scratch and replace the 1985 file will destroy necessary code placed on the disk. This code is not accessed in the normal fashion, so it will be overwritten if we do a scratch and save of the 1985 file. Finish the job by following these steps:

A] We know the code was originally encrypted, so we must place our values on the disk in encrypted form. The three bytes at $0B47 and the single byte at $0B65 are the only changes needed. Reload the 1985 file <> L" 19*" ,08 <>. Again go to location $0B61 and place a 00 in memory. Inspect the three bytes a $0B47. They should be 29 7A 91. The byte at $0B65 should be a A2. These are the bytes we will look for on our backup with the sector editor.

B] The code can now be decrypted by typing G OBCF. Again the screen will turn black. After a few seconds, reset the computer and reactivate the monitor with SYS 49152.Using the MEMORY command (N 0B47), change the code at $0B47 from 8D 65 OB to EA EA EA. change the code at $0B65 from A2 to 55.

C] Now that our changes are in memory, we may re-encrypt the file (and our byte changes) by again typing G OBCF. Again, reset out and SYS the monitor back in with SYS 49152 and hit RETURN. Examine memory at $0B47 and find the encrypted byte changed to SB. Now we know the changes, and the location, so changes. They should be 4E F5 70. The byte at $0B65 has we may now do the actual changes to the backup.

D] Reset the computer and load the sector editor from the utility disk [LOAD"DISK DR",8,l]. When the cursor appears, type run and hit RETURN. Remove the utility disk and place your backup in the drive. flit RETURN. You will be shown track 18, sector 1. By placing the cursor at position 35 you will be on the file pointers of the 1985 file. Press the J key to jump to the beginning of the 1985 file. When the sector comes on the screen, examine the first four bytes. The first two are links to the next sector of the file. The next two are the address bytes in reverse order ($0700). We know our changes are in memory block $OBOO so we can use the N key to page through memory. Press N to go to approximately 0800, press again to go to 0900, press again to go to OAOO, and once more to $OBOO. This block turned out to be track 17, sector 3 on our version. Yours could be in a different location on the disk but the idea will be the same.

E]   Using the cursor key to move through the code, we find the original three bytes 29 7A 91 at location 83. The change to 4E F5 70 can be accomplished with the @ key. The changes must be the decimal equivalent. These are 78 245 112. Change each byte by placing the cursor over the byte to be changed, and type @ and then the decimal number change. Hit RETURN when the change is made to lock it in. When all three bytes are changed, continue searching with the cursor for the A2 byte. This can be found at 113. Using the same change procedure, change it to a decimal 91 ($5B). When all changes have been made and locked in, press C to copy the sector back to the disk.

You now have a copy that can be fast copied. The placement of data on the disk in methods other than directory files will not allow you to file copy. One other point of interest is the fast loader installed in many pieces of this publisher's software. This fast loader is NOT compatible with the 1571 disk drive. In many (but not all) of the programs, you may disable the fast loader and allow the program to load on the 1571 by changing the jump to the main program in the autoboot. Countdown does not work by doing this but, just as an example, you would change the 79 OB (JMP 0B79) to CF OB (use DISK Doctor and the decimal equivalents). This would bypass the fast-loader and the logo screen. A small price to pay for the 1571 owners.

WEB DIMENSION : ACTIVISION

Procedure:

Loading the original produces a rattle free load, and an error scanner shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A backup made with a nybbler produces the same non-working copy. Before starting to work on this program, please make a (non-working) backup of the original, and a disk log to log the file addresses.

1) Turn off your computer and insert your reset button assembly into the cartridge port. Turn the computer on again. Load the $C000 monitor from your utility disk <> LOAD"49152",8,l <>. At the completion of the load, type SYS 49152 and hit RETURN. The monitor should be active now.
working with your backup:

2) With your backup in the drive and the monitor active, load the boot file <> L "COP*",08 <>. When the load is complete, disassemble memory at $02E0. You'll find a loader routine that loads in the 1985 file and jumps to $OC3D.

3) Load the 1985 file into memory <> L "19*",08 <>. After the load, start disassembly of code at $OC3D (D OC3D). The code is as follows: $OC3D-$OC5B sets up a fast loader and loads in the logo screen. $OC5C is a JSR (GOSUB in BASIC) to the logo screen. $OC5F is the start of the main program load. It is this code that is of interest to us.

4) The code at $OCE5-$0D06 is a decryption routine. Examine it, because it is the key to the de-protection. This routine allows decryption and examination of the protection code. At the end of this decryption routine is a RTS ($0D06). Using the MEMORY command (M 0D06), change the 60 to a 00. This will allow a normal operation of code until the 00 (Break or Stop) is encountered. The program, once started, will stop right after the decryption, allowing us to examine the protection routine.

5) For our purposes, we will skip over the fast loader and logo screens. Let's start the program after the logo screen is run ($OC5F). Type G OC5F and hit RETURN. The screen should turn black. Wait for about five seconds and reset the computer. Return to the monitor with SYS 49152. Using the INTERPRET command, examine code from $OAOO on (I OAOO). Code at $OAB6 reveals a Block Execute (executes the protection check placed in drive memory) and code at $OAC2 reveals a Memory Read that reads the value placed in the drive by the protection check. This value is, in this scheme, always an $FF. Examine code at $0A92. The value is being returned to the computer by a Memory Read with a kernal routine. The $FFCF routine brings back the value $FF. It is then EORed with $FF which turns it into a $00 and then stores it at location $OAB5. Our job is to place the correct value in $OAB5 and disable the routine overwriting it. This can be accomplished by placing three NOPs at $0A97 which will allow the routine to Memory Read the value but not place it in computer RAM. All that is left is to place the value of $00 at $OAB5.

6) Now we have the correct values to plug into the code to disable

and give the protection check what it wants. The last step is to place the changes on the disk. This is best done with a sector editor because to scratch and replace the 1985 file will destroy necessary code placed on the disk. This code is not accessed in the normal fashion, so it may be overwritten if we do a scratch and save of the 1985 file. Finish the job by following these steps:

A]   We know the code was originally encrypted, so we must place our values on the disk in encrypted form. The three bytes at $0A97 and the single byte at $OAB5 are the only changes needed. Reload the 1985 file <> L 19*" ,08 <>. Again, go to location $0D06 and place a 00 in memory. Inspect the three bytes at $0A97. They should be 19 8E E8. The byte at $OAB5 should be a 8A. These are the bytes we will look for on our backup with the sector editor.

B]   The code can now be decrypted by typing G OC5F. Again the screen will turn black. After a few seconds, reset the computer and reactivate the monitor with SYS 49152. Using the MEMORY command (M 0A97), change the code at $0A97 from 8D B5 OA to EA EA EA. Change the code at $OAB5 from AC to 00.

C]   Now that our changes are in memory, we may re-encrypt the file (and our byte changes) by again typing G OC5F. Again, reset out and SYS the monitor back in with SYS 49152 and hit RETURN. Examine memory at $0A97 and find the encrypted byte changes. They should be 7E Dl 08. The byte at $OAB5 has changed to 26. Now we know the changes, and the location so we may now do the actual changes to the backup.

D]   Reset the computer and load the sector editor from the utility disk <> LOAD"DISK DR",8,l <>. When the cursor appears, type RUN and hit RETURN. Remove the utility disk and place your backup in the drive. Hit RETURN. You will be shown track 18, sector 1. By placing the cursor at position 35, you will be on the file pointers of the 1985 file. Press the J key to jump to the beginning of the 1985 file. When the sector comes on the screen, examine the first four bytes. The first two are links to the next sector of the file. The next two are the address bytes in reverse order ($OAOO). We know our changes are in memory block $OAOO so we are in the proper block to make our changes. This block turned out to be track 17, sector 2 on our version. Yours could be in a different location on the disk, but the idea will be the same.

E]   Using the cursor key to move through the code, we find the original three bytes 19 8E E8 at location 155. The change to 7E Dl 08 can be accomplished with the @ key. The changes must be the decimal equivalent. These are 126 209 08. Change each by placing the cursor over the byte to be changed, and type @ and the decimal number change. Hit RETURN when the change is made to lock it in. When all three bytes are changed, continue searching with the cursor for the BA byte. This can be found at position 185. Using the same change procedure, change it to a decimal 38 ($26). When all changes have been made and locked in, press C to copy the sector back to the disk.

You now have a copy that can be fast copied. The placement of data on the disk in methods other than directory files will not allow you to file copy. One other point of interest is the fast loader installed in many pieces of this publisher's software. This fast loader is NOT compatible with the 1571 disk drive. In many of the (but not all) of the programs, you may disable the fast loader and allow the program to load on the 1571 by changing the jump to the main program in the autoboot. Web Dimension will work by doing this. Just change the 3D OC (JMP OC3D) to SF OC (use DISK Doctor and the decimal equivalents). This will bypass the fast loader and the logo screen. A small price to pay for the 1571 owners.

FIREWORKS CELEBRATION KIT : ACTIVISION

Procedure:

Loading the original produces a rattle free load, and an error scanner shows no standard errors. A backup made with Three Minute Backup produces a non-working copy. A backup made with a nibbler produce the same non-working copy. Before starting to work on this program, please make a (non-working) backup of the original, and a disk log to log the file addresses.

1) Turn of f your computer and insert your reset button assembly into the cartridge port. Turn the computer on again. Load the $COOO monitor from your utility disk <> LOAD"49152",8,l <>. At the completion of the load, type SYS 49152 and hit RETURN. The monitor should be active now.

working with your backup:

2) With your backup in the drive and the monitor active, load the boot file <> L "COP*" ,08 <>. When the load is complete, disassemble memory at $02E0. You'll find a loader routine that loads in the 1985 file and jumps to $OC3D.

3) Load the 1985 file into memory <> L " 19*",08 <>. After the load, start disassembly of code at $OC3D (D OC3D). The code is as follows: $OC3D-$OC5B sets up a fast loader and loads in the

logo screen. $OC5C is a JSR (GOSUB in BASIC) to the logo screen. $OC5F is the start of the main program load. It is this code that is of interest to us.

4) The code at $OCE2-$0D03 is a decryption routine. Examine it, because it is the key to the de-protection. This routine allows decryption and examination of the protection code. At the end of this decryption routine is a RTS ($0D03). Using the MEMORY command (M 0D03), change the 60 to a 00. This will allow a normal operation of code until the 00 (Break or Stop) is encountered. The program, once started, will stop right after the decryption, allowing us to examine the protection routine.

5} For our purposes, we will skip over the fast loader and logo screens. Let's start the program after the logo screen is run ($OC5F). Type G OC5F and hit RETURN. The screen should turn black. Wait for about five seconds and reset the computer. Return to the monitor with SYS 49152. Using the INTERPRET command, examine code from $OAOO on (I OAOO). Code at $OAB6 reveals a Block Execute (executes the protection check placed in drive memory) and code at $OAC2 reveals a Memory Read that reads the value placed in the drive by the protection check. This value, in this scheme, is always an $FF. Examine code at $0A92. The value is being returned to the computer by a Memory Read with a kernal routine. The $FFCF routine brings back the value $FF. It is then EORed with $FF which turns it into a $00 and then stores it at location $0AB5. Our job is to place the correct value in $OAB5 and disable the routine overwriting it. This can be accomplished by placing three NOPs at $0A97 which will allow the routine to Memory Read the value but not place it in computer RAM. All that is left is to place the value of $00 at $OAB5.

6} Now we have the correct values to plug into the code to disable and give the protection check what it wants. The last step is to place the changes on the disk. This is best done with a sector editor because to scratch and replace the 1985 file will destroy necessary code placed on the disk. This code is not accessed in the normal fashion, so it may be overwritten if we do a scratch and save of the 1985 file. Finish the job by following these steps:

A] We know the code was originally encrypted, so we must place our values on the disk in encrypted form. The three bytes at $0A97 and the single byte a $OAB5 are the only changes needed. Reload the 1985 file <> L " 19*" ,08 <>. Again go to location $0D06 and place a 00 in memory. Inspect the three bytes at $0A97. They should be 19 8E E8. The byte at $OAB5 should be an BA. These are the bytes we will look for on our backup with the sector editor.

B]   The code can now be decrypted by typing G OC5F. Again, the screen will turn black. After a few seconds, reset the computer and reactivate the monitor with SYS 49152. Using the MEMORY command (M 0A97), change the code at $0A97 from 8D B5 OA to EA EA EA. Change the code at $OAB5 from AC to 00.

C]   Now that our changes are in memory, we may re-encrypt the file (and our byte changes) by again typing G OC5F. Again reset out and SYS the monitor back in with SYS 49152 and hit RETURN. Examine memory at $0A97 and find the encrypted byte changes. They should be 7E Dl 08. The byte at $OAB5 has changed to 26. Now we know the changes, and the location so we may now do the actual changes to the backup.

D]   Reset the computer and load the sector editor from the utility disk <> LOAD"DISK DR",8,1 <>. When the cursor appears, type RUN and hit RETURN. Remove the utility disk and place your backup in the drive. Hit RETURN. You will be shown track 18, sector 1. By placing the cursor at position 35, you will be on the file pointers of the 1985 file. Press the J key to jump to the beginning of the 1985 file. When the sector comes on the screen, examine the first four bytes. The first two are links to the next sector of the file. The next two are the address bytes in reverse order ($OAOO). We know our changes are in memory block $OAOO so we are in the proper block to make our changes. This block turned out to be track 17, sector 4 on our version. Yours could be in a different location on the disk, but the idea will be the same.

E]   Using the cursor key to move through the code, we find the original three bytes 19 8E E8 at location 155. The change to 7E Dl 08 can be accomplished with the @ key. The changes must be the decimal equivalent. These are 126 209 08. Change each byte by placing the cursor over the byte to be changed, and type @ and the decimal number change. Hit RETURN when the change is made to lock it in. When all three bytes are changed, continue searching with the cursor for the 8A byte. This can be found at position 185. Using the same change procedure, change it to a decimal 38 ($26). When all changes have been made and locked in, press C to copy the sector back to the disk.

You now have a copy that can be fast copied. The placement of data on the disk in methods other than directory files will not allow you to file Copy. One other point of interest is the fast loader installed in many pieces of this publisher's software. This fast loader is NOT compatible with the 1571 disk drive. In many (but not all) of the programs, you may disable the fast loader and allow the program to load on the 1571 by changing the jump to the main program in the autoboot. Fireworks Kit will work by doing this. Just change the 3D OC (JMP OC3D) to SF OC (use DISK Doctor and the decimal equivalents). This will bypass the fast loader and the logo screen. A small price to pay for the 1571 owners.

RINGS OF ZILFIN : S.S.I.

Procedure:

Loading the original produces a rattle free load, and an error scanner shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A nybbled backup produces the same non-working copy. Before starting to work on this program, please make a (non-working) backup of the original, and a disk log to log the file addresses.

working with your backup:

1) Turn off your computer and insert your reset button assembly into the cartridge port. Turn the computer on again. Load the backup disk <> LOAD "*",8,l <>. Hit RETURN and the program will autoboot. Let the load continue until the screen turns black and the drive comes to a stop. The program has failed protection and has "crashed".

2} Hit the reset button to return the system back to normal. Remove the backup, and insert your utility disk in the drive. Load the $COOO monitor <>LOAD "49152",8,1<> and sys it in by typing SYS 49152 and hit RETURN. When the monitor comes up, use the INTERPRET command to search memory for any drive commands. Start your search at the beginning of BASIC memory (I 0801). scrolling down through memory, keep your attention on the left side of your screen. When you come to the memory at $6FDE, you'll find a B-E (Block Execute) and a M-R (Memory Read). This is the area of memory that contains the protection code.

3) Disassemble memory at $6F77 (D 6F77), and scroll slowly down through the code. The code from $6F7A to $6FDD represents a subroutine that is called from the main program. This code does a Block Execute from track 35 sector 10. This means it loads that block from the program disk into the disk drive memory and executes that routine. At the completion of the routine, the code returns to computer RAM and resumes operation. Upon it's return, a Memory Read of the drive memory is done, looking for a single byte placed in drive memory by the protection check. This byte is transferred from the drive to computer RAM location $6FDD and is then compared to an $FF. If the byte is not an $FF, the code is directed to an endless loop. If it is an $FF, the code continues until a JUMP FFC3 is encountered. Because the

kernal routine FFC3 has been accessed by a JUMP and not a JSR, it forces an RTS in the code flow. This RTS returns the protection check to the main program.

4) Defeating this protection scheme is simple. We can place a RTS at the beginning of the routine. This will short circuit the protection check completely by sending the program flow back to the code that called for it originally. Before changing code, let's find out which file contains the protection check. Looking over the disk log, we find that the file P99 is the only likely candidate. The starting address is $6000 and the ending address is $6FF4. Remove your utility disk from the drive and again insert the backup in it's place. Double check the file by loading P99 directly from the backup <> L "P99",08 <> . Again disassemble code around $6F7A (D 6F7A) and make sure this file is the correct one that has the protection check. When satisfied, use the MEMORY command to change the byte at the address $6F7A (M 6F7A) to a 60. Now scratch and save this file to your backup. Remember to add one byte to the ending address. <> S "@0:P99",08,6000,6FF5 <>.

Your backup is now completely broken. It can be fast copied and, because we have forced the program to not use the protection check, it can even be file copied. Remember, the Block Execute (which now is not used) accesses a specific spot on the disk, and is not picked up by directory files. Finally, note the name placed on the diskette directory. You'll find it on many programs. Now you know the secret of XEMAG 2.0 protection.

Four examples using this scheme have been discussed above. We must assume that you have mastered the techniques used to defeat those titles. Many titles have been released using Fat Tracks. Some were relatively simple to break and others were quite difficult. Some protection programmers have been checking not only for the Fat Track but also to see if either their computer OR drive code had been tampered with. This was done by checksumming. If any sign of tampering was evident, the program refused to run - even if the break code was technically sound. If you have applied the methods in Kracker Jax Vol I to a similar protection, and it refused to work, you can assume they caught you in their code. We are going to give you examples of how to defeat the drive code, computer code, and the checksumming. Be advised, these examples show tricks and techniques that can be used again on other schemes. Breaking protection involves thought and ingenuity.

TITANIC : ACTIVISION

Procedure:

Loading the original disk produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy. A backup made with a nybbler produces the same non working backup. Before starting to work on this program, please make a (non working) backup of the original, and a disk log to log the file addresses.

Working with your backup:

1) Let's start by plugging Hesmon in the cartridge port and loading the boot < L "*" 08 > . Checking with the disk log, start disassembly of code at $02D7 <D 02D7> and cursor down through the code. The code from $O2EE to $0301 opens a channel for loading, sets the file name " 1985 ", loads that file in and Jumps to $4635. We can load that file in ourselves and inspect it.

2) Cursor down to a clear spot and load the 1985 file as < L " 1985*" 08 > . Be sure to use two spaces before the 1985 file name. The disk log shows this file ranges from 4400-46D8. Look at the file in ASCII by using the Interpret command I 4400 and cursor down through memory. Take note of what it looks like, because we will be looking again later. Let's start disassembly at the Jump to $4635 <D 4635> . Cursor down through the code and note code from $4657 to $4668. Values are being set for the decrypter at $466F to $4690 (see Kracker Jax Revealed Vol I for more details). We want to execute the decrypter and stop the execution after the decryption takes place. To do this we must place a 00 (Break Instruction) at $4690. Use the Memory command to make your change <M 4690> and change the 60 to a 00 and hit return. Now we can decrypt the code by executing at $4657. Use the GO command <G 4657>

3) When the monitor breaks, use the Interpret command again starting at $4400 [I 4400] and cursor down through memory again. This time note the Block-Execute at $4571. This command opens channel 2, addresses drive 0, and sends the code at track 3 sector 0 to the RAM of the disk drive ($0400 in this case) and executes the code in the drive. This code is the protection check routine. While in the Interpret mode, also note the Ul (Block-Read) of the same Track 3/Sector 0. This block read is used to checksum the drive code to check for tampering. Checksums throughout the computer code also check strategic areas of the computer code for tampering. If changes in the

original code are found, the program will not run even if the break is correct. Here's a trick to break the drive code and still keep the checksums intact.

4) Turn the computer off and back on again to clear memory. X to BASIC <X> and from the Utility Disk, load the Block Read file < LOAD"BLOCK READ",8 > . When the ready prompt comes up. LIST the file and on line 10 set the TRack variable to 03 and the SEctor variable to 00. Hit RETURN to lock your changes in and relist the file to check your changes. This utility will Block Read Track 3/Sector 0 and send the code to $C000 in the computer where we can inspect it. Place the backup in the drive and start the Block Read by Typing RUN and hitting return. The drive will spin and in about 30 seconds, the ready prompt will appear. Return to the monitor by hitting Run/Stop-Restore. Disassemble code at $COOO <D COOO> . Cursor down through the code. The code from $COOO-$COll is the decryptor and will have to be executed before we can inspect the drive code. You'll see that it is set to decrypt this code in the $0400 Buffer in the drive and must be readdressed to decrypt at $COOO. Using the Memory Command, change the 04 at $C006,$C009, $COOC,and $COOF to CO. Now Disassemble starting at $COOO again and check the decrypter again. It should now be set up to decrypt code in the $COOO buffer.

5) Let's execute the decrypter and inspect code. Type <G Cool>, and when the monitor breaks, Disassemble code at $COOO <D COOO> and cursor down through the code. The code from $C012-$CO4C checks Track 35, bumps the head a half track and if the check is satisfactory, stores a 0 in $0009. The Instruction at $CO4D loads the accumulator with the value in $0009. Next, if that value is not a 0, the code branches around the next two instructions. These are the keys to the protection. The value of $FF is stored at $OlFF in the drive memory. Later a Memory Read in the computer code will check for the $FF and if it is in place at $OlFF, the protection check will be passed. Our job now is to force this routine to pass even if the protection isn't in place. One way would be to place two NOPs ($EA) at $C050 to erase the BNE C057. This would force the code to fall through and store the $FF byte even if protection wasn't passed. This would work, but the checksum would catch us. Here's a trick to force the code to fall through and still pass the checksum.

6) Because the key to this break is the BNE command at $C050, let's flip those bytes and see what instruction comes up. Use the Memory command to change the DO O5 at $C050 to O5 DO <M COSO>. Disassemble $C050 again <D CO5O>. The BNE instruction has now become an ORA DO. This has effectively negated the BNE because this instruction is essentially worthless and performs no task that is actually used. The checksum will also pass because we

haven't actually changed any bytes, only their position. Let's prepare to make our changes to the disk. Turn of f the computer and remove Hesmon.

7) From the utility disk, load and run the Disk Doctor. Place the backup in the drive and using the b [b] command read in Track 3/Sector 0. At position $50 (remember $C050), 80 in decimal you'll find the two bytes that we need to flip. These are $D4 and $01. Remember, these are the bytes in their encrypted form. Change these to $01, $D4. You may use the @ key and the decimal values. Starting at position 80, change two bytes to 01, 212. Hit r <r> to rewrite the block and y <y> for yes. This title is now broken from protection, and may be fast copied. Because of the Block Execute to Track 3/Sector 0, you may not file copy this title. The drive code, even though broken, must be in place on the disk.

ROCKY HORROR SHOW : ACTIVISION

Procedure:

Loading the original disk produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy. A backup made with a nybbler produces the same non working backup. Before starting to work on this program, please make a (non working) backup of the original, and a disk log to log the file addresses.

This break method is presented to add a trick to your arsenal. If it is confusing at first, a little studying of the code will make the break clear. Print-outs of any confusing code may also help to make things clear.

Working with your backup:

1) We will start by filling the BAM with zeros so the drive will be fooled into believing our backup disk is full. This way we can scratch and then save a file back to the disk without overwriting any program code that isn't allocated in the BAM. Use this trick whenever you suspect any hidden files not in the directory.

   Load Disk Doctor from the Utility Disk. Place the backup in the drive and go to Track 18/Sector 0 using - command. This is the BAN sector. Using the @ key, fill position 4 through 71 with zeros (@). Skip over 72 to 75 which is the directory track and fill 76 through 143 with zeros (@). When finished, rewrite the changes to the disk by hitting <r> for rewrite and <y> for yes.

2) With Hesmon in the cartridge port, load the boot < L"*",08 >. Checking with the disk log, start disassembly of code at $02D7 < D 02D7 > and cursor down through the code. The code from $02E8 to $0301 opens a channel for loading, sets the file name " 1985 ", loads that file in and Jumps to $135A. We can load that file in ourselves and inspect it.

3) Cursor down to a clear spot and load the 1985 file as < L " 1985*",08 > . Be sure to use two spaces before the 1985 file name. The disk log shows this file ranges from lO00-143F. Look at the file in ASCII by using the Interpret command [I 1000] and cursor down through memory. Take note of what it looks like, because we will be looking again later. Let's start disassembly at the Jump to $135A < D 135A > . Cursor down through the code and note the decrypter code from $139B to $l3BC. We want to execute the decrypter and stop the execution after the decryption takes place. To do this we must place a 00 (Break) at $1398. Use the Memory command to make your change <M 1398> and change the 4C to a OO and hit return. Now we can decrypt the code by executing at $137D. Use the GO command < G 137D > . After the monitor breaks, use the Interpret command to examine the code from $1OOO-$143F again < I 1000 > . You'll find it to be quite different now and you should be able to see quite a few commands in ASCII. Finally use the Memory command to change 00 we placed at $1398 back to a 4C < M 1398 >

4) Let's trace the code starting at $135A commenting the code pertaining to the protection check.

```
$135A-$1394    : Sets up the decryption values.
$1395 JSR 139B : Executes decryption of  1985  file.
$1398 JMP l3BD : Jump around decrypter already executed.
$l3BD JSR 1184 : JSR to protection check.
$1184 JSR 1206 : Sets up for protection check.
$1187 JSR 118E : checks drive memory for a value of $FF at
                 $OlFF.  EORs that value with an $FF which
                 produces a Zero (0).  Places that zero at
                 $1294. Later the value at $1294 is used in
                 the program decryption.
$118A JSR 1269
$118D RTS
$13C0 JSR 1116 : Continue on.
```

5) This protection would be simple to deprotect if it weren't for the checksums used throughout the code. Every strategic point has been checked and if we are caught tampering with the code, the program won't work, even if the break is sound. We need to trick the checksums. Testing in various spots has uncovered an area that is not checksummed. The decrypter routine is not checked and if moved, will provide us with a work area to place

our code in and short circuit the protection check. Let's begin here.

6} Reload the 1985 file to provide fresh undecrypted code < L "   198*" 08 > . First let's move the decrypter to the outside bounds of this file. Since the file ends at $143F we can move it to $1440. Use the Transfer command < T 139B l3BC 1440 >. Disassemble code at $1440 <D 1440> and cursor down through the moved decrypter. You'll find the last byte, a $60 at $1461. This will become the new end address of this file.

7) Now that the decrypter has been moved, lets prepare the work space. Fill the area from $1395-$13BC with NOPs < F 1395 l3BC EA > . Now let's use the assembler in Hesmon to rewrite the code in our work spot. A printout of the prior code to compare with our changes should make the reasons for our changes clear. We can start writing our code a $139A. Start by using the assemble command < A 139A > . Here's the code to write.

```
 A 139A JSR 1440  : Decrypt code from new decrypter location.
        JSR 1206  : Set up for protection.     --code from here
        LDA #$0O  : Substitutes for protection --to here will
                                               --replace
        STA 1293  : check at $118E             --the JSR 1184 at
        JSR 1269  :                            --$l3BD.
        JMP 13C0  : Jump around JSR 1184 at $13BD, which is no
                    longer needed
```

8) When done, hit return a few times to a clear spot and Disassemble code and check to make sure the changes are correct < D 139A >. If all is well, all that's left is to scratch the old file and save the new. X to BASIC <X> and scratch the 1985 file. < OPENl5,8,15,"S0: 1985   " >. Be sure to use two spaces before 1985 and three spaces after. When done, hit Run/Stop-Restore to re-enter the monitor and save the new 1985 file. Our new start/end addresses are $l000-$1461+1. < S " 1985   " 08 1000 1462 > You're backup is now completely broken and may be fast copied. You can't file copy this title because of the various Block-Executes used in the loader for the fast load routine as well as protection checks. These Block-Executes access code not allocated by directory files.

TRIO : SOFTSYNC

Procedure:

Loading the original disk produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy. A backup made with a nybbler produces the same non working backup. Before starting to work on this program, please make a (non working) backup of the original, and a disk log to log the file addresses. Please note the XEMAG 2.0 in the directory header. This is the signal to you of Fat-track protection.

Working vith your backup:

1) Let's start by plugging Hesmon in the cartridge port and loading the boot < L "*" 08 > . Checking with the disk log, start disassembly of code at $02A7 <D 02A7> and cursor down through the code. The code from $02C3 to $02C9 loads in a file with 7 characters in it's name. Interpret memory at $02A7 [I 02A7] to see that file name. You'll find a name using a combination of regular and reverse characters. Again disassemble memory at $02A7 <D 02A7> and cursor down through the code. At $02F7 you'll find a jump to $A483 which causes BASIC to execute.

2) Power off and on again. When the monitor appears, <X> to BASIC and load and list the directory < LOAD "$",8 > . Near the end, you'll find the file with regular and reverse characters. Load that file directly from the directory with a <,8:> . When the READY prompt comes up, cursor down to a clear spot and list that file. Examination of this file shows that it loads and runs the TRIO CALC, TRIO WORD, OR TRIO FILE depending on the menu choice picked by the user.

3) Again cursor down to a clear spot and load TRIO FILE < LOAD "TRIO FILE",8: > . List out this file for examination. This program loads the file TRIO3, does a sys 32768 ($8000) to it, comes back, and reads drive memory at $OlFF and compares the value there to a (2 up arrow 8-1) which is a decimal 255 or $FF. If the value is not equal to an $FF, a NEW occurs which crashes the program. If it is equal to $FF then the program falls through to a GOTO 70. (You'll find similar programming in the TRIO WORD and TRIO CALC files.)

4) Because the file TRIO3 resides at $8000, which is where our Hesmon cartridge resides, we must use a different monitor. Turn off the computer and pull the Hesmon. From the Utility Disk, load the $2000 monitor < LOAD "8192",8,l > . WHEN THE READY prompt comes up, sys the monitor in <SYS 8192> . Load the TRIO3 file from the TRIO backup <L NTRIO3II,08> and start disassembly at $8000 <D 8000> . The code from $8000 to $8036 does a BLOCK EXECUTE to Track 35/Sector 10/. $8037 to $8062 NEMORY READS the drive at location $O1FF and compares to an $FF. If the value is not equal to an $FF, then a branch to $8070 takes place. To see what happens, cursor to a clear spot and do a Go $8070 <G 8070>

When done, hit Run/Stop-Restore and again sys the monitor in with <SYS 8192> . Again disassemble code at $8000 and cursor down through the code. You'll find that if the comparison to $FF is satisfactory, the programming falls through to $808B, which is a JUMP to $FFC3. This is a KERNAL routine that when JUMPed to, does a RTS which in this case returns the program flow back to the basic program (TRIO FILE in this case.).

5) Turn the computer off, insert the Hesmon, and power up again. X to BASIC <X> and from the Utility Disk, load the Block Read file < LOAD"BLOCK READ",8 > . When the ready prompt comes up. LIST the file and on line 10 set the TRack variable to 35 and the SEctor variable to 10. Hit RETURN to lock your changes in and relist the file to check your changes. This utility will now Block Read Track 35/Sector 10 and send the code to $COOO in the computer where we can inspect it. Place the backup in the drive and start the Block Read by Typing RUN and hitting RETURN. The drive will spin and in about 30 seconds, the READY prompt will appear. Return to the monitor by hitting Run/Stop-Restore. Disassemble code at $COOO <D COOO> . Cursor down through the code. The code from $COOO-$CO1O is the decryptor and will have to be executed before we can inspect the drive code. You'll see that it is set to decrypt this code in the $0400 Buffer in the drive and must be readdressed to decrypt at $COOO. Using the Memory Command, change the 04 at $C005 and $COOB to Co. Notice the ADC $08 at $C007. This instruction uses the value in the drive at location $08 to help decrypt this code. The location $08 is the track value last loaded into the Buffer at $0400. We know that this was track 35 (remember the BLOCK EXECUTE to Track 35/Sector 10). Let's change the instruction from a ADC $08 to a ADC #$23. We are now using the known value of $23 (decimal 35) and not using any values in drive memory. The bytes for this instruction change are $69,$23. Use the MEMORY command to make your changes at $C007 < M C007 >. Again disassemble memory at $COOO and cursor down through the code to check to see the changes are correct.

6) Let's execute the decrypter and inspect code. Type <G COOl>, and when the monitor breaks, disassemble code at $COOO <D COOO> and cursor down through the code. The code from $CO11-$C043 checks Track 35, bumps the head a half track and if the check is satisfactory, stores a 0 in $0009. The instruction at $C044 loads the accumulator with the value in $0009. Next, if that value is not a 0, the code branches around the next two instructions. These are the keys to the protection. The value of $FF is stored at $O1FF in the drive memory. Later a Memory Read in the computer code will check for the $FF and if it is in place at $OlFF, the protection check will be passed. Our job now is to force this routine to pass even if the protection isn't in place.

7) One way to break this code is to write a simple routine to place an $FF in drive location $O1FF and return to the programming that sent it in the first place. This is accomplished simply. Cursor down to a clear spot and go into the ASSEMBLY mode by typing <A COOO> . Here's the code:

```
     A COOO LDA #$FF <RET> (A9 FF)
     A C002 STA 01FF <RET> (8D FF 01)
     A COOS RTS  <RET> (60)
```

   When done, cursor down to a clear spot and disassemble at $CO00 [D COOO] to see the bytes needed. You'll find the following six bytes: A9 FF 8D FF Ol 60. You can use the hex to decimal converter in Hesmon to convert the bytes to decimal <$A9 RET, and so on>. You'll find that the following is the decimal equivalent: 169 255 141 255 Ol 96.

8) From the Utility Disk, load and run the Disk Doctor. Place the backup in the drive and using the b [b] command read in Track 35/Sector 10. Starting at position $00, write in the six bytes. You may use the @ command to write them one at a time in Decimal (169 255 141 255 Ol 96). When the changes have been made, hit r <r> to rewrite the block and y <y> for yes. This title is now broken from protection, and may be fast copied. Because of the Block Execute to Track 35/Sector 10, you may not file copy this title. The drive code, even though broken, must be in place on the disk.

ALIENS : ACTIVISION

Loading the original produces a rattle-free load, and an error scanner shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A nybbled backup produces the same non-working copy. Before starting to work on this program, please make a (non-working) backup of the original, and a disk log to log the file addresses.

1) Turn off your computer and insert your reset button assembly into the cartridge port. Turn on the computer again. Load the $COOO monitor from your Utility Disk < LOAD"49152",8,1 >. At the completion of the load, type < SY549152 > and hit < RETURN >. The monitor should be active now.

2) With your backup in the drive and the monitor active, load the boot file < L "0:*",08 > . When the load is complete, disassemble memory at $O2CB. You'll find a loader routine that loads in the

"ACTIVISION INC." file and jumps to $8000.

3) Load the "ACTIVISION INC." file into memory < L"A*",08 >. After the load, start disassembly of code at $8000 < D 8000 > . Also do an ASCII dump < I 8000 > to check for DOS commands. Examine the routines carefully. You will soon find a Block-Execute (B-E 2,0,18,7) drive command at $8ODD. Further examination of the code reveals that the protection scheme is doing a lot of direct access to the serial port at $DDOO.

The key to cracking this variation on Activision's standard protection scheme is to ignore this code because it has a rather involved loop that is a pain to follow and de-protect. With this code, the drive is where the action's at. Let's take a closer look at that Block-Execute code on track/sector 18/7.

4) Reset the computer and load ALIENSLOADER from the Utility Disk < LOAD "ALIENSLOADER",8 > , < RUN > and follow the instructions. Reload the 49152 monitor and < SY549152 > . In the drive, the code would be located at $0300. We will be using $2300 (in the computer). Disassemble the code at $2300 < D 2300 > . The routine at $2322 - $234A, despite it's apparent complexity, does nothing more than load the code in track/sector's 18/7 - 18/11 into drive memory locations $0400 - $O7FF. The ALIENSLOADER routine has conveniently loaded these for us already. The code, from $2400 - $27FF, is encrypted. A routine at $2356 does the decryption. We can modify the code to decrypt it for us by simply adding $2000 to the LDA and STA address references, i.e. $0400 becomes $2400, $0500 becomes $2500, etc..< A 2358 LDA $2400,Y etc.. > . Also put a break command at $237F < A 237F BRK > and run the code < G 2356 >

NOW examine the code starting at $2400 < D 2400 > . Most of this code is the fast loader. Armed with the knowledge that Activision fat tracks start with track 35 ($23), we find a suspicious routine at $24D0 - $24F8. This is it, folks. This itty-bitty loop is the heart and soul of this protection scheme. It can be disabled easily with one byte change. Change the LDA operand byte at $24DE from $80 to $01 < A 24DD LDA #$01 >. Instead of READING the intended sector, the $01 byte tells the drive's DOS that the job was completed successfully. This is exactly what you want it to do. The fringe benefit of this method is that the program loads about 8 seconds faster and you'll hear a pleasant clicking noise when the protection scheme executes the code with your byte change (when the screen blanks).

5) Re-encrypt the code using the same routine at $2356 < G 2356 > Before we load up the sector editor to write the bytes back, let's look back at the decryption loop at $2356 < D 2356 >

it's exchanging bytes between $2400) $2500 and $2600) $2700. Our changed byte (now $54) is at $25DE, -not- at $24DE. It will be written to track/sector 18/9 at position $DE (222 decimal).

6) Now reset the computer, re-insert the Utility Disk and reload the sector editor < LOAD"DISK D*",8 > . Insert your backup and [RUN]. Press the < B > key. Enter 18 < RETURN > and 9 < RETURN > to read in track/sector 18/9. Move the cursor to position 222 and press the @ key. Enter 84 and press < RETURN > To write the modified sector, press < R and Y >

7) Reset and load the backup. It DOES load faster than the original, doesn't it?

TRANSFORKKRS : ACTIVISION

Procedure:

Loading the original produces a rattle-free load, and an error scanner shows no standard errors. A backup made with the C-64 Fast Copier produces a non-working copy. A nybbled backup produces the same non-working copy. Before starting to work on this program, please make a (non-working) backup of the original, and a disk log to log the file addresses.

1) Turn off the computer and insert your reset button assembly into the cartridge port. Turn on the computer again and load the $C000 monitor from your Utility Disk < L0AD"49152",8,1 > . At the completion of the load, type < 5Y549152 > and hit < RETURN>. The monitor should be active now.

2) With your backup in the drive and the monitor active, load the boot file < L"COP*",08 > . When the load is complete, disassemble memory at $02E0. You'll find a loader routine that loads in the " 1986 " file and jumps to $0506.

3) Because the " 1986 " file loads into screen memory where we normally can't look at it, we must first change the load address to something more accessible. Reset the computer, insert the Utility Disk and load the sector editor < LOAD "DISK D*",8 > Insert your backup disk and < RUN > . Go to track/sector 18/01 < B 18 RETURN 1 RETURN >. Cursor over to position 35. The first sector of "1986" is 17/01 ($11/Ol-hex). Jump to there < j > Move to position 3, press the '@' key and change the byte $05 to 37 ($25) and press RETURN. This changes the load address to $2500. Write the sector back to disk < R Y > and reset your computer.

4) Again insert the Utility Disk and load and activate the 49152 monitor. Load the " 1986 " file into memory < L "19*",08 > After the load, start disassembly of code at $2500 < D 2500 > Also do an ASCII dump < I 2500 > to check for DOS commands. Examine the routines carefully. You will soon find a Block-Execute (B-E 2,0,1,1) drive command at $271E. Further examination of the code reveals that the protection scheme is doing a lot of direct access to the serial port at $DDOO. The key to cracking this variation on Activision's standard protection scheme is to ignore this code because it has a rather involved loop that is a pain to follow and de-protect. With this code, the drive is where the action's at. Let's take a closer look ~t that Block-Execute code on track/sector 1/1. (Before going on to step five, change the load address of the " 1986 " file back to $0500. Use the same procedure as outlined in step 3.

5) Reset the computer and load TRANSLOADER from the Utility Disk < LOAD "TRANSLOADER" ,8 >, < RUN > and follow the instructions. Reload the 49152 monitor and < 5Y549152 > . In the drive, the code would be located at $0300. We will be using $2300 (in the computer). Disassemble the code at $2300 < D 2300 > . The routine at $2321 - $2349, despite it's apparent complexity, does nothing more than load the code in track/sector's 1/2 - 1/5 into drive memory locations $0400 - $O7FF. The TRANSLOADER routine has conveniently loaded these for us already. The code, from $2400 - $27FF is encrypted. A routine at $2355 does the decryption. We can modify the code to decrypt it for us by simply adding $2000 to the LDA and STA address references, i.e. $0400 becomes $2400, $0500 becomes $2500, etc... < A 2357 LDA $2400,Y etc.. > . Also put a break command at $237E
< A 237E BRK > and run the code < G 2355 >

NOW examine the code starting at $2400 < D 2400 > . Most of this code is the fast loader. Armed with the knowledge that Activision fat tracks start with track 35 ($23), we find a suspicious routine at $24B4 - $250F. This is it, folks. This itty-bitty loop is the heart and soul of this protection scheme. It can be disabled easily with one byte change. Change the LDA operand byte at $24C2 from $80 to $01 < A 24C1 LDA #$0l > Instead of reading the intended sector, the $01 byte tells the drive's DOS that the job was completed successfully. This is exactly what you want it to do. The fringe benefit of this method is that the program loads about 8 seconds faster and you'll hear a pleasant clicking noise when the protection scheme executes the code with your byte change (when the title screen appears).

6) Re-encrypt the code using the same routine at $2355 < G 2355 > Before we load up the sector editor to write the bytes back,

let's look back at the decryption loop at $2355 < D 2355 > it's exchanging bytes between $2400) $2500 and $2600) $2700. Our changed byte (now $54) is at $25C2, -not- at $24C2. It will be written to track/sector 1/3 at position $C2 (194 decimal).

7) Now reset the computer, re-insert the Utility Disk and reload the sector editor < LOAD "DISK ?*",8 > . Insert your backup and < RUN > . Press the [B] key. Enter 1 < RETURN > and 3 < RETURN > to read in track/sector 1/3. Move the cursor to position 194 and press the < @ > key. Enter 84 and press < RETURN > . To write the modified sector, press < R > and < Y >

8) Reset and load the backup. It DOES load faster than the original, doesn't it?

INTRO : PROTECTION SCHEME TYPE D

When this protection scheme was first introduced, the copy programs available were unable to backup any software that used it.

Most of the nybble utilities on the market today have the capability of producing a backup. This scheme is usually referred to as the "long sector". The following similarities are characteristic of this protection. A nybble utility can back up the title, while a fast copier can't. The load is rattle free and smooth. An error scan produces a number twenty read error on the last sector of any particular track.

This protection is based on placing an extra sector on any chosen track (sometimes more than one track) on the original disk. This sector contains one block of valid program data. A non-nybbler or file copy utility will not pick up this sector, because it is not standard disk format. This will prevent the program from operating properly. Our job in each of the following programs will be to gather the block of data and place it in the program at the proper location.

The protection itself is nothing more than a special Block Read set up to read the non-standard block of data. The routine almost always starts out as an encrypted block. This block begins as a decryption routine that decrypts one block of data. This, in turn, reveals a protection check that does nothing more than read in the long sector and place that long sector data directly over itself. By doing this, the valid code completely hides the protection check itself.

Recognizing the decryption routine is the best way to locate the protection check. Once located, we will start the routine up and let it gather the data we need to break the title. Then a simple memory save is all that's needed to complete the job.

The benefit of breaking the programs using this protection scheme is the fact that almost all of them are file copyable afterwards. This means they can be placed on a disk with other programs.

Please note that this protection scheme is very important to understand. The reason for this is the fact that there is a new scheme now on the market that very closely resembles it. This new scheme is NOT copyable by any nybble utility and must be hand broken. You'll find this new scheme discussed in the next chapter.

IMPOSSIBLE MISSION : EPYX

Procedure:

Loading the original produces a rattle free load, and an error scan shows a number twenty error on track 16, sector 20. A backup made with the C-64 Fast Copier provides a non-working backup. Nybble utilities are capable of providing a backup. Loading the backup results in a load that stalls rather quickly. We can assume the protection is in the loader file. Before starting to work on this title, please make a backup and do a disk log (print-out is best).

working with your original:

1) Turn off your computer and insert your reset button assembly into the cartridge port. Turn the computer on again and, from the utility disk, load the $8000 monitor <> LOAD "32768",8,1 <>. Sys the monitor in with SYS 32768 and hit RETURN. Let's begin by loading and inspecting the boot file <> L "RUN ME",08 <>. At the end of the load, start disassembly at $02A7 (D 02A7). Scroll down through the code and notice that the boot loads the file LOADER (LO*) and jumps to $BOOO.

2) Load the LOADER file <> L "LO*",08 <>. Because this tile resides in the BASIC interpreter location, we must turn BASIC off before we can examine any code. Change address location $0001 from 37 (77 on C-128) to 36 (76 on C-128). Use the MEMORY command (M 0001) to make your change. When the change has been made, we can inspect the code beginning at $BOOO.

3) Disassemble starting at $BOOO (D BOOO) and inspect the code from $BOOO to BOOF. This is a decryption routine and is the heart of this protection scheme, as discussed in the introduction. Our

job will be to trade the protection code for the valid program code. Believe it or not, this is the easy part.

4) Make sure you have a write protect on your ORIGINAL and that the original is in the disk drive. Start the program working by typing GO BOOO and hit RETURN. The drive should start up and, a few moments later, the screen should change colors. At this point, reset your computer and turn the disk drive off and on again. Re-SYS the monitor back in (SYS 32768) and again turn off BASIC as described above. Disassemble code at $BOOO (D BOOO) again and note that the code has, indeed, changed. The encrypted code has been replaced with loader code. All that's left now is to save the file back to the backup.

Working with your backup:

5) Checking the disk log provides the start and ending addresses ($BOOO-$BlA2) to the LOADER file. When saving it, be sure to add one byte to the end address <> S "@0:LOADER",08,BOOO,BlA3 <>.

   Your backup is now protection free and may be file copied. One small problem remains. That is the directory. The repair for this is simple. Using the Name/Id Changer on the utility disk, change the disk name AND the ID number. You must use five figures when changing the ID number. For example, you could name the disk IMPOSSIBLE and renumber it IM 2A. When this is completed, your break will be complete and even the directory can be 'viewed.

BREAK DANCE : EPYX

Procedure:

Loading the original produces a rattle free load, and an error scan shows a number twenty error on track 16, sector 20 and track 15, sector 20. A backup made with the C-64 Fast Copier provides a non-working backup. Nybble utilities are capable of providing a backup. Before starting to work on this title, please make a backup and do a disk log (print-out is best).

Working with your original:

1) Turn of f your computer and insert the reset assembly into the cartridge port. Turn your computer on again. From your utility disk, load the $8000 monitor <> LOAD "32768" ,8,1 <>. Now, type NEW, and hit RETURN. When loading the boot file on this disk, it will autoboot and continue running. In order to inspect it, here's a trick to use. We're going to load the autoboot into BASIC memory for the purposes of inspection. Load the boot file

this way: <> LOAD "BOOT",8 <>. When the load is complete (you may have to hit RUNSTOP/RESTORE), sys the monitor in with SYS 32768 and hit RETURN. You can now find the boot file in BASIC memory at $0801. Interpret memory and scroll down from $0801 (I 0801). Notice the INTRO. Disassembly of memory at $0801 (D 0801) and scrolling down reveals a loader file that loads the INTRO file and jumps to $2015.

2) Load the INTRO file <> L "INTRO",08 <>. Start by disassembling memory at $2015 (D 2015). Scroll down through memory, and at $201A note the JSR $26B9. Disassemble $26B9 (D 26B9). Here we find the decryption routine that is the heart of this protection scheme. Refer to the Introduction for general information on this. Our task is to replace the encrypted data with valid program data. This is relatively easy. Be sure you have a write protect on your original and that the ORIGINAL is in the drive. Type G 26B9 to start the program up. The drive will run for a short time, and then stall. When the drive stops, reset the computer and re-SYS the monitor back in (SYS 32768). Disassemble memory at $26B9 again and notice that the code has indeed changed. This is the valid program code we needed for the break.

Working with your backup:

3) Now, all that's left is to save the retrieved data back to the backup. Checking the disk log provides the start and end addresses of $2000-2A00. Be sure to add one byte to the end address and save it to the backup
<> S "@0:INTRO",08,2000,2A01 <>.

4) Turn the computer off and on, and boot up your backup. It should load past the point that it loaded before our break. Unfortunately, It still refuses to load fully. Remember, we did find two separate number twenty errors on the original. We have disabled half of the protection, now let's do the rest.

5) Reload the $8000 monitor <> LOAD "32768",8,1 <> Sys it in with SYS 32768. From the half broken BACKUP, reload the INTRO file <> L "INTRO" 08 <> . Again, start your disassembly at $2015 (D 2015). Scroll down, and try to follow the program flow. At $2140 you'll find a JUMP $COOO. Using the MEMORY command change the 4C at $2140 (M 2140) to 00 and hit RETURN. This will stop or BREAK the program flow just before it jumps to $COOO, allowing us to inspect memory in the LOADER file. Activate the INTRO file by typing GO 2015.

6) When the drive stops, reset the computer and reload your $8000 monitor <> LOAD "32768",8,1 <> . Sys it in with SYS 32768. Start by disassembling the code at $COOO (D COOO). You'll find a jump to $C024. Disassembly of $C024 reveals another decryption

scheme. This is the second protection routine.

Working with your original:

7) Place the original disk in the drive and Type GO C024 to start the program up. The drive should start up and in a short time the game menu will come on the screen. At this point, reset the computer and re-SYS the monitor back in with SYS 32768.

Working with your backup:

8) Checking the disk log provides us with the start and end address of the LOADER file. Again, remember to add an extra byte to the end address. Save it back to your BACKUP <> S "@0:LOADER",08,COOO,CF81 <>. When the save is complete your backup will be completely broken. One small problem remains. The directory cannot be read properly. To fix it easily, just use the NAME/ID CHANGER on your utility disk. Be sure to use five figures when you give it a new ID number. For example, you could name it BREAK DANCE and renumber it BD 2A.

PIT$TOP II : EPYX

Procedure:

Loading the original produces a rattle free load, and an error scan shows a number twenty error on track 16, sector 20. A backup made with the C-64 Fast Copier provides a non-working backup. Nybble utilities are capable of providing a backup. Before starting to work on this title, please make a backup, format a blank work disk, and do a disk log (print-out is best).

Working with your original:

1) Turn off your computer and insert your reset button assembly into the cartridge port. Turn the computer on again and, from the utility disk, load the $C000 monitor <> LOAD "49152",8,1 <>. Sys the monitor in with SYS 49152 and hit RETURN. Let's begin by loading and inspecting the boot file <> L "PITSTOP",08 <>. At the end of the load, start disassembly at $02A7 (D 02A7). Scroll down through the code and notice that the boot loads the file RUN ME and jumps to $0820.

2) Load the RUN ME file <> L "RUN ME",08 <> . Disassemble memory starting at $0820 (D 0820) and scroll down through the code. This file loads all game files and then at $08E4 does a jump to $9403. The disk log tells us this address is located in the PITS file. Load the PITS file <> L "1PITS",08 <>. When the load is

complete, we can start our inspection at $9403.

3) Because this file occupies memory in the BASIC interpreter ($AOOO-$BFFF), we have to turn BASIC off. This can be accomplished by changing $0001 from a 37 (77 on the C-128) to a 36 (76 on the C-128). Use the MEMORY command to make your changes (M 0001). When done, start disassembly at $9403 (D 9403). You'll find a decryption scheme at this location ($9403-$9412) that is the heart of this protection scheme (see the Introduction). Make sure your ORIGINAL has a write protect tab on it and is in the drive. Start the program working by typing G 9403 and hit RETURN. The drive should start up and a few moments later, the game menu should come on the screen. At this point, reset your computer and remove the original disk from the drive. From the utility disk, reboot the $COOO monitor and sys it in again (SYS 49152). Again turn off BASIC. Now place your formatted work disk in the drive and save the changed code from $9403-$9512 <> S "SECTOR",08,9403,9512 <> When the save is complete, remove the work disk from the drive.

Working with your backup:

4) Complete the break by following the steps below.

A] Reset the computer. Place your backup in the drive and scratch the PITS file <> OPENl5,8,15,"S0:PITS" <>. Re-SYS the monitor back in (SYS 49152).

B] From the original disk, load the PITS file <> L "PITS",08 <>.

C] From the work disk, load the saved SECTOR file <> L "SECTOR",08 <>. This will lay the code retrieved from the break process over the encrypted protection check code.

D] Turn off BASIC again as described above.

E] Place your backup in the drive and save the PITS file now in memory <> S "PITS",08,l000,COOO <>.

Your Backup is now broken. All that's left is to repair the directory. This can be accomplished easily with the NAME/ID CHANGER on the utility disk. Be sure to use five figures in the new ID number. For example, you could name the disk PITSTOP and renumber it PS-II. When this is complete, you can view the directory and file copy this title.

THE BODY TRANSPARENT : DESIGNWARE

Loading the original produces a rattle free load, and an error scan shows a number twenty error on track 32, sector 16. A backup made with the C-64 Fast Copier provides a non-working backup. Nybble utilities are capable of providing a backup. Before starting to work on this title, please make a Three Minute Backup, and do a disk log (print-out is best).

Working with your backup:

1) Let's begin our break by preparing the backup to receive the changes we will be making. From your utility disk, load the NAME/ID Changer and rename and re-ID your backup. Be sure to use five figures in the new ID. For example, you could name the backup BODY TRANS and number it BT-2A. This will make the directory listable.

2) Because this program does not use directory files to store information, we run the risk of overwriting program code when we save our changes to the backup. There is a sure way to avoid this. That is to allocate or use all available blocks in the BAM. What we are going to do is fool the drive into believing that there are no blocks free on the disk. When we scratch a file, the blocks used by THAT file will become free for use. Then when we save that file back to the disk, they will be placed on the exact same blocks that they came from.

3) From the utility disk, load and run DISK DR. Place the backup in the drive and press RETURN to get to track 18, sector 1. Press - to go to track 18, sector 0. This is the BAM Sector and here is where we will allocate all blocks. Use the cursor key to cursor to position 4 (all references will be in decimal). With the cursor on position 4 press the @ key and then press 0. Repeat the @ key and 0 key until all values from position 4 through 71 are changed to zero. This takes care of tracks 1 through 17. Now cursor over to position 76 and do the same changes from position 76 through 143. This will take care of tracks 19 through 35. Now, to make the changes to the disk, press R and then Y and hit RETURN. The new BAM is now on the backup. Your backup is now ready to receive new information. Load the directory and check it. You should have a listable directory with zero blocks free.

Working with your original;

4) Turn off the computer and insert the reset assembly into the

cartridge port. Turn the computer on again and load the boot file from the original <> LOAD "DWARF",8: <>. You can list this file and inspect it. You'll find it loads the file called BOOT2 and then a SYS 49152 ($COOO).

5) From your utility disk, load the~$2000 monitor <> LOAD "8192",8,1 <>. Sys it in with SYS 8192. Now load the BOOT2 file <> L "BOOT2"108 <> and start disassembly at $cooo (D COOO). The first instruction at $COOO is a JSR to $C028. Disassemble $C028 (D C028) and here you'll find the decryption routine that is the heart of this protection scheme. It resides from $C028 to $C037. The break itself is very simple. Make sure you have a write protect tab on the ORIGINAL and that it is in the drive. Start the program by typing G C028 and press RETURN. The drive will spin for a short time and then stop. At this point, reset the computer and re-SYS the monitor back in with SYS 8192. Again disassemble code at $C028. You should find new code in the place of the encrypted code. All that's left is to save this broken loader back to the backup.

working with your backup:

6) Reset the computer and place your prepared backup in the drive. Scratch the BOOT2 file <> OPENl5,8,15,"S0:BOOT2" <> . Re-SYS the monitor in with SYS 8192. The disk log provides the start and end addresses of the BOOT2 file. Be sure to add one byte to the end address. With your backup in the drive, save the BOOT2 file back to the backup <> S "BOOT2",08,COOO,C151 <>.

Your backup is completely broken and can now be copied with any whole disk copier. Unfortunately, it remains non-file copyable because of the way the programers set up the disk files.

INTRO : PROTECTION SCHEME TYPE K

This protection scheme is, at this writing, one of the most effective and prevalent methods of defeating today's nybble copiers. When you know what to look for, you'll find this scheme is being employed by many different software houses. I like to think of this protection as the "big brother" of the long sectors discussed in the previous section.

This scheme can be recognized by the following similarities. When a disk error check is done, no write errors will be found on the original. When booted, no drive rattle will be encountered. The program cannot be backed up with either a fast copier or a nybbler. Usually, you will find data in the directory other than normal directory data. Most important: when tracing the program through it's loading process, you will generally run into a decryption routine and a sector or two of encrypted code.

When this encryption is located, you can be sure it is hiding the protection check code.

Remember, I stated that a sector or two in memory will be encrypted, and that this area in memory surely contained the protection check. Well, one other thing needs to be mentioned.

This is the fact that this encrypted memory starts out as garbled code, then decrypts into a protection check routine and finally after the protection check has been satisfied, is REPLACED with valid programcode. This code, as previously stated, is one or two sectors in length and can be found anywhere on the program disk. You'll find that the directory track (track 18) is the most likely spot. In most cases, we can let the program insert the hidden code in it's proper place. Then a memory save and replacement over the encrypted code in the proper file will not only defeat protection but will totally remove the check for it.

Most of the programs protected with this scheme can be defeated with a simple memory save, but a few have had to have some of the code re-written by hand. This is relatively uncommon and cannot be explained in the scope of this manual.

Experience will prove to be the best teacher.

Before starting to work on the following programs, please do a disk file log (print out is best), format a blank work disk, and have a (non-working) backup available. Please make sure you have a write protect tab on your original program disk as you will be using it in the breaking process. Now let's get on to the specifics.

INFILTRATOR : NINDSCAPE

Procedure:

Loading the original produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier provides a non-working backup. Nybble utilities also provide a non-working backup. Loading the backup results in a load that stalls rather quickly. We can assume the protection is in the loader file. Before starting to work on this title, please make a backup and a disk log (printout is best).

working with your original:

1) Place a write protect tab on your original to protect it during the breaking process.

2) Turn off your computer and insert the reset assembly into the cartridge port. Turn your computer on again. From your utility disk, load the $C000 monitor <> LOAD "49152",8,l <>. When the load is complete, sys the monitor in with SYS 49152. When loading the boot file on this disk, it will autoboot and continue running. In order to inspect it, here's a trick to use. We're going to load the autoboot in BASIC memory for the purposes of inspection. With the monitor active, type X and hit RETURN. You are now back to BASIC. Type NEW and hit RETURN. Now load the boot file this way: <> LOAD "INFILT*",8 <>. When the load is complete, return to the monitor by hitting RUNSTOP/RESTORE. Then re-SYS the monitor back in with SYS 49152. You can now find the boot file in BASIC memory at $0801. Interpret memory and scroll down from $0801 (I 0801). Notice the INTRO. Disassembly of memory at $0801 (D 0801) and scrolling down reveals a loader file at $082D-$0854. This loader loads the INTRO file and jumps to $0880.

3) Load the INTRO file <> L "INTRO",08 <>. When the load is complete, disassemble memory at $0880 (D 0880). Scroll down through memory to $089A. You'll find a JSR 0A25. Disassemble $0A25 (D 0A25) and scroll down to $0A25. Here you'll find a JSR 0C18. Disassemble $0C18 (D 0C18) and notice that we have just run into a decryption routine. Inspect this routine because this is the heart of this protection scheme. Scroll down through the code and notice that it is garbled for about one sector ($0C18-$0D18). As mentioned in the introduction, this code is an encrypted protection scheme that will decrypt into a protection checker and then load valid program code over itself. This will not only allow the program to operate properly, but will also hide the protection code from the curious.

4) The break is fairly simple now that we know where the protection is. Start the program code up by typing G 0C18 and hit RETURN.

The drive should start up and run for a short time. When the drive stops, turn the drive OFF and ON again and reset the computer with your reset button. Restart the monitor by again typing SYS 49152 an hit RETURN. Now go back and disassemble code at $0C18 again (D 0C18). Surprise; the code has changed into good code. To get an idea what is there, interpret memory at $0C18 (I 0C18) and scroll down through memory. You'll see that this is the completion of the loader file. All the data needed to run the loader file properly is now in memory. All that is left to do is replace the INTRO file on the disk with the INTRO file NOW in memory. This can be accomplished with a small memory save. From the disk log, we know that the INTRO file starts at $0880 and ends at $16C3. Remove the original disk from the drive and insert your backup in it's place. Replace the INTRO file now in memory with the one now on your disk. Remember to add one byte to the ending address <> S"@0:INTRO",08,0880,16C4 <>.

5}   Your backup is now broken and will not even check for protection. For those wishing to look at the protection check code, redo the steps above but when you type G OCl8, reset the computer in about one second. If the drive is allowed to run more than a moment or two, the protection code will be hidden.

BOP 'N WRESTLE : KINDSCAPE

Procedure:

Loading the original produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier provides a non-working backup. Nybble utilities also provide a non-working backup. Before starting to work on this title, please make a backup, format a blank disk, and do a disk log (printout is best).

Working with your original:

l)   Make sure a write protect tab is on your original to protect it during the breaking process.

2)   Turn of f your computer and insert the reset assembly into the cartridge port. Turn your computer on again. From your utility disk, load the $2000 monitor <> LOAD "8192",8,l <>. When the load is complete, sys the monitor in with SYS 8192. When loading the boot file on this disk, it will autoboot and continue running. In order to inspect it, here's a trick to use. We're going to load the autoboot in BASIC memory for the purposes of inspection. With the monitor ~ctive, type X and hit RETURN. You are now back to BASIC. Type NEW and hit RETURN. Now load the boot file this way: <> LOAD "BL",8 <>. When the load is complete, return to the monitor by hitting RUNSTOP/RESTORE then re-SYS the monitor back in with SYS 8192. You can now find the boot file in BASIC memory at $0801. Interpret memory and scroll down from $0801 (I 0801). Notice the BOPl.

Disassembly of memory at $0801 (D 0801) and scrolling down reveals a loader file at $082D-$0854. This loader loads the BOPl file and jumps to $0816.

3)   Load the BOPl file <> L "BOPl",08 <>. When the load is complete, disassemble memory at $0816 (D 0816). Scroll down through memory to $0889. You'll find a JMP COOO. Using the MEMORY command (M0889), place a 00 (BRK) at $0889. If we start the code running from $0816 it will execute and stop just before it would have jumped to $COOO. We can then disassemble memory at $COOO and trace the program flow. Use the GO command to execute this code (G 0816).

4) The load will resume and the LOGO file and LOADALL file will be loaded. When the program stalls, reset out and reboot your monitor from the utility disk <> LOAD"8192",8,l <>. When the load is complete, sys the monitor in with SYS 8192. Disassemble code at $C000 (D C000) now and scroll down through memory. you'll find a very long loader file. When you reach the code at $C27A you'll find a JMP C3FD. Disassembly of C3FD shows no valid code so this is a likely spot to place another break in the program flow. Using the MEMORY command (N C27A), place a O0 (BRK) at $C27A. Now restart the program with another GO command (G C000). When the program stalls out, reset the computer again and reload and activate your 2000 monitor <> LOAD"8192",8,l <>. Now we can disassemble memory at $C3FD and again follow the program flow (D C3FD). This returns a JMP to 0B40. Disassembly of memory at $0B40 reveals the decryption code that we discussed in the introduction. This is the heart of this protection scheme.

5) Let's execute the code at $0B40. Make sure your original is in the drive. Start up the code with G 0B40. The drive should start up and soon stall again. Reset out, re-SYS your monitor in (SYS 49152), and disassemble code again starting at $0B40. You'll now find different code. Remove the original copy and place your formatted work disk in the drive. We can now save this new code to our work disk <> S "CODE",08,0B40,0C52 <>.

working with your backup:

6) We now have the code necessary to break this title. Now we have to place it on the disk in the proper spot. Checking the disk log, we find the files LOGO, BNKl2A, TITLE, and BOPl all have the correct addressing to be likely places for this file. We must load and check in each one with our monitor the address $0B40. The file BNKl2A turns out to be the correct file. Now all that is left is to place our changed code over the original code. Because BNKl28 begins in screen memory, we will have to pull a few tricks out of the bag to replace our revised code. Remember, this file starts in screen memory, and we can't save screen memory properly. Follow these steps and try to reason them out as we go through them.

A) Load DISK DR from your utility disk. When the cursor reappears, type RUN and hit RETURN. Place your backup in the drive and hit RETURN. You'll be shown track 18, sector 1. The jump link to the BNHl2A file is at position 195. Cursor over to position 195 and hit the J key. You will be taken to the first sector in the file. The first four bytes in the file are the pointer bytes. We want to change the program address from $0400 to $0900, so cursor over to position 3 and hit the @ key. Now, hit the 9 and press RETURN. Hit the R key to make the change to the backup.

B) Reset the computer and load the $COOO monitor from your utility disk <>LOAD"49152",8,l <>. Sys it in with SYS 49152. Now, from your formatted disk, load the CODE file <> L"CODE",08 <>. We now will transfer it to a holding spot in memory, for later use <> T 0B40 0C50 7B40. This will send the code to $7B40.

C) Now from the BACKUP load the altered file BNDl28 <> L"BNDl28",08 <>. Remember, it will now load five sectors ahead of it's normal spot (from $0400 to $0900). When the load is complete, disassemble the code at $1040. Again here is our decryption routine.

D) Transfer the code we placed at $7B40 to its proper place in the altered file <> T 7B40 7C50 1040 <>. When the cursor reappears, check the code at $1040. It should now contain the new code we saved from the break.

E) Save the altered file back to the backup <> S "@0:BNKl2A",08,0900,6101 <>. Note we are adding five sectors to every address, plus one byte to the end address.

F) Now all that's left is to change the file address back to $0400. Follow the same procedure as in step 6a, except change the address pointer from an 09 to an 04.

You now have a completely broken copy. The protection scheme has been totally wiped out.

PRINT SHOP COMPANION : BRODERBUND

Loading the original produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier provides a non-working backup. Nybble utilities also provide a non-working backup. Before starting to work on this title, please make a backup of both sides, and do a disk log (printout is best).

I must admit that this program was fairly difficult to trace through the loading sequence. After several tries, it was time to reason the situation out. Watching the backup load a few times lit up the old mental light bulb. The load seemed complete; the only problem were the ICONS on the first menu screen. They were there, but non-operative. Checking the directory provided the file I felt deserved immediate attention.

Working with the original:

1) Make sure to place write protect tabs on the original to protect it during the breaking process.

2) Turn the computer off and insert your reset assembly into the cartridge port. Turn the computer on again and from your utility disk, load the $COOO monitor <> LOAD"49152",8,l <>. Sys the monitor in with SYS 49152. Remove the utility disk from the drive and replace it with your original (Side A). Load the file ICONS <> L "ICONS",08 <>. The disk log tells us this file resides at $6000 in memory, so let's start our disassembly at $6000 (D 6000). Cursor down through memory and notice the decryption scheme at $6005-$6012. Remember from the introduction, this is the heart of this protection scheme.

3) Let's execute the code at the beginning of the decryption scheme. Start it working with G 6005. The drive should start up, and in a short time, stall again. Reset the computer and re-SYS the monitor back in with SYS 49152. Disassemble memory at $6000 again. Cursor down through memory and notice the code HAS changed. We now have all the data necessary in memory to break this program. Let's save our altered ICONS file back to the backup.

Working with your backup:

4) Checking the disk log shows that the ICONS file starts at $6000 and ends at $69AD in memory. With the backup in the drive, save the ICONS file, remembering to add one byte to the end address <> S "@0:ICONS",08,6000,69AE <>. Now turn the disk over and save the file to Side B as well.

You may now load and check your backup. You'll find it to be completely broken, and now it can even be fast copied. For those who want to see the actual protection check, you can go back through the same steps as before. When you do the G 6005, just reset out after about ONE second. If the drive is allowed to run, it will pick up the break data from the original, and hide the protection check in memory.

BANK STREET SPELLER : BRODERBUND

Procedure:

Loading the original produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast

Copier provides a non-working backup. Nybble utilities also provide a non-working backup. Before starting to work on this title please make a backup, and do a disk log (printout is best).

Working vith the original:

1) Make sure to place a write protect tab on the original to protect it during the breaking process.

2) Turn the computer of f and insert your reset assembly into the cartridge port. Turn the computer on again and from your utility disk, load the $COOO monitor <> LOAD"COOO",8,l <>. Sys the monitor in with SYS 49152. Remove the utility disk from the drive and replace it with your original. Load the boot file BSS <> L"BSS",08 <>. Using the disk log to guide us, let's disassemble memory at $02C4 (D 02C4). Cursor down through memory and notice the loader loads the file BSSL and does a jump to $7000. Let's load the BSSL file ourselves and follow the load sequence <> L "BSSL",08 <>.

3) When the drive stops, disassemble memory at $7000 (D 7000) Cursor down through memory, and inspect the long loader file that loads in the entire program and the jumps to the start address. At the address $20C7 you'll find a JNP 0803. Using the MEMORY command (M 70C7) type a 00 over the 4C and hit RETURN. This will allow the loader to operate, and, when done, will BREAK just before the jump to $0803. We can then follow the program flow, beginning at $0803. Start the loader execution by doing a GO 7000 (G 7000). The drive will start up and the files will appear on the screen as they are being loaded. When the drive finally stops, reset the computer and re-sys the monitor back in (SYS 49152).

4) Now let's disassemble memory at $0803 (D 0803). The first instruction we find is a JSR 09E1, so disassemble $09E1 (D 09E1). This disassembly reveals the decryption scheme that is hiding the protection check. You'll find it resides at $09E1 - $09F2. Study it closely, for it is the heart of this protection scheme.

5) Be sure your original disk is in the drive and start the code up by doing a GO 09E1 (G 09E1). The drive will start up and in a few seconds will stall again. Again, reset the computer and re-SYS the monitor in with SYS 49152. Disassemble memory at $09E1 (0 09E1) and inspect the code again. It has changed into valid program code. Now all that's left is to save the changed code back to the disk.

Working with the backup:

6) Inspection of the disk log tells us that the file BSSO is the likely candidate to contain the protection code. You may , as we did, load BSSO and inspect the proper addresses to ensure our save is to the proper file. Then, when satisfied, just redo step five and, when the code has been replaced again1 save the file back to your BACKUP disk. The disk log tells us the file resides for $0800-$1600. Be sure to add one byte to the end address
<> S "@O:BSSO",08,0800,1601 <>.

When this save is complete, your backup will be completely broken, and may be copied with any fast copier. For those who want to inspect the protection code, just load in the BSSO file and do a GO to 09E1. After about one second, reset out and re-SYS the monitor in and inspect that memory area. You'll find the protection code intact. If you allow the drive to run for long, the protection code will be replaced by valid program code.

EXPRESS RAIDER : DATAKAST

Procedure:

Loading the original disk produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy, A backup made with a nybbler produces the same non working backup. Before starting to work on this title, make a fast-copier backup and a formatted work disk. Because the only file on the directory of this title is the loader, special procedures will be required. You will need a reset button of some sort.

working with your backup:

1) With the reset switch in place, load the backup three or four times to get the feel of when the program stalls. When you have gotten the timing down, try to reset the computer just before that stall occurs. You will hear the head swing out if you are too late. We want to reset just before it does. After reset, from the Utility disk, load the $COOO monitor
< LOAD "49152",8,l > and after the load sys it in <SYS 49152>.

2) If you have performed previous breaks in Section E, you will remember that we are looking for a decrypter that hides the protection check. That decrypter ALWAYS begins with AO 00 A9. So we can search most of memory, flip out the BASIC Interpreter by changing memory location $0001 from a $37 to a $36 ($76 on the C-128) < M 0001 > . Now do a hunt for the key bytes in memory
< H 0800 BFFF AO 00 A9 > . If you have reset out at the proper time, the following addresses will be returned: 84C0 8759 9629

9A4C . Start by disassembling $84C0 < D 84C0> and inspecting the code below that address. If the code is clean, it is not what we are looking for. Inspect all the returned addresses and look for programming that has code beneath it that does not disassemble properly (usually you'll find a lot of ?????) . You'll find that $9626 fits the bill exactly. Here you'll find the decrypter with about a sector of encrypted code beneath it

3) Because of the no directory files problem, this break poses a slight inconvenience. We will have to search the disk for the proper place to lay down the break code. This type of loader uses a Track & Sector method of loading. You'll find that each page in memory occupies its own sector on the disk. Because the break code is between $9600 and $9800, we need to record the first 5 or 6 bytes from $9600 and $9700 to make it easier to locate these on the disk. (Remember these will be the first bytes in the sectors they occupy.) Using the Memory command, inspect and record the first few bytes in each: $9600 96 4C EO 97 4C FB 97/$9700= 40 ED 84 99 Ol 99 74. Again disassemble memory at the decrypter and use the cursor key to scroll down through memory < D 9626 > . You must scroll down at least a full sector ($9726) and a bit more, until you see clean code again. At $9736 you'll find a JUMP to $9744 (4C 44 97). Record this information for later reference.

Working with your original:

4) Power off and on again to clear memory. Load the original disk until the game has started up and again hit the reset button. From the Utility Disk, again load and activate the $C000 monitor as before. Start disassembly at $9626 < D 9626 > . You'll find new code has replaced the previous encrypted code. The key to breaking this type of protection is to replace the encrypted code with this new code. Disassemble again at $9626 and cursor down through memory. At $9736, you'll find the same three bytes as we recorded earlier: 4C 44 97. This tells us that the code from here on is the same as it was in the unrun and encrypted state. Place your formatted work disk in the drive and save the new code < S "BLOCK",08,9626,9738 >

Working with your backup:

5) Our task now is to transfer the code in the BLOCK file to the backup disk in the proper location. Here's the procedure. Power off and on again. Load the Disk Dr from the Utility Disk and RUN it < LOAD "DISK DOCTOR",8,l > . Using the - command from Disk Dr., search from Track 18/Sector 0 backwards one sector at a time. You'll be looking for the Sector that contains 96 4C EO 97 4C FB 97 as it's first seven bytes ($9600 in Memory) and 40 ED 84 99 Ol 99 74 as it's first seven bytes ($9700 in Memory). This

search is time consuming but necessary. You will find that $9600-$96FF will be at Track 11) Sector 8 and $9700-$97FF at Track 11/Sector 16. Thus the code must be placed at Track 11/Sector 8 Position $26 (38 in decimal) and continues on to Track 11/Sector 16 position $00 to end.

6) Using Hesmon, convert our BLOCK start and end addresses to decimal. $9626 = 38438 and $9736 = 38710. Power down and remove Hesmon. Now let's begin creating the parameter that will lay down the saved code in the proper location on the backup for us. Follow these instructions precisely.

A] From the work disk load the BLOCK file <LOAD "BLOCK",8,1>

B] Type NEW and hit RETURN.

C] From the Utility Disk load the PARM TEMPLATE
< LOAD "PARM TEMPLATE",8 >

D] List out the template and inspect. Start the data maker by typing GOTO6OO

E] Hit RETURN to continue. Enter Start as 38438 and END as 38710.

F] Record the number of bytes for use later (273 bytes) and hit RETURN.

C] The datamaker will now PEEK memory where our BLOCK is stored and convert the bytes to data statements in decimal.

H] When the program ends, LIST again. Edit line 5 for the desired title.

I] List out line 100 and Edit :TR=ll:SE=8:FB=38:NB=218
Tr=TRack(ll),SE=SEctor(8),FB=First Byte Position (38), NB=NUmber of bytes (218) <256-38=218>. Hit RETURN to lock in.

J] Type a 101 over the 100 in line 100 and Edit
Tr=ll : SE=16 : FB=00 NB=55
Tr=TRack(11) ,SE=SEctor(16) ,FB=First Byte Position (00), NB=Number of bytes (55) <273-218=55>.Hit RETURN to lock in.

K] Save the new parm to the work disk < SAVE "TEST",8 > .

7) Now run the parameter on the backup. Load the backup, and test it. You'll find that it doesn't work. Some titles require a little more work. Again with the reset switch in, load the original again, resetting just before the head swing. Again load the $COOO monitor and sys it in < SYS 49152 > . We need to find

the routine that either does a JSR or a JNP to the protection routine at $9626. Again change the $0001 address to $36 and use the HUNT command to search for a JSR 9626 or a JMP 9626 < H 0800 BFFF 20 26 96 >, and < H 0800 BFFF 4C 26 96 >. You should get a 76A1 returned. Disassembly of $76A1 shows a JSR 9626. Occasionally you will have to change the JSR (20) to a JMP (4C) or completely erase it with NOP's RA RA RA. As before, record the bytes at $7600 so we may find the sector containing this code on the disk. $7600 = 2F SD 11 DO BE 20 DO BE. Again power down and on again and load the Disk Dr from the Utility disk. Search the first bytes of each sector until you locate the desired pattern. We found it at Track 19/Sector 10. The 20 26 96 bytes are located at position $Al (72 in decimal).

8) Reload the TEST parameter for another change. List out line l00. Type a 99 over the 100 in line 100 and Edit Tr=19:SE=l0:FB=72:NB-03 (Tr=TRack(19),SE=SEctor(l0), FB=First Byte Position (72), NB=NUmber of bytes (03) <EA RA EA>). Hit RETURN to lock in. Finally add a new data statement. In a clear spot, TYPE : < 1900 DATA234,234,234 > and hit RETURN. Again, save the new parm to the work disk < SAVE "TEST 2",8 > . Run the parameter on the backup again. This time you'll find it works fine. This title although not file copyable is completely void of copy protection. Note: if you are confused as to how the parameter should look after you're done, list out the Express Raider parm from the Utility disk and list it out. It may become a little clearer to you.

BREAKTHROUGH : DATABAST

Procedure:

Loading the original disk produces a rattle free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working copy, A backup made with a nybbler produces the same non working backup. Before starting to work on this title, make a fast-copier backup and a formatted work disk. Because the only file on the directory of this title is the loader, special procedures will be required. You will need a reset button of some sort.

Working with your backup:

1) With the reset switch in place, load the backup three or four times to get the feel of when the program stalls. When you have gotten the timing down, try to reset the computer just before that stall occurs. You will hear the head swing out if you are too late. We want to reset just before it does. After reset,

from the Utility disk, load the $COOO monitor
< LOAD "49152",8,l > and after the load sys it in < SYS 49152 >.

2) If you have performed the previous breaks in Section E, you will remember that we are looking for a decrypter that hides the protection check. That decrypter'ALWAYS begins with AO 00 A9. So we can search most of memory, flip out the BASIC Interpreter by changing memory location $0001 from a $37 to a $36 ($76 on the C-128) < M 0001 > . Now do a hunt for the key bytes in memory < H 0800 BFFF AO 00 A9 > . If you have reset out at the proper time, the following addresses will be returned: 0F13 B4ED B9E8 Start by disassembling $0F13 < D 0F13> and inspecting the code below that address. If the code is clean, it is not what we are looking for. Inspect all the returned addresses and look for programming that has code beneath it that does not disassemble properly (usually you'll find a lot of ?) You'll find that $0F13 fits the bill exactly. Here you'll find the decrypter with about a sector of encrypted code beneath it.

3) Because of the no directory files problem, this break poses a slight inconvenience. We will have to search the disk for the proper place to lay down the break code. This type of loader uses a Track & Sector method of loading. You'll find that each page in memory occupies its own sector on the disk. Because the break code is between $OFOO and $1100, we need to record the first 5 or 6 bytes from $OFOO and $1000 to make it easier to locate these on the disk. (Remember these will be the first bytes in the sectors they occupy. Using the Memory command, inspect and record the first few bytes in each: $OFOO= 8D 5A OD A9 81 85 02/$l000= 00 00 00 00 00 00 00. Again disassemble memory at the decrypter and use the cursor key to scroll down through memory < D 0F13 > . You must scroll down at least a full sector ($1013) and a bit more, until you see clean code again. From $1013-$1041 you'll find all zero bytes. Record this information for later reference.

Working with your original:

4) Power off and on again to clear memory. Load the original disk until the game has started up and again hit the reset button. From the Utility Disk, again load and activate the $COOO monitor as before. Start disassembly at $0F13 < D 0F13 > . You'll find new code has replaced the previous encrypted code. The key to breaking this type of protection is to replace the encrypted code with this new code. Disassemble again at $0F13 and cursor down through memory. At $1013-$1041, you'll find the same zero bytes as we recorded earlier. This tells us that the code from here on is the same as it was in the unrun and encrypted state. Place your formatted work disk in the drive and save the new code < S "BLOCK",08,0F13,1014 >

Working with your backup:

5) Our task now is to transfer the code in the BLOCK file to the backup disk in the proper location. Here's the procedure. Power off and on again. Load the Disk Dr from the Utility Disk and RUN it < LOAD "DISK DOCTOR",8,1 > . Using the - command from Disk Dr, search from Track 18/Sector 0 backwards one sector at a time. You'll be looking for the Sector that contains 8D SA OD A9 81 85 02 as it's first seven bytes ($OFOO in Memory) and 00 00 00 00 00 00 00 as it's first seven bytes ($1000 in Memory). This search is time consuming but necessary. You will find that $OFOO-$OFFF will be at Track 17) Sector 19 and $1000-$lOFF at Track 17/Sector 6. Thus the code must be placed at Track 17/Sector 19 Position $13 (19 in decimal) and continues on to Track 17/Sector 6 position $00 to end.

6) Using Hesmon, convert our BLOCK start and end addresses to decimal. $0F13 3859 and $1013 4115. Power down and remove Hesmon. Now let's begin creating the parameter that will lay down the saved code in the proper location on the backup for us. Follow these instructions precisely.

A) From the work disk load the BLOCK file < LOAD "BLOCK",8,1 >

B) Type NEW and hit RETURN.

C) From the Utility Disk load the PARM TEMPLATE
< LOAD "PARM TEMPLATE",8 >

D) List out the template and inspect. Start the data maker by typing GOTO6OO

E) Hit RETURN to continue. Enter Start as 3859 and END as 4115.

F) Record the number of bytes for use later (257 bytes) and hit RETURN.

G) The datamaker will now PEEK memory where our BLOCK is stored and convert the bytes to data statements in decimal.

H) When the program ends, LIST again. Edit line 5 for the desired title.

I) List out line 100 and Edit :TR=17:SE=19:FB=19:NB=237 / Tr=TRack(17) ,SE=SEctor(19) ,FB=First Byte Position (19), NB=NUmber of bytes (237) <256-19=237>. Hit RETURN to lock in.

J) Type a 101 over the 100 in line 100 and Edit Tr--17 :SE=6 : FB=00 :NB=20 / Tr=TRack(17), SE--SEctor(6), FB=First

Byte Position (00), NB=NUmber of bytes (20) <257-237=20>.Hit RETURN to lock in.

K] Save the new parm to the work disk < SAVE "TEST",8 >

7) Run the parameter on the backup again. You'll find it works fine. This title although not file copyable is completely void of copy protection. Note: if you are confused as to how the parameter should look after you're done, list out the Breakthrough parm from the Utility disk and list it out. It may become a little clearer to you.

INTRO : PROTECTION SCREME TYPE F

This protection scheme although tough to copy, can usually be reproduced by a few of the modern nybblers such as The Shotgun. Because the protection is on one of the outer tracks (36-40), you must copy out to track 40. This scheme was developed in England and is seen on many of the Firebird releases. A few other publishers have used this scheme but those also had obvious English origins.

Characteristics of this scheme are references to GMA in the loader code or in the directory. Shortly after booting a non working copy, you can hear the head swing out and then the drive will lock up. Opening the drive door produces no flicker of the working drive light. Many times after a load failure, you will have to initialize your drive.

In short, this protection is executed at the beginning of the boot up process. It is generally accessed by a JSR to code that checks special code placed on an outer track, usually track 38. A numeric value is returned back only if the protection is in place. If a non working copy is being booted, the drive head will swing out and lock up. Some of these schemes use the ~umeric value brought back and some do not. We will examine three types. In all cases we will show you how to fool the code into not even doing the protection check. Also we will show you how to repair the often times corrupted directories.

Before working on these titles, please make a Fast Copy, and repair the directory. (See the general instructions below.) A disk log would also be helpful.

Using Disk Doctor and the map below, you should be able to repair most any directory. Let's begin with Track 18 sector 1. The first two bytes represent the link bytes. They will either point to the next track and sector or will indicate that this sector is the last one in the directory. Generally if more than one sector is used in the directory, you will find an rd at position 0. This represents a link to track 18 sector 4. A @ followed by a decimal 255 represents the last sector of the directory. If when starting at track 18 sector 0 you cannot use the n key and link the directory sectors together, you will have to repair these pointer bytes. After a little practice, this task will become easy. Now for the file entries. Most changes can be made in the text mode. Program type is rarely corrupted and a @ at that position indicates a scratched file. These are normal and should remain scratched.

The track and sector pointers must point to valid tracks and sectors or they are most likely dummy files meant to prevent file copying. Titles may have only upper and lower text in them. Those with text followed by other than a shifted space (decimal 160) should be filled with shifted spaces. Only occasionally will a program demand an unstandard file name. Finally, the number of sectors are not of major importance and will be normalized after file copying (when possible).

Track 18 Sector 0 represents the BAM and is often corrupted also. The main spots are position 2 which is the DOS flag byte. A byte other than an A will prevent you from writing to that disk. Change this byte if not normal using the text mode. Position 144 (decimal) represents the disk title and ID. These are in almost all cases, cosmetic and should be normalized. The title should be normal text and any unused title spaces should contain shifted spaces (decimal 160). The ID beginning at decimal position 162 can if desired, be 5 characters. These must however be normal text characters.

Maps of normal sectors have been given. Use these maps and Disk Doctor to examine our Utility Disk. When you understand the normal format, the abnormal will become easy to fix.

Track 18/Sector 0

	Title	Sh/Spaces	ID	Sh/Space	2A	Sh/Spaces
Pos:	144-159	160-161	162-163	164	165-166	167-170

	Track	18/Sector	1-18	
Program Type	Track	Sector	Title	# of Sectors
2	* 3	* 4	* 5-20	* 30
34	* 35	* 36	* 37-52	* 62
66	* 67	* 68	* 69-84	* 94
98	* 99	* 100	* 101-116	* 126
130	* 131	* 132	* 133-148	* 158
162	* 163	* 164	* 165-180	* 190
194	* 195	* 196	* 197-212	* 222
226	* 227	* 228	* 229-244	* 254

ARTIST 64 : WIGMORE

Procedure:

Loading the original disk reveals the GMA symbol on the opening loader screen. A fast copy when booted, locks up the drive and sends it into an endless spin. Before starting, make a fast copy using our C-64 Fast Copy. Repair the directory according to the step one instructions and then validate the disk. Finally, a disk log may be helpful.

Working with your backup:

1) Load Disk Doctor from the Utility Disk, and inspect Track 18/Sector 1. You should find this sector to be normal. Use the - key to go to Track 18 } Sector 0. You'll find the NAME and ID number to be corrupted. One way to repair it is as follows: Cursor to position 144 and type <t> for text mode. In this mode type ARTIST 64 followed by shifted spaces (decimal 160) to position 162. Now type AR/64 and hit RETURN. Write these changes to the backup by typing <r> followed by a <y>. Lastly while at this sector hit <n> to go to the next sector in the directory.

You'll find that it goes to Track 18/Sector 4. At Track 18/Sector 4 you'll find no directory entries. The correct path is to Track 18/Sector 1 so go back to Track 18/Sector 0 and change the first two bytes from rd to ra or 18 1 in decimal. Use the @ key to make each change and be sure to rewrite your changes to the backup. Now power down and load and check the directory. The file names should be present. Validate the disk and then using the disk logger, log the file addresses.

2) With Hesmon in the cartridge port, load the boot file < L"B" 08 >. At the end of the load, Disassemble code at $02A7 <D 02A7> and using the cursor down key, scroll down through memory. The code highlights are:

A] D 02C6 : JSR FF90 (control load messages)
B] D O2CF : JSR FFBA (set logical addresses)
C] D 02D8 : JSR FFBD (set file name:3 characters located at $02C1:Use I command to see [I 02C1] the file name GM1.
D] D 02F1 : JSR FFD5 (load into ram)
E] D O2FD : JMP COOO (Jump to location $COOO.)

3) We now know that the next file loaded in is GM1 and that the code at $COOO is the jump link. Load the GM1 file as < L"GM1" 08 >. Start disassembly of code at $COOO.

A) Let's execute the code at $COOO and see what happens. Type [G COOO]. Notice the beginning screen comes up and asks for y or n for fast loader. Type n and listen. A short load takes place and the head swings out. The drive will be locked up. Power down and up again, type X to return to basic and initialize your drive. When the drive stops, hit RUNSTOP/RESTORE to return to the monitor.

B] Again load the GM1 file as before and start Disassembling code at $C0000 < D COOO >. Cursor down through the code to $C024. Here you 'll find a JSR C800. This is the actual protection check routine. Notice the next instruction is a PHA which places the numeric value returned from the protection check on the stack. This value is the key to this protection scheme.

C] Make sure you place a write protect on the ORIGINAL Artist 64 and place it into the drive. Using the Memory Command, change the PHA(48) at $C027 to a BRK(00). <M C027>. We can now execute the protection code from the original and the value in the left in the A register when the code breaks will be the numeric value we're looking for. Execute the code by typing < G COOO >.

D] The opening screen will again appear and input N again and the load will continue. This time the head will swing out and a few moments later the program will break. The registers will be on the screen. Note the A register has a value of 24. This is the value we're looking for. (Those who want to inspect the drive routine that checks protection may find it starting at $C800.)

E] The break is now quite simple. We can replace the JSR C800 instruction with the value and totally skip the protection check. By replacing it with A9 24 EA (LDA 24 NOP) we can directly load the accumulator with a 24 which then will be pushed onto the stack. Let's make our changes with Disk Doctor.

F] Using the converter in Hesmon, find the decimal equivalent to A9 24 EA. In a clear work space type <$ 00A9>. The decimal value 169 will be returned. The same procedure for 0024 and OOEA will return 36 and 234 respectively. Power down and remove Hesmon. From the Utility disk, load Disk Doctor and again insert the backup into the drive. At Track 18/Sector 1, position 34, you'll find the Prg byte for the GM1 file. Place the cursor on the Track pointer at position 35 and press j to Jump to Link. You'll be taken to Track 17, Sector 1. Starting at position 0 cursor along and look for the hex bytes 20 00 C8 (JSR C800) pattern. At position 40 you'll find the first byte of that pattern. Use the @ key to change three bytes starting at position 40 to 169, 36, 234 (decimal equivalent). Hit the <r> key to rewrite the sector and then < y > for yes. Your title is now free from all protection and may even be file copied if desired.

COLOSSUS CHESS : FIREBIRD

Procedure.

Loading the original disk reveals the GMA symbol on the opening loader screen. A fast copy when booted, locks up the drive and sends it into an endless spin. Before starting, make a fast copy using our C-64 Fast Copy. Repair the directory according to the step one instructions. Be sure to validate the disk and do a log the disk as a part of your preparation.

Working with your backup:

1) Load Disk Doctor from the Utility Disk, and inspect Track 18/Sector 1. You should find this sector to be normal. Use the - key to go to Track 18 Sector 0. You'll find the NAME and ID

number to be corrupted. One way to repair it is as follows: Cursor to position 144 and type <t> for text mode. In this mode type COLOSSUS followed by shifted spaces (decimal 160) to position 162. Now type CHESS and hit RETURN. Write these changes to the backup by typing <r> followed by a <y>. Lastly while at this sector hit <n> to go to the next sector in the directory. You'll find that it goes to Track 18/Sector 1. Continue hitting n to go to each linked sector in the directory. You'll find every sector to be normal with the last directory sector at Track 18) Sector 5. Now power down and load and check the directory. The file names should be present. Validate the disk and then using the disk logger, log the file addresses.

2) With Hesmon in the cartridge port, load the boot file < "Firebird",08 >. At the end of the load, Disassemble code at $02A7 <D 02A7> and using the cursor down key, scroll down through memory. The code highlights are:

A] D 02C6 : JSR FF90 (control load messages)

B] D O2CF : JSR FFBA (set logical addresses)
C] D 02D8 : JSR FFBD (set file name:3 characters located at $02C1:Use Interpret command to see < I 02C1 > the file name GMl.

D] D 02F1 : JSR FFD5 (load into ram)

E] D O2FD : JMP 0334 (Jump to location $0334.)

3) We now know that the next file loaded in is GMl and that the code at $0334 is the jump link. Load the GM1 file as < L "GMl" 08 >. Notice that the code fills the screen. This is because it is loaded into screen memory. Cursor down and start disassembly of code at $0334.

A] Disassemble code at $0334 <D 0334>. You'll find a jump to $034B. Cursor down and inspect the code from $034B-$036B. This code represents the key to the protection. This particular code can be found in many similar titles and the break for all is about the same. This code sets up a load of the actual protection check code within the GMA3 file. (Those of you interested in the drive code for the protection should load and inspect GMA3.) A JSR to $C800 within this code checks protection, and if the check is successful, a value of $97 is place at computer location $0002. Upon return from the JSR C800, the value in location $0002 is loaded into the accumulator and EORed with a value of $97. Lastly the code Branches if Equal (to 0) to $036C. Remember, if protection WAS satisfied, a value of $97 was placed at $0002. The EOR Truth Table in the back of the book tells us that $97 EORed with

$97 is in fact zero. If the branch does take place, it will cause a jump around the instruction at $0369 which is a JMP ($FFFC). This instruction is actually a Jump to a Kernal routine that does a system reset, which in turn will crash the load process.

B] The break is now quite simple. We can jump around the whole protection check. All that is necessary is to replace the JSR C800 with a JUMP around the reset code to $036C. We will replace the 20 00 C8 with 4C 6C 03 (JMP 036C). Remember, we don't want to allow any protection check because if the protection is not in place, the drive hangs up and goes into an endless spin. Let's make our changes with Disk Doctor.

C] Using the converter in Hesmon, find the decimal equivalent to 4C 6C 03. In a clear work space type <$ 004C>. The decimal value 76 will be returned. The same procedure for 006C and 0003 will return 108 and 03 respectively. Power down and remove Hesmon. From the Utility disk, load Disk Doctor and again insert the backup into the drive. At Track 18/Sector 1, position 34, you'll find the Prq byte for the GM1 file. Place the cursor on the Track pointer at position 35 and press <j> to Jump to Link. You'll be taken to Track 17, Sector 1. Starting at position 0 cursor along and look for the hex bytes 20 00 C8 (JSR C800) pattern. At position 48 you'll find the first byte of that pattern. Use the @ key to change three bytes starting at position 48 to 76, 108, 03 (decimal equivalent). Hit the <r> key to rewrite the sector and then <y> for yes. Your title is now free from all protection and may even be file copied if desired.

COMPUTER SCRABBLE : LEISURE GENIUS

Procedure:

Loading the directory of the original disk reveals the GMA symbol. A fast copy when booted, locks up the drive and sends it into an endless spin. Before starting, make a fast copy using our C-64 Fast Copy. Repair the directory according to the step one instructions. Be sure to validate the disk and do a log the disk as a part of your preparation.

Working with your backup:

l) Load Disk Doctor from the Utility Disk, and inspect Track 18/Sector 1. You should find this sector to be normal. Use the - key to go to Track 18) Sector 0. You'll find the NAME and ID number to be corrupted. One way to repair it is as follows: Cursor to position 144 and type <t> for text Rode. In this mode type SCRABBLE followed by shifted spaces (decimal 160) to position 162. Now type LaG/SC and hit RETURN. Write these changes to the backup by typing <r> followed by a <y>. Lastly while at this sector hit <n> to go to the next sector in the directory. You'll find that it goes to Track 18/Sector 1. Continue hitting n to go to each linked sector in the directory. You'll find every sector to be normal with the last directory sector at Track 18) Sector 4. Now power down and load and check the directory. The file names should be present. Validate the disk and then using the disk logger, log the file addresses.

2) With Hesmon in the cartridge port, load the boot file < L"B" 08 >. At the end of the load, Disassemble code at $02A7 < D 02A7 > and using the cursor down key, scroll down through memory. The code highlights are:

```
 A]   D 02C6 : JSR FF90 (control load messages)

 B]   D O2CF : JSR FFBA (set logical addresses)

 C]   D 02D8 : JSR FFBD (set file name:3 characters located at
        $02C1: Use Interpret command to see< I 02C1 >the file
               name GMl.

 D]   D 02F1 : JSR FFD5 (load into ram)

 E]   D O2FD : JMP 3800 (Jump to location $3800.)
```

3) We now know that the next file loaded in is GMl and that the code at $3800 is the jump~link. Load the GMl file as < L"GMl" 08 >. Start disassembly of code at $3800.

A] Let's execute the code at $3800 and see what happens. Type < G 3800 >. Notice the beginning screen comes up and asks for y or n for fast loader. Type n and listen. A short load takes place and the head swings out. The drive will be locked up. Power down and up again, type X to return to basic and initialize your drive. When the drive stops, hit RUNSTOP/RESTORE to return to the monitor.

B] Again load the GMl file as before and start Disassembling code at $3800 <D 3800>. Cursor down through the code to $384A. Here you'll find a JSR C800. This is the actual protection check routine. Notice the next instruction is a PHA which places the numeric value returned from the protection check on the stack. This value is the key to this protection scheme.

C] Make sure you place a write protect on the ORIGINAL Artist 64 and place it into the drive. Using the Memory Command, change the PHA(48) at $384D to a BRK(00). < M 384D >. We can now execute the protection code from the original and the value in the left in the A register when the code breaks will be the numeric value we're looking for. Execute the code by typing <G 3800>.

D] The opening screen will again appear and input N again and the load will continue. This time the head will swing out and a few moments later the program will break. The registers will be on the screen. Note the A register has a value of 58. This is the value we're looking for. (Those who want to inspect the drive routine that checks protection may find it starting at $C800.)

E] The break is now quite simple. We can replace the JSR C800 instruction with the value and totally skip the protection check. By replacing it with A9 58 EA (LDA 58 NOP) we can directly load the accumulator with a 58 which then will be pushed onto the stack. Let's make our changes with Disk Doctor.

F] Using the converter in Hesmon, find the decimal equivalent to A9 58 EA. In a clear work space type <$ 00A9>. The decimal value 169 will be returned. The same procedure for 0058 and OOEA will return 88 and 234 respectively. Power down and remove Hesmon. From the Utility Disk, load Disk Doctor and again insert the backup into the drive. At Track 18/Sector 1, position 34, you'll find the Prg byte for the GMl file. Place the cursor on the Track pointer at position 35 and press <j> to Jump to Link. You'll be taken to Track 17, Sector 1. Starting at position 0 cursor along and look for the hex bytes 20 00 C8 (JSR C800) pattern. At position 78 you'll find the first byte of that pattern. Use the ~ key to change three bytes starting at position 78 to 169, 88, 234 (decimal equivalent). Hit the <r> key to rewrite the sector and then <y> for yes. Your title is now free from all protection and may even be file copied if desired.

FAULKLANDS 82 : FIREBIRD

Procedure:

Loading the original disk reveals the GMA symbol on the opening loader screen. A fast copy when booted, locks up the drive and sends it into an endless spin. Before starting, make a fast copy using our C-64 Fast Copy. Repair the directory according to the step one instructions. Be sure to validate the disk and do a log the disk as a part of your preparation.

working with your backup:

1) Load Disk Doctor from the Utility Disk, and inspect Track 18/Sector 1. You should find this sector to be normal. Use the - key to go to Track 18) Sector 0. You'll find the NAME and ID number to be corrupted. One way to repair it is as follows: Cursor to position 144 and type <t> for text mode. In this mode type FALKLANDS followed by shifted spaces (decimal 160) to position 162. Now type FL/82 and hit RETURN. Write these changes to the backup by typing <r> followed by a <y>. Now take a look at position 2. Anything other than a capital A in that spot will prevent you from writing to the disk. You must cursor up to position 2 and hit <t> for text mode then type A to that position. Hit <r> for rewrite and <y> for yes. Lastly while at this sector hit <n> to go to the next sector in the directory. You'll find that it goes to Track 18/Sector 1. Continue hitting n to go to each linked sector in the directory. You'll find every sector to be normal with the last directory sector at Track 18) Sector 4. Now power down and load and check the directory. The file names should be present. Validate the disk and then using the disk logger, log the file addresses.

2) With Hesmon in the cartridge port, load the boot file < L"Firebird" 08 >. At the end of the load, Disassemble code at $02A7 <D 02A7> and using the cursor down key, scroll down through memory. The code highlights are:

   A] D 02C6 : JSR FF90 (control load messages)

   B] D O2CF : JSR FFBA (set logical addresses)

   C] D 02D8 : JSR FFBD (set file name:3 characters located at $02C1: Use Interpret command to see< I 02C1 >the file name GMl.

   D] D 02F1 : JSR FFD5 (load into ram)

   E] D O2FD : JMP COOO (Jump to location $COOO.)

3) We now know that the next file loaded in is GMl and that the code at $COOO is the jump link. Load the GMl file as < L"GMl" 08 >. Cursor down and start disassembly of code at $COOO.

 A] Disassemble code at $C000 < D COOO >. You'll find a Jump to $COOF. Cursor down and inspect the code from $COOF-$C029. This code represents the key to the protection. This particular code can be found in many similar titles and the break for all is about the same. This code sets up a load of the actual protection check code within the GMA3 file. (Those of you interested in the drive code for the protection should load and inspect GMA3.) A JSR to $C800 within this code checks protection, and if the check is successful, a value of $97 is place at computer location $0002. Upon return from the JSR C800, the value in location $0002 is loaded into the accumulator and EORed with a value of $97. Lastly the code Branches if Equal(to 0) to $CO2A. Remember, if protection WAS satisfied, a value of $97 was placed at $0002. The EOR Truth Table in the back of the book tells us that $97 EORed with $97 is in fact zero. If the branch does take place, it will cause a jump around the instruction at $C027 which is a JMP ($FFFC). This instruction is actually a Jump to a Kernal routine that does a system reset, which in turn will crash the load process.

B] The break is now quite simple. We can jump around the whole protection check. All that is necessary is to replace the JSR C800 with a JUMP around the reset code to $CO2A. We will replace the 20 00 C8 with 4C 2A Co (JMP CO2A). Remember, we don't want to allow any protection check because if the protection is not in place, the drive hangs up and goes into an endless spin. Let's make our changes with Disk Doctor.

C] Using the converter in Hesmon, find the decimal equivalent to 4C 2A CO. In a clear work space type <$ 004C>. The decimal value 76 will be returned. The same procedure for 002A and OOCO will return 42 and 192 respectively. Power down and remove Hesmon. From the Utility Disk, load Disk Doctor and again insert the backup into the drive. At Track 18/Sector 1, position 34, you'll find the Prg byte for the GMl file. Place the cursor on the Track pointer at position 35 and press j to Jump to Link. You'll be taken to Track 17, Sector 1. Starting at position 0 cursor along and look for the hex bytes 20 00 C8 (JSR C800) pattern. At position 34 you'll find the first byte of that pattern. Use the @ key to change three bytes starting at position 34 to 76, 42, 192 (decimal equivalent). Hit the r key to rewrite the sector and then y for yes. Your title is now free from all protection and may even be file copied if desired.

INTRO : PROTECTION SCHEME TYPE G

Most computer software houses utilize some form of "copy protection" that prevents the average consumer from making backup copies of the program(s) that the company distributes. Even the most basic Commodore user is aware that protection is included on most of the commercial programs he buys. Using a simple data-copier to archive the original usually fails to make a working copy.

One company on the other hand, uses a different approach for their latest series of sports games. Instead of encoding the protection upon the diskette where the game is stored, included with the sale of each of their programs is a device called a Ndongleel The dongle is simply a small plastic device that plugs into the cassette port of your Commodore 64/l28~. The dongle includes a small resistor that makes it look complicated, but it is actually a very simple device. The resistor merely ties a positive 6 volt lead to an input port that the Commodore uses for cassette load/save interfacing. The fact is, the resistor on the dongle could be replaced with a simple piece of wire. The resistor serves merely either to avoid "shorting" out your Commodore (which is doubtful), or, as most of us tend to see it, as a deceiving device.

Through software, the programmer checks a certain memory location to see if that particular bit has a 0 value (dongle in place), or a 1 value (dongle not plugged in.) If the bit value retrieved is a "1", the program refuses to operate.

The following tutorials will deal with deprotecting the software checks in the program code. Looking through machine-language code for a protection-check is quite a time-consuming task since there are probably a million ways to check if a bit value at a certain memory location is either on or of f. In the following pages, we will try to give you some of the more popular methods.

The bit that the dongle triggers is located at memory location $0001. Using a machine-language monitor, we can verify that bit 4 is always on without the dongle plugged in.

```
 $0001:

 Bit 7 6 5 4 3 2 1 0

     X X X l X X X X
```

Bit 4 will become "0" when the dongle is plugged in. A short machine-code program assembled in the cassette buffer ($0334) can check the 4th bit:

```
        A 0334 LDA #$l0
          0336 BIT $01
          0338 BEQ $033A
          0339 BRK
          033A BRK
```

Type G 0334 with the dongle in or out.

The BIT instruction "AND's" memory location $01 with the value in the accumulator (#$l0 = check bit 4). If the dongle is plugged in, both bits will match up (both l's), and the branch instruction will be bypassed and the program will break into the monitor at $0339.

Running the program again with the dongle plugged in will AND a 1 bit with the dongle 0 bit, causing the branch to be executed. The program will break into the monitor at $033A. This is just one method in which ACCESS checks their protection. We can "break" their protection checks by replacing LDA #$l0 with LDA #$00. This way, the BIT instruction will always result in setting the zero flag, which emulates the dongle!

Here are some other code forms for checking the dongle:

```
 LDA #$l0
 BIT $00  (memory location zero, bit 4 holds an image of $0001)
 BEQ dongle in
```

Solution: replace LDA #$l0 with LDA #$00.

```
 LDA $01
 AND #$l0
 BEQ dongle in
```

Solution: replace AND #$l0 with AND #$00.

```
 LDA #$40
 LSR
 LSR
 TAX
 AND $FFFl,X
 BEQ dongle in
```

Solution: replace LDA #$40 with LDA #$00.

```
 LDA $0001
 ASL
 TAX
 ASL
 ASL
 ASL
 BCS dongle out
```

Solution: replace BCS with two "NOP"'s.

There are many other ways to check memory location $0001 for the dongle bit. In the following pages you will find instructions on how to disable the checks in four programs. These should give you the insight necessary to continue on your own.

LEADERBOARD : ACCESS

Use the C-64 Fast Copier utility to make an exact data-copy of the original. This backup will run like the original ONLY if the dongle is in place. The following procedure will eliminate all dongle-checks:

Working with your backup:

1) Turn on your computer and from the Utility Disk, load the Disk Logger by typing < LOAD "DISK LOGGER",8 > . Then type RUN. Insert your backup copy of Leaderboard in the drive and log it. The two files on the disk that contain code that check for the dongle are called "L" and "H". Take note of the addresses in memory where these programs reside:

```
 "L"  $081D - $3E32
 "H"  $9280 - $AB9A
```

This information is important since we need to load a machine-language monitor into memory where these programs aren't! We can choose from one of three monitors ($2000 = 8192, $8000 = 32768, or $COOO = 49152). The monitor at $COOO does not conflict with Leaderboard memory, so let's use it.

2) Turn on the computer again and load the $COOO monitor from your utility disk < LOAD "49152",8,l > followed by < SYS 49152 > to execute it.

3) To start with a clean slate, let's clear out all memory below the monitor by typing < F 0800 BFFF EA >

4) From the monitor, we must load the two Leaderboard files. Insert your backup copy in the drive and load both files: < L "H",08 > and < L "L",08 >

5) Since the "H" file resides in the RAM underneath the BASIC ROMS ($AOOO-$BFFF), we have to use the bank select bits to bank out the ROM and bank in the RAM so we can view the "H" file code. Using the memory command, change location $0001 to 36 (76 on the 128) < M 0001 >.

6) Now we will began searching for the certain "dongle-check" byte

sequences. We can use the monitor "H" command to hunt through memory for these patterns. Type < H 0800 BFFF A9 10 24 Ol > After a brief wait, the monitor should return addresses: 0AA2 112F AO3C.

7) Disassemble each of these addresses using the < D > command. Use the cursor-down key to scroll through the next couple of addresses. At the top after each assembly, change the LDA #$l0 command to: LDA #$00 (see intro). i.e. - < D 0AA2 >, < A 0AA2 LDA #$00 >... do the same for the other two addresses. The rest of the byte changes are performed in this manner, so they won't be in detail.

8) Type < H 0800 BFFF A9 40 4A 4A AA > Monitor finds: 1245 9D20.

9) Disassemble both addresses, and change the LDA #$40 command to LDA #$00 (see intro).

10) Type < H 0800 BFFF AD Ol 00 > Monitor finds: 9AEO.

11) Disassemble $9AEO and cursor down 10 or 11 times. Find the BCS instruction and replace it with two NOPs (see intro).
< A 9AE8 NOP > < A 9AE9 NOP >

12) Type < H 0800 BFFF 58 FF > . Monitor finds: 14D1 A6F4.

13) First, disassemble a few bytes before $14D1, say at $14C0. You will discover a routine that looks something like the following:

```
LDX #$09
LDA $14D8,X
EOR #$FF
STA $FF58,X
```

Notice that this routine decrypts a sequence of bytes beginning at $14D8 by EOR'ing it with the value of #$FF and stores it in hi-memory hidden beneath the Kernal ROMs. The routine itself breaks into the IRQ routine and checks the dongle bit every time the IRQ routine pointed to by vector $0314-$0315 is executed. To see the decrypted code, you will have to point the routine to a location in RAM that is easily visible, say $0801 (FF58   0801). If you do, be sure to start the break procedure over, for you will have corrupted our work up to now.

14) To "trick" the routine into thinking that the dongle is always in, type < M 14D8 > . The monitor should return a sequence of 8 bytes.

15) Edit the 4th byte over (should be $EF) and change it to $FF.

16) Next, disassemble memory a few bytes before $A6F4 by typing < D A6FO > . Use cursor/down to display the next 14 or 15 bytes. The monitor should show you something like:

```
LDX #$09
CLC
ADC $FF58,X
DEX
```

17) This group of instructions is simply a checksum check of the IRQ dongle-check routine we just finished working with. In other words, they are "double-checking" their protection code. Find the instruction that compares the checksum value in the accumulator with a set value. Notice the 'BEQ' immediately afterwards that bypasses protection failure. Simply change 1CMP #$5A' with 'LDA #$00'. We have just set the zero flag permanently, and the routine is tricked".

18) Now that we have finished removing all the dongle-check routines, we need to re-save the two files to your backup disk. Type:< S"@0:L",08,081D,3E33 > < S"@0:H",08,9280,AB9B >

19) You now have a dongle-free backup of Leaderboard. It may be archived using any simple data copier. Note: The parameter LEADERB. PARM 1 represents this particular break method. LEADERB. PARM 2 is a variation of this break and can be run on a backup and examined with the monitor.

EXECUTIVE LEADERBOARD : ACCESS

Use the C-64 Fast Copier utility to make an exact data-copy of the original. This backup will run like the original ONLY if the dongle is in place. The following procedure will eliminate all dongle-checks:

working with your backup:

1) Turn on your computer and from the Utility Disk, load the Disk Logger by typing < LOAD "DISK LOGGERN,8 > . Then type RUN. Insert your backup copy of Executive Leaderboard #1 in the drive and log it. The two files on the disk that contain code that check for the dongle are called "L" and "H". Take note of the addresses in memory where these programs reside:

"L" $081D - $3FAF

"H" $9280 - $BAEC

This information is important since we need to load a machine-language monitor into memory where these programs aren't! We can choose from one of three monitors ($2000 = 8192, $8000 = 32768, or $COOO = 49152). The monitor at $COOO does not conflict with Exec Leaderboard #1 memory, so we will use it.

2) Turn on the computer again and load the $COOO monitor from your utility disk < LOAD "49152",8,l > followed by < SYS 49152 > to execute it.

3) To start with a clean-slate, let's clear out all memory below the monitor by typing < F 0800 BFFF EA >

4) From the monitor, we must load the two Exec Leaderboard #1 files. Insert your backup copy in the drive and load both files: < L "H",08 > and < L "L",08 >

5) Since the "H" files resides in the RAM underneath the BASIC ROMS ($AOOO-BFFF), we have to use the bank select bits to bank out the ROM and bank in the RAM so we can view the "H" file code. Using the memory command, change memory location $0001 to 36 (76 on the 128) < M 0001 >

6) Now, we will began searching for the certain "dongle-check" byte sequences. We can use the monitor "H" command to hunt through memory for these patterns. Type < H 0800 BFFF A9 10 24 Ol > After a brief wait, the monitor should return addresses: OA9C 1114 9FA2.

7) Disassemble each of these addresses using the "D" command. Use the cursor-down key to scroll through the next couple of addresses. At the top after each assembly, change the LDA #$l0 command to: LDA #$00 . i.e. - < D OA9C >   < A OA9C LDA #$00 > do the same for the other two addresses. The rest of the byte changes are performed in this manner, so they won't be in detail!

8) Type < H 0800 BFFF A9 40 4A 4A AA > . Monitor finds: 1237 9D3E.

9) Disassemble both addresses, and change the LDA #$40 command to LDA #$00

10) Type < H 0800 BFFF A9 10 24 00 > . Monitor finds: 93EF.

11) Disassemble and change LDA #$l0 to LDA #$00.

12) Type < H 0800 BFFF AD OE C2 OA AA > . Monitor finds: 9AFE.

13) Disassemble $9AFE and scroll down 6 or 7 times. Find the BCS

instruction and replace it with two NOPs. < A 9B06 NOP > , < A 9B07 NOP > .

14) Type < H 0800 BFFF 58 FF > . Monitor finds: 14AC A5E7.

15) First, disassemble a few bytes before $14AC, say at $14A3. You will discover a routine that looks something like the following

```
LDX #$09
LDA $14B3,X
EOR #$FF
STA $FF58,X
```

Notice that this routine decrypts a sequence of bytes beginning at $14B3 by EOR'ing it with the value of #$FF and stores it in hi-memory hidden beneath the Kernal ROMs. The routine itself breaks into the IRQ routine and checks the dongle bit every time the IRQ routine pointed to by vector $0314-0315 is executed. To see the decrypted code, you will have to point the routine to a location in RAM that is easily visible, say $0801 (FF58 = 0801). If you do, be sure to start the break procedure over, for you will have corrupted our work up till now.

16) To "trick" the routine into thinking that the dongle is always in, type < M 14B3 > . The monitor should return a sequence of 8 bytes.

17) Edit the 4th byte over (should be $EF) and change it to $FF.

18) Next, disassemble memory a few bytes before $A5E7 by typing < D A5E1 > . Use cursor-down to display the next 14 or 15 bytes.

The monitor should show you something like:

```
LDX #$09
CLC
ADC $FF58,X
DEX
```

19) This group of instructions is simply a checksum check of the IRQ dongle-check routine we just finished working with. In other words, they are "double-checking" their protection code. Find the instruction that compares the checksum value in the accumulator with a set value. Notice the BEQ immediately afterwards that bypasses protection failure. Simply change CMP #$5A with LDA #$00 . We have just set the zero flag permanently, and the routine is tricked.

20) Now that we have finished removing all the dongle-check routines, we need to re-save the two files to your backup disk.

Type: < S"@0:L",08,081D,3FB0 > < S"@0:H",08,9280,BAED >

21) The exact same procedure described above must be repeated for two files "L5" and "H5", which are identical other than name to "L" and "H". So repeat steps 3-20 but use "L5" and "H5" as filenames instead!

22) After this is done, you will have a dongle-free backup of Executive Leaderboard #1. It may be archived using any simple data copier. Note: The parameter for LB Exec #1 on the utility disk represents a variation of this break and can be run on a backup and examined with the monitor. You'll find all changes in about the same memory locations.

LEADERBOARD TOURNAMENT DISK : ACCESS

Procedure:

Use the C-64 Fast Copier to make an exact data-copy of the original. This backup will run like the original ONLY if the dongle is in place. The following procedure will eliminate all dongle-checks:

Working with your backup:

1) Turn on your computer and from the Utility Disk, load the Disk Logger utility by typing < LOAD "DISK LOGGER",8 > . Then type < RUN >. Insert your backup copy of Leaderboard Tournament in the drive and log it. The file on the disk that contains the code that checks for the dongle is called "B". Take note of the addresses in memory where this program resides:
"B" $9280 - $BF53 . This information is important since we need to load a machine-language monitor into memory where this program isn't! We can choose from one of three monitors ($2000 = 8192, $8000 = 32768, or $C000 = 49152). The monitor at $C000 does not conflict with Leaderboard memory, so we will use it.

2) Turn on the computer again and load the $C000 monitor from your utility disk < LOAD "49152",8,1 > followed by < SYS 49152 > to execute it.

3) To start with a clean-slate, let's clear out all memory below the monitor by typing < F 0800 BFFF EA > .

4) From the monitor, we must load the Leaderboard file. Insert your backup copy in the drive and load the file: < L"B",08>.

5) Since the "B" file resides in the RAM underneath the BASIC ROMS ($A000-BFFF), we have to use the bank select bits to bank out

the ROM and bank in the RAM so we can view the "B" file code. Using the memory command, change memory location $0001 to 36 (76 on the 128) < M 0001 > .

6) Now, we will began searching for the certain "dongle-check" byte sequences. We can use the monitor "H" command to hunt through memory for these patterns. Type
< H 9000 BFFF A9 10 24 01 >. After a brief wait, the monitor should return address: A03C

7) Disassemble this address using the "D" command. Use the cursor-down key to scroll through the next couple of addresses. At the top, change the LDA #$10 command to: LDA #$00 . i.e. - < D A03C > , < A A03C LDA #$00 >. The rest of the byte changes are performed in this manner, so they won't be in detail!

8) Type < H 9000 BFFF A9 40 4A 4A AA > . Monitor finds: 9D20.

9) Disassemble and change the LDA #$40 command to LDA #$00 .

10) Type < H 9000 BFFF A9 10 24 00 > . Monitor finds: 93EF.

11) Disassemble and change LDA #$10 to LDA #$00 .

12) Type < H 9000 BFFF AD 01 00 0A AA > . Monitor finds: 9AE0.

13) Disassemble $9AE0 and scroll down 6 or 7 times. Find the BCS instruction and replace it with two NOP's. < A 9AE8 NOP >,
< A 9AE9 NOP > .

14) Type < H 9000 BFFF 58 FF > . Monitor finds: A6F4.

15) Disassemble memory a few bytes before $A6F4 by typing
< D A6E0 >. Use cursor-down to display the next 14 or 15 bytes. The monitor should show you something like:

```
LDX #$09
CLC
ADC $FF58,X
DEX
```

16) This group of instructions is simply a checksum check of the IRQ dongle-check routine we worked with in the Leaderboard portion of this manual. In other words, they are "double-checking" their protection code. Find the instruction that compares the checksum value in the accumulator with a set value. Notice the BEQ immediately afterwards that bypasses protection failure. Simply change CMP #$5A with LDA #$00 . We have just set the zero flag permanently, and the routine is tricked.

17) Now that we have finished removing all the dongle-check routines, we need to re-save the file to your backup disk. Type: < S"@0:B",08,9280,BF54 > .

18) Now that you have removed all the dongle-check routines, you have a dongle-less working copy of Leaderboard Tournament Disk #1. It may be backed-up with any data copier. Note: the parameter for LB Tourn #1 on the Utility Disk represents a variation of this break and can be run on a backup and examined with the monitor. You'll find all changes in about the same memory locations.

TENTH FRAME : ACCESS

Procedure:

Use the C-64 Fast Copier to make an exact data-copy of the original. This backup will run like the original ONLY if the dongle is in place. The following procedure will eliminate all dongle-checks.

Working with your backup:

1) Turn on your computer and from the Utility Disk, load the Disk Logger by typing < LOAD "DISK LOGGER",8 > . Then type < RUN > . Insert your backup copy of Tenth Frame in the drive and log it. The two files on the disk that contain code that check for the dongle are called "L" and "S". Take note of the addresses in memory where these programs reside:

```
"L"  $081D - $3FFE
"S"  $6E00 - $9FFE
```

This information is important since we need to load a machine-language monitor into memory where these programs aren't! We can choose from one of three monitors ($2000 = 8192, $8000 = 32768, or $C000 = 49152). The monitor at $C000 does not conflict with Tenth Frame memory, so we will use it.

2) Turn on the computer again and load the $8000 monitor from your utility disk < LOAD "49152",8,1 > followed by < SYS49152 > to execute it.

3) To start with a clean slate, let's clear out all memory below the monitor by typing < F 0800 BFFF EA > .

4) From the monitor, we must load the two Tenth Frame files. Insert your backup copy in the drive and load both files: < L"H",08 >

and < L"S",08 >

5) Now, we will began searching for the certain "dongle-check" byte sequences. We can use the monitor "H" command to hunt through memory for these patterns. Type
< H 0800 9FFF A9 10 24 01 >. After a brief wait, the monitor should return addresses: 0F66 17DO.

6) Disassemble each of these addresses using the < D > command. Use the <cursor-down> key to scroll through the next couple of addresses. At the top after each assembly, change the LDA #$10 command to: LDA #$00 . i.e. - < D 0F66 >,< A 0F66 LDA #$00 >.. do the same for the other address. The rest of the byte changes are performed in this manner, so they won't be in detail!

7) Type < H 0800 9FFF A9 40 4A 4A AA >. Monitor finds: 0FF9 16E6

8) Disassemble both addresses, and change the LDA #$40 command to LDA #$00

9) Type < H 0800 9FFF A9 10 25 01 >. Monitor finds 162B 1E3D.

10) Disassemble both addresses and change LDA #$10 to LDA #$00 .

11) Type < H 0800 9FFF A5 01 29 10 >. Monitor finds 0EEC 11CA 1C5A 2C85 3141.

12) Disassemble each address and change AND #$10 to AND #$00 .

13) Type < H 0800 9FFF A9 10 24 00 >. Monitor finds 1227.

14) Disassemble and change LDA #$10 to LDA #$00 .

15) Type < H 0800 9FFF A9 08 0A EA 31 2B >. Monitor finds 2BB6.

16) Disassemble and change AND ($2B),Y to AND #$00 .

17) Type < H 0800 9FFF A9 D0 49 FF D1 2B >. Monitor finds 2C37.

18) Disassemble the next 9 or 10 bytes. Find the BEQ instruction and replace the next instruction immediately after it with an RTS :< A 2C3F RTS > . The BEQ instruction is executed if the dongle is in, and it hits an RTS too, so putting another RTS after the BEQ guarantees that the program will not crash with the dongle out.

19) Type < H 0800 9FFF 18 A9 00 7D 00 C0 > . Monitor finds 6EC2.

20) Disassemble $6EC2 and scroll down 15 or 16 instructions. Find the BEQ instruction and replace the next instruction after it with an RTS again: < A 6ED9 RTS > . (We just fixed a

dongle-related checksumming problem.)

21) Now that all the dongle-check routines in these files have been removed, we need to re-save the two files to our backup: < S"@0:L",08,081D,3FFF > < S"@0:S",08,6E00,9FFF >

22) There are two other dongle-routines that need to be changed on the Tenth Frame disk. The only problem is that they reside in a file called "P", which loads underneath the KERNAL ROM. There is no way to use our monitors to view, change and re-save this file. Instead, load the Disk Doctor utility by typing < LOAD "DISK D*",8 > and then < RUN >.

23) Insert your backup copy of Tenth Frame and press <RETURN>. Use the < B > command to read Track 20, Sector 1. Use the cursor keys to move to position 63. Using the < @ > command, change the byte value (48) to 32. Re-write the sector with the < r > command. Then use < + > key to read Track 20, Sector 2. Move cursor to position 150. Change byte value (15) to 7. Re-write the sector with the < r > command.

24) Let's investigate why we made the changes in the "P" file. From the $C000 (49152) monitor, load the "P" file from your backup copy by typing <> L"P",08.

25) Since the "P" file resides in memory from $E000-$FEBF, it is now residing in the RAM that is "hidden" beneath the KERNAL ROM's ($E000-SFFFF). Our monitor won't let us view the RAM, so we need to write a short ML program to transfer $E000-$FFFF down to lower memory from $4000-$5FFF so we can look at the "P" code.

Type in the following routine starting at $0334:

```
A 0334 SEI
  0335 LDA #$35
  0337 STA $01
  0339 LDX #$00
  033B LDA $E000,X
  033E STA $4000,X
  0341 INX
  0342 BNE $033B
  0344 INC $0340
  0347 INC $033D
  034A BNE $033B
  034C LDA #$37
  034E STA $01
  0350 BRK
```

25) Type <> G 0334 to execute the routine.

26) Type <> D 550E and cursor-down a few bytes. You should see a dongle-check routine that looks like:

```
 550E  LDA $01
       AND #$30
       ORA #$8C
       STA $3A4E
```

The byte we changed using the DISK DOCTOR on track 20, sector 1 changed the "AND #$30" instruction to "AND #$20". This permanently masks out the dongle-bit to a "0" value, so the computer "thinks" that the dongle is actually in place.

27) Type <> D 575D and cursor-down a few bytes. You should see:

```
 575D  LDA $01
       LSR
       STA $A01F
       AND #$0F
       STA $A027
```

The byte we changed on track 20, sector 2 changed the "AND #$0F" instruction to "AND #$07". This also masks out the dongle bit from location $01 to appear to be on (0 bit).

28) After all changes have been made, your Tenth Frame disk is completely broken and the dongle is no longer necessary.

===

< < < RAPIDLOK PROTECTION REVEALED > > >

Most Commodore users are aware of the standard format that the 1541/71 disk drives read. We can load and save programs, directory the disk, and perform a variety of other commands. The program code that knows how to execute all these functions is stored within the ROM's of the disk drive. Most Disk-Drive Operating Systems are called "DOS". RapidLok is a recent protection scheme that has appeared on the disks of some recent big-name producers (Accolade, Avalon Hill, Microprose...), and uses its own "DOS" system to load files. RapidLok disks will usually have only track 18 standard formatted, the rest of the tracks being formatted in the RapidLok manner. The RapidLok DOS resides in an encoded format on track 18, sectors 18, 15, 12, 9, 6, and 3. Each time a file is loaded through RapidLok, a short machine-language auto-boot file loads the RapidLok DOS from track 18 and stores it in the disk-drive memory from S0300-07FF. Currently, we know of 6 different versions of RapidLok DOS. Each relies on the same basic track formatting, but in addition to loading RapidLok files, they do a complicated check on certain sync lengths, header lengths, and track to track alignment.

RAPIDLOK FORMAT

Like Commodore DOS, RapidLok formats its tracks by first writing a header block, and then a $0255 byte long data block. The method through which RapidLok converts this data into REAL bytes is much too confusing to explain in this overview. The following is how RapidLok would format one track:

1/ The Reference Header:

The first header on a RapidLok track is the track reference header. It is actually a normal Commodore DOS header for that track, sector 0 in GCR format. It is written with a SYNC LENGTH of $0029 bytes, if RapidLok DOS detects a reference header without the correct sync length, the load will abort.

Example:

```
SYNC: $0029 Bytes: 52 57 35 29 6B 74 DC B5  = track 19, sector 0
```

2/ The LONG-SYNC RapidLok Header:

The second header on a RapidLok track is actually the header for RapidLok sector 0. All RapidLok headers begin with a $75, and contain 7 important bytes that the RapidLok loader needs to detect. These bytes are followed by 3 or 4 GAP BYTES that are written out as #$OO's. (Any attempt to read these bytes will return a different byte value each time.) The RapidLok header block for sector 0 (1st header block) has a SYNC LENGTH of $003c bytes, though. The RapidLok loader will fail if this sync length is not found.

Example:

```
SYNC: $003C BYTES: 75 93 59 25 D6 ED 7A 4C 00 00 00 00 = sector 0
```

The remaining headers for sectors 1 through the maximum have SYNC LENGTHS of $0005, and are not checked by the loader.

3/ The RapidLok Data block:

Each data block begins with a $6B value and follows the header for that particular sector. Each data block contains approximately $0255 bytes of data, which is converted into normal DATA and sent from the drive to the computer. Each data block has a sync-length of $0005 bytes, and is not checked by the loader. Sometimes a RapidLok sector will be blank. The data block will then begin with a $55 byte and continue with $0254 more #$55 bytes.

Example of Full RapidLok Data block:

```
SYNC $0005 Bytes: 6B BB C9 24 BA FF 35 DF...........
```

Example of Empty RapidLok Data block:

```
SYNC $0005 Bytes: 55 55 55 55 55 55 55 55...........
```

4/ The RapidLok Bit Rate:

As far as BIT RATES and storage sizes go, RAPIDLOK formats tracks in the following manners for the following zones:

Track Zone	Bit-Rate	# of Sectors
1-17	$60	12
19-35	$40	11

5/ The RapidLok EXTRA-SECTOR

After all the headers and data blocks for each sector of a track are written out, a special "extra-sector" is written on the disk as part of the RapidLok's main protection scheme. The block has a SYNC LENGTH of $0014, and begins with a #$55 byte. The first byte is followed by a certain number of #$7B bytes in a row, giving the entire block a specific LENGTH. A special "decoder" master-key block is written on track 36 of each RapidLok disk. At the beginning BOOT of the program, RapidLok DOS moves the disk-drive head to track 36, reads in the special key, decodes it and ends up with a list of 35 numbers. Each number is the specific length of the EXTRA SECTOR for each equivalent track! During RapidLok file loads, if the DOS extra-sector length does no match the master-key number for that track, the DOS dies. The MASTER-KEY on track 36 is the most difficult portion of RapidLok formats to reproduce.

Example of Extra-sector:

```
SYNC $0014 Bytes: 55 7B 7B 7B 7B 7B.....7B (x amount of bytes)
```

6/ Overview of RapidLok DOS:

Each track contains sectors 0-11 (Tracks 1-17) or sectors 0-10 (Tracks 19-35). Each "sector" is composed of a header block beginning with a $75 and is followed by a data-block beginning with a $6B (or a $55 if blank). Each RapidLok track also contains a reference header AND an extra-sector of special length that must match a "master-key". Remember, during loads, RapidLok DOS is constantly checking the special sync lengths described above. Even the slightest mismatch from the norm will halt the program load. Thus, if your DISK DRIVE speed is slightly off from 300 RPM, you may experience difficulties in loading some RapidLok formatted programs.

If you examine the directory sectors of track 18 on a RapidLok disk with a track and sector editor, you will notice that after each file name is a sequence of two or three bytes. RapidLok DOS actually uses these bytes much in the way Commodore DOS does the track and sector pointer! The actual beginning track and sector number and program length are embedded (encoded) in these bytes.

Little is known about the RapidLok master-key on track 36. The routine that RapidLok uses to decode it can be copied, but actually writing out the key has not yet been done!

On recent RapidLok versions (5 and 6 to be specific), they use TRACK to TRACK alignment. What this means is that if your were on track 19 and you had just read sector 0, if you were to immediately skip the drive-head to track 20 and read the first information you encountered, you would be reading the data for sector 0 of track 20! This is a very simple explanation. Sometimes track-to-track alignment can be done with a "skew". i.e. track 19, sector 0 matches track 20, sector 6, which in turn matches track 21, sector 12. The skew is 6.

RapidLok DOS uses a combination of blank sectors ($55) and full sectors ($6B) on one track. This track must be perfectly aligned with the track before it. When DOS finishes reading the last sector of the first track, it bumps the drive-head to the half and half track. If the track-to-track alignment is correct, it will encounter a full RapidLok sector, and will continue to load. If the alignment is incorrect, even off by one sector!, the drive will encounter an empty sector ($55) and the loader will then commit suicide within your drive! So even if a person could exactly duplicate two RapidLok tracks, he would also have to get the timing within his format routine exact enough to align the tracks correctly.

An example RapidLok Protected track:

SYNC LENGTH	BYTES	DESCRIPTION
$0029	52 55 35 29 4B 74 DC B5	track 1,0 reference header
$003C	75 93 59 25 D6 ED 7A 00	sector 0 header
$0005	6B BB C9 24 BA FF 35 DF	sector 0 data block
$0005	75 92 59 25 D6 ED 6E 00	sector 1 header
$0005	6B DE 59 24 96 7B ED F7	sector 1 data block
.....	....	....
.....	....	....

```
  .....   .......................    .................
  $0005   75 92 E9 25 D6 ED 65 00    sector 11 header
  $0005   6B F7 D9 24 EF 4E AD DB    sector 11 data block
  $0014   55 7B 7B 7B 7B 7B 7B 7B   "extra-sector" for key
```

7/ Points of Interest:

Track 18 on ALL RapidLok disks is formatted in standard Commodore DOS (i.e.- 18 sectors), but it also contains the RapidLok "extra-sector" ($55 7b 7b 7b 7b...etc). The RapidLok auto-boot will not load RapidLok DOS into drive memory UNLESS this extra sector is found. It uses the 2nd byte (78) as a decoder for the DOS stored on sector 18,15,12,9,6 and 3.

On all RapidLok disks released in the past 2 years, tracks 19 through 35 have ALWAYS been formatted in RapidLok style. Tracks 1-17 usually vary, depending upon the program. Huge programs will RapidLok format all these tracks, others will use combinations of standard format with RapidLok format. Often if a game has a highscore list that is saved to disk, the RapidLok format will leave track 1 open as standard Commodore DOS so the high-score list can be written to disk using a simple B-W or U2 command. Writing out in RapidLok format is almost impossible! (it would take up too much disk drive memory!)

8/ In Conclusion:

As we have seen, the RapidLok format is not standard in any way to the format that Commodore DOS is using to reading. Because of this, the only way to break the protection of titles that have the RapidLok format is to break the separate files from the computer memory and tie them together. This, unfortunately is beyond the scope of this manual.

We can however, give you a method of reproducing most RapidLok protected disks. This system (developed by the Kracker Jax team) is the most effective RapidLok copier on the market. In the next section, we will document our RapidLok copier in detail. With our scanner, you will be able to distinguish the RapidLok tracks from the standard tracks and even know the RapidLok copier version. Armed with this information, you will build your own copier driver.

Enjoy!

KRACKER JAX RAPIDLOK COPIERS

RapidLok protection has offered software publishers a very effective means of copy protecting their software. Other copy utility companies have released copiers for a title or two, but because of multiple protection schemes and the extreme difficulty in writing the copiers for those titles, they have been relatively ineffective.

After many months of research and testing, we have developed copiers for what we believe to be ALL existing versions of RapidLok. Unlike our competitors, we have not only developed individual copiers for every version of every title we could find, but have even provided you with an extremely easy way of examining and copying ANY RapidLok protected disk released to date (July 1987). We are confident that if you follow our instructions carefully, YOU will easily construct a copier to archive your particular version of a RapidLok protected disk.

In order to copy a RapidLok protected disk with our system, we must first identify it as such. Companies such as Microprose Accolade, Avantaqe, Avalon Hill, and Capeom are known users of RapidLok. Others do exist and using our system will identify them.

The heart of our system is our RapidLok Scanner. With this scanner you can not only tell if the disk in question is in fact RapidLok, but also the variation of tracks and which version it is. No more guessing and endless backup attempts. You are armed with EXACT information, and that information can be plugged into a Skeleton Copier to provide fast results.

Let's try one out. From the Utility Disk, LOAD and RUN the RapidLok Scanner. When the program is loaded, insert any suspect disk into the drive and press RETURN. If it is a Rapidloked9(tm)N disk, you will see the red and green indicators fill the track line. Last, the version number will appear. The RapidLok disk is made up of two completely different formats. Those tracks shown as red donuts are tracks that must be copied with a RapidLok copier and those shown as green circles must be copied with our Nibbler. The version number shown determines the correct RapidLok copier to use.

After writing down all RapidLok tracks, all regular tracks and the version number, you are ready to create your own copier. Follow these easy steps.

1) Format a work disk. File copy from the Utility disk to your work disk the following files: RLVO, RLVl, RLV2, RLV3, RLV4, RLV5, RLV6, NIBBLER, and COPIER TEMPLATE.

2) When completed, load the BASIC file < COPIER TEMPLATE > (from your work disk directory) and LIST it out.

3) Lines 10 and 20 are reserved for standard tracks. Lines 30 and 40 are for RapidLok tracks. Simply type each track number, seperated by commas, on the appropriate lines. To end the sequence, type a 0 (zero). See the example below:

```
10 data l,2,3,4,5,6,7,8,9,18,31,32,33
20 data 34,35,0
30 data l0,1l,12,13,14,15,16,17,19,20
40 data 21,22,23,24,24,26,27,28,29,30,0
```

This is an example of coping standard tracks 1-9, 18, 31-35, and RapidLoked tracks 10-17, and 19-30. Notice the zeros ending the sequences in each copier type. Do not end a line with a comma, or forget to place information on EACH data line. This WILL cause the copiers to stall.

4) Line 50 contains the title information. You may type any title you wish instead of PARAMETER TITLE. Adjust the quotes to suit the length of the title.

5) Line 60 must contain the proper version number of RapidLok copier to use. Type the correct number in the quotes following the RLV . Be sure to press RETURN to lock in all changes made.

6) When these steps are done, list the parameter out again and double check the changes. If all is well, you may save the file to your work disk. Name it appropriately as it is a custom copier for your title.

To use your custom copier, simply load and run it. The screen will prompt you for disk swaps. When the procedure is done, power down and up again and try your copy. It should run just as the original did. If not, double check your parameter for possible errors.

In Conclusion:

We've found NO RapidLok'ed titles we couldn't back-up. We HAVE had to try a different drive on occassion. Some drives just don't like writing some titles. 1571 drives seem to be extremely effective copiers using this system, but most 1541 drives will work fine. Also, we have had to copy a title or two a bit differently than normal. The tracks had to be copied in an out of order sequence.

Other points of interest are: This system only works with working ORIGINALS. Backups made with other copiers can't be backed up. You may back up second generations of backups made with this system, but you must use the RLVO copier with the correct track sequences (again, use the scanner). The original protection scheme is flakey in loading and the copies are no better (sorry).

INTRO : PROTECTION SCHEME TYPE I

GEOS (Graphic Environment Operating System), from Berkeley Softworks, has revolutionized the way people use their C-64s. It's icon-based, user-friendly, desktop interface has extended the life of this machine to 1990 and lured leery buyers into the world of Commodore computing. With the newly available 1764 RAM expansion, GEOS will allow a C-64 to approach the capabilities of its younger, but more powerful brother, the C-128.

But unlike other operating systems (CP/M, MS-DOS..), GEOS is copy-protected. Who needs a copy-protected operating system? What if you own a large selection of GEOS application programs and your GEOS original crashes? The programs are usel~ss while you try to attain a replacement and you can't borrow a' friend's copy because of the serial number protection! Clearly, it benefits only Berkeley.

Meanwhile, we've been agonizing over which Berkeley releases to cover in this edition of Kracker Jax Revealed. We were reasonably sure that most of you would own GEOS vl.2 and Deskpak I, so we've included those. PLUS a quick-n-dirty way to defeat GEOS vl.3's "Trojan horse" scheme, which will erase your system files if the file "GEOSBoot" fails a checksum test. GEOS vl.3's protection might be covered in a future edition if readers demand it, but its complexity might be intimidating to some.

Be forewarned,though, that the going will be tough if you aren't familiar with "The Official GEOS Programmer's Reference Guide" or Richard Immers/Gerald Neufeld's "Inside Commodore DOS". GEOS and it's protection schemes are heavily I/O bound and good working knowledge of the 1541 drive and GEOS KERNAL routines is essential to understanding the following articles.

Please note:
Geos, Geos vl.2, Deskpak I, Geos vl.3, Berkeley Softworks, and The Official Geos Programmer's Guide are all registered trademarks of Berkeley Softworks.

HOBBLING GEOS vl.3's TROJAN HORSE : BERKELEY SOFTWORKS

The now infamous 'Trojan Horse', is an incredibly sneaky and rather sloppily-executed scheme that deletes your system files "GEOS", "GEOS BOOT", "KERNAL" and "DESKTOP" from an unauthorized copy of GEOS vl.3 while you are rearranging your directory pages. It usually occurs within four moves. It actually doesn't delete the files, it completely zeroes out their directory entries.

The mechanism, located in "DESKTOP", is rather simple. A counter is incremented randomly during directory moves. At certain intervals, a checksum routine is performed on "GEOS BOOT". If the checksum is wrong, the Desktop checks the first four entries of the first directory page for GEOS file-type soc (system boot file). If they match, it fills them with 00's and writes the block back to disk. The disk is no longer bootable unless you can re-create the directory entries.

The GEOS file-type I.D. is located in byte # 24 (18) of each file's directory entry. If this byte is changed to a GEOS system file-type ($04) in the above-mentioned files, the old horse never gets rolled into Troy and you can rearrange your directory with peace-of-mind.

GEOS v1.2 : BERKELEY SOFTWORKS

1) A fast-copied or nybbled copy of GEOS vl.2 will not run. It will merely do a system reset after the protection check. An error scan shows no normal DOS errors but there is data on track 36 (visible with a good GCR Editor). Track 36 is not normally copyable because it has no sync marks.

2) Load the $COOO monitor "49152" from your Utility Disk then load "GE0S" from a backup copy of GEOS vl.2. Disassemble the code at $0123. This routine loads "GEOS BOOT" and jumps to $6000. Load in "GEOS BOOT" and disassemble the code at $6000. Examination of the code reveals that the majority of it is encrypted but the decryption routine at $606C is rather simple. The code will decrypt it for us by placing a BRK instruction at $6086 and executing the code at $606C.

3) Now look at the code again. Sharp-eyed hackers will notice the drive code starting at $623F. Here's some of the other high points of the loader:

   $6167:Print "Booting GEOS...".

$6177:Execute Memory-Write command and output fast-loader routine to drive, then send Memory-Execute command at $61AD.

$6013:Direct I/O to drive through the serial port $DDOO. After the Memory-Execute command is sent, the code at $61BB waits for a signal back from the drive. At $61D4, a byte comparison is done. If it fails, the JMP instruction at $6086 is altered to SFCE1 (C-64 system reset). It then Jumps back to the decryption routine which, this time, re-encrypts the code and then performs the system reset. Let's disable the reset by placing a "BEQ $61EC" at $61D8. Re-encrypt the code by again executing the routine at $606C. Note the new encryption values at $61D8. These will be written to the proper sector on your backup copy.

4) Load the sector editor from the Utility Disk and trace the "GEOS BOOT" file on your backup copy. Address $61D8 would be in the second block of the file (it should be Track/Sector 1/4) starting at byte # $DE (222). Place our byte changes there and rewrite the sector back to the disk. Now reboot GEOS. What happens? No reset this time but the drive shuts off and the screen fills with garbage. The real meat must be in the drive code.

5) Use the sector editor to restore T/S 1/4 back to its original state. Again load the $C000 monitor and "GEOS BOOT". Decrypt the code again as mentioned above. The drive code starts at $623F but we want to relocate to an address we can equate to the actual drive address. This code is written to $0300 in drive memory so lets move our code to $1300 (T 632F 642B 1300). The Memory-Execute command at $60CD jumps to $0375 in the drive so disassemble code at $1375. Remember to add or subtract $1000 from the address references (i.e. JSR $0300 - the subroutine would be located at $1300) when appropriate.

6) Study the code for a while just to get a feel for it. Remember from our scan of the disk that track 36 is suspicious. 36 in hexadecimal is $24. See any references to $24? That's right! At $143A, the accumulator is loaded with the value $24 then the subroutine at $13BB ($03BB) steps the head to track $24 (36). Then a counter of $8000 (32,768) is set up, and a comparison for specific byte values read from track 36 begins. If the counter times-out to zero or all values don't match, the code at $148A is executed. Otherwise it branches to $1485. We want it to branch to $1485 unconditionally. A great place would be at the first byte comparison from $1463 - $1466: if the byte's not equal, make it go to $1485 (A 1465 BNE $1485). Apply this change to the equivalent drive code at $63A4.

Note your encrypted byte changes and use the sector editor to write them to you backup copy. It should be Track/Sector 1/20, byte positions $AE/AF (174/175). Also make sure you have corrected the first change we made. Now reboot the GEOS backup. "Booting GEOS..."... no reset... You hear the drive head swing out to 36 and back. Its loading! The screen clears, the Desktop appears, and ... where's the mouse pointer? The joystick's dead. We've been caught! But how?

8) The most common method is through checksums. If any bytes in the code have been changed, a checksum routine will usually detect it. The protection scheme can then assume tampering and take appropriate action. We could hunt for the checksum code or we could cover our tracks. Let's try covering our tracks.

9) We really only altered one byte in "GEOSBOOT" but we'll have to change a few more to pull this one off. Where could we place our code? A technique we use is to add it right to the end of the file. The last byte of "GEOSBOOT" is at $642B so we can start our code at $642C. But what's going to call our routine? Look for a jump instruction away from the $6000 area. At $621F, the code jumps to the $C000 area. Change that to jump to our code (JMP $642C).

10)Now we have three bytes to correct: the drive code branch address at $63A5 and the JMP to our new code at $6220/6221. Our new code should be similar to the following:

```
A 642C  LDX #$E7 ; restore original drive code BNE address
        STX $63A5
        LDX #$03 ; restore original JMP address - lo-byte
        STX $6220
        LDX #$CO ; restore original JMP address - hi-byte
        STX $6221
        JMP $621F
```

Re-encrypt the code and look at our new code at $642C. It, too, has been encrypted. Write down the encrypted bytes and the new jump address at $6220. We'll write these to the backup.

11)After loading the sector editor, write our new, encrypted jump address to Track/Sector 1/20 - byte position 40 ($28). Then add our new, encrypted code to the last sector in the file - T/S 1/7. Don't forget to change the last byte pointer at position 1 to the last byte of the new code. Using the above example code, the new bytes would be start at position 56 ($38) and the last byte would be at position 73 ($49). Position 1 will changed to 73 ($49).

12)Now reboot GEOS. It should load clean as a whistle. Just remember to watch your step when dealing with protection from Berkeley. They are notorious for their endless checksum routines.

DESKPAK I : BERKELEY SOFTWORKS

Dealing with Berkeley's protected applications presents a two-fold problem: 1) The installation code1 which stamps your GEOS serial number on the master and does a protection check and checksum routine. 2) The I%&%#$'&$ serial number verification that prevents you from taking your GEOS application to a friend's house and using it with his GEOS. Both, however, are relatively easy to break. This will be a general discussion of the first-generation of Berkeley applications, using Deskpak I as an example.

The protection scheme on this first-generation is essentially the same. The code first checks to see if the disk has been installed. If it hasn't, it whips out to Track/Sector 35/0 and reads in the block. The block contains a direct I/O routine and some drive code that looks for non-standard data. If every thing checks out, it installs your internal GEOS serial number to the master (no write-protect tabs allowed). It never does the check again, allowing you to copy the application to work disks. From then on, it does nothing but the serial number check. This works fine in theory, but is rather inconvenient if you want to show it to somebody else and you've forgotten your copy of GEOS.

The protection does checksum itself, however. To bypass this, we'll demonstrate a technique we use called the byte-swap. This entails switching bytes in the code among themselves to force the protection to pass.

Get out your GEOS Programmer's Reference Guide and make a backup of an UNINSTALLED Deskpak I master. Load the "DESKPAK READ" file from the Utility Disk and and run it. The program reads Track/Sector 35/0 into 32768 ($8000) in memory. Load the $COOO monitor ("49152") from the utility disk and study the code at $8000. Look up the GEOS subroutine calls in the reference guide. Half of this code is the drive routine that is sent to the 1541. The other half suspends GEOS I/O and sends the drive routine to the 1541.

The protection check itself is at $803E. It reads in some bytes and compares them. If they all match, it falls through to $8061. Otherwise, it branches to $8064. In fact, its not unlike GEOS vl.2 protection (see previous GEOS vl.2 discussion). We can break the installation protection right here. However, we must contend with a checksum routine located in the main code, so we must keep the bytes intact. A simple way is the byte swap. The code contains many branch instructions. What if we swapped a BEQ (branch-if-equal) and a BNE (branch-if-not-equal) instruction at just the right place? Experimentation will reveal that swapping the branch opcodes at $803c and $804B will force the code to go to $8064.

Write this change to Track/Sector 35/0 using Disk Doctor from the Utility Disk. Load "GEOS" and boot "Graphics Grabber" (the only protected application on the disk). The protection fails. Look at the code at $8061-$8065 again. There are two sets of LDA instructions there, each loading a different value. Why not try another byte swap? Switch the two bytes that are being loaded at $8061-$8065. Now it will be forced to load a different value. Make this change to sector 0 on track 35. You should now have both sets of byte swaps written to 35/0. Boot "Graphics Grabber" again. This time it installs successfully. But you still can't use it with a different GEOS, only the copy from which it was installed.

The serial number check is really the toughest part of some of the applications. Writer's Workshop and GEOdex both try to disguise the call to GetserialNumber", an internal GEOS routine ($c196). One uses encryption and the other uses GEOS's ~callRoutine" which does an indirect JSR (Jump-To-Subroutine) to the serial number routine. An additional problem is that GEOS workspace starts at $0400 in memory, which the C64 normally uses as screen memory. Resetting the computer will lose all the code located from $0400-$0800. Yet another problem is that some of the applications are stored in VLIR (variable length indexed record) files, which are split into multiple parts and special modifications have to be made to the directory to load these files like normal programs. We'll save these for a future exercise.

Deskpak I's serial number check is conveniently located at $2362 on our version. To catch this code, reset the computer while the application is loading. Load the "49152" monitor and disassemble the code at $2362. You'll see this same routine in most of the Berkeley applications. It first checks to see if the serial number is zero. If it is, it executes the install routine that we disabled earlier (the GetBlock and checksum routine starts at $2448). If the serial number is there, it branches to $240D and checks the serial number in GEOS to see if it matches. If it doesn't, it displays a Dialogue Box asking you to reboot with the correct GEOS.

The whole protection and serial number check can be disabled rather simply by placing a CLC (clear-carry-flag) and RTS (return-from-subroutine) instruction at the top of the code ($2362). On our version of Deskpak I, the location on the original is Track/Sector 12/18, byte position # 156 ($9C). You might have to calculate the position or do a manual search of the file to track down the offending code. Write byte values 24 ($18) and 96 ($60) to the appropriate location in the file. You should have no trouble booting "Graphics Grabber" from any copy of GEOS now.

A good Snapshot type utility is helpful for some of the latest applications (GEOfile etc...). They will inevitably place the protection in screen memory and the snapshotter can capture that code for your casual viewing.

DEEP SPACE : SIR TECH

Procedure :

Loading the original produces a rattle-free load, and an error scan shows no standard errors. A backup made with the C-64 Fast Copier produces a non working backup. A backup made with a nybbler produces the same non working backup. Before starting, make a fast copy using the C-64 Fast Copier and use the Disk Logger to log the files.

Working with your backup:

1) The disk log shows us that the boot file "DS" loads into memory from $0302 to $09EB. This means that it starts in the autoboot area and runs through screen memory and into BASIC RAM. Load the boot and you will see the screen react and the program will fail in the first few seconds. This means the protection is probably in the boot file.

2) Turn off the computer and insert Hesmon. X to Basic and load the boot file < LOAD "DS",8,1 > . When the program stalls, hit < RUNSTOP/RESTORE > to activate the monitor. Interpret memory starting at $0801 < I 0801 > because this is the beginning of BASIC RAM. Scroll down through memory and notice the BASIC program there. The Boot starts at the autostart vectors for BASIC and continues on to place a BASIC boot in memory. This is a good way to hide it from the average person. Type < X > to return to BASIC and type < LIST > to see the boot. Inspection shows that this is the protection check as well as the loader.

3) Lets go through the code line by line.:

   1- Lock up the keyboard and set number of trys to 0.

   2- Initialize the drive.

   3- Send Memory-Writes to the drive locations $06/$07 which represent the Track and Sector read into $0300 in the drive.

The Memory-Writes place a 37/0 into those locations (Track 37/Sector 0).

4- Send Memory-Write to drive location $00 (Job Queue) $B0 (dec 176) = Seek any Sector.

5- Set up Memory Read loop of drive location $00.

6- Get value at S00.

7- Set a numerical value for E (M-R value). If trys=500 then test for protection pass.

8- If E has not been read in as an error code ($01-$10) then try all over again.

9- Initialize, close channels, and test E for $01; job completed successfully, and if so then branch to line 10 (pass protection). If not, goto line 10 and crash.

10- Jump to $02A7 and crash (because no loader has been poked in.

11- Poke in a loader and JUMP to it.

4) Armed with this information, the way to break this code easily is to delete line 10. One way to do that is to put a REM right after the 10 which will nullify the whole line. The REM instruction is actually represented by one byte called a token. It is a 143 in decimal. We can easily install the byte with Disk Doctor.

5) From the Utility Disk, load Disk Doctor < LOAD "DISK D*",8 > and < RUN >. At Track/Sector 18/1 Cursor to position 3 and < j > Jump to the first sector of the DS file. Use the < n > key to follow the file to Track/Sector 31/4. Cursor to position 232 and use the < @ > key to change the Poke byte to a REM with a 143. Hit < r and y > to rewrite the sector.

You'll find that the backup works perfectly now and can probably be file copied.

===

GRAPHICS INTEGRATOR II : INKWELL

Procedure:

Loading the original produces a rattle-free load, and an error scan shows no standard errors. A backup made with the C-64 Fast

Copier produces a non working backup. A backup made with a nybbler produces the same non working backup. Before starting, make a fast copy using the C-64 Fast Copier.

Working with your backup:

1) Before beginning the break, let's repair the directory so we can view our files. From the Utility Disk, load the Disk Dr. as < LOAD"DISK D*",8,1 > and RUN. When the title screen comes up, insert your backup and hit RETURN. The Track/Sector brought up will be 18/1 which is the first sector of directory entries. To repair the directory, you must fill the following positions (in decimal) with shifted spaces (decimal 160).

```
 pos   4-44
 pos  72-76
 pos 104-108
 pos 136-140
 pos 169-174
 pos 200-204
 pos 232-236
```

   When your changes have been made, hit <r> for rewrite and <y> for yes. Now hit <n> for the next block (Sector 7) and make the appropriate changes to that Sector (pos 12). Again rewrite the track and hit <n> to go to the next block (sector 5). Notice the first two bytes direct the load back to Track 18/Sector 1 which causes the endless directory. Using the <@> key, change position 0 and 1 to 0 and 255 respectively. Again be sure to rewrite the Sector.

   Finally with the <b> command, go back to Track 18/Sector 0. Repair the title and ID by using the <t> text command and placing spaces at position 144-148 and at pos 162 give a new ID number such as GI and again rewrite the sector. Power down and check the directory. It should appear normal.

2) With the directory repaired, you may use the Disk Logger utility from the Utility Disk to log all files on the backup. < LOAD"DISK LOGGER",8,1 > . Inspection of the log shows a file that resides in BASIC memory starting at $0801 which is the beginning of BASIC. Let's check it out. Power down, insert your Hesmon cartridge and power up again. <X> to BASIC and load the ME file < LOAD"ME",8,1 > . List the file out. Lines 600-630 represent the call for the protection check. Let's examine the call, line by line.

   600 Open channels, initialize, set the Track (T) to 34 and Sector (S) to 8.

610 Open a channel to the drive.

620 Send a Block Execute command to the drive. CHR$(66)=B CHR$(44)=-CHR$(69)=E. In other words read Track 34, Sector 8 from the disk and send it to a buffer in the drive. Execute that code starting at the first byte.

630 Close channels : RETURN to GOSUB that called the check in line 65.

3) Let's examine the Block Execute code. From the utility disk, load the program called BLOCK READ. < LOAD"BLOCK READ",8,1 > . list the code and in line 10 set the TRack to 34 and the Sector to 8. Place the backup in the drive and type RUN. The drive will read the proper block and transfer the code to $C000 in the computer memory. When the READY prompt comes up, hit RUNSTOP/RESTORE to enter the monitor.

4) Begin disassembly at $C000 < D C000 >. Examine the code from $C015-$C02A. The drive reads Track 35/Sector 0 through the job Queue. The Error message is read at position $00 and if equal to $02 (header block not found), the code falls through and places a value of $7F at $003B in the drive and returns to the BASIC program that called the B-E in the first place. If the check is not satisfied, a Branch is taken to $C038 which causes the head to go to track one and go in an endless loop.

5) The break is now quite simple. If we place two NOPs at $C029 and $C030, the code will not be able to Branch and must fall through even if the protection doesn't pass. The changes can be made with Disk Dr. Power down and remove your Hesmon cartridge. Power up and with the Utility Disk in the drive, < LOAD"DISK D*",8,1 > . Use the <b> command to read in Track 34/Sector 8 from the backup. At pos $29 (decimal 41) you'll find the BNE command. Using the <@> key, change position 41 and 42 to 234 ($EA=NOP).

6) This title is now broken and can be fast copied with any data copier. Because it still uses the B-E command, you will not be able to file copy. One way to possibly break the B-E code might be to store the $7F at $3B in the drive using a M-W (memory-Write) command. Replace the B-E in the Me file with a M-W (Line 620). We will leave this to you as an exercise for further practice.

INTRODUCTION To K.J. REVELED III

« Publisher's Notes »

Welcome to Kracker Jax Revealed Vol III. We at Kracker Jax want to thank you for your purchase and let you know that we do appreciate your support of our products.

First of all, we'll assume that you have read Kracker Jax Revealed Vols I & II (the previous sections in this manual) and that you've performed many of the procedures in those sections. The format of Vol III has changed substantially. Although we've retained the cookbook approach, we have been forced to drop the major types. Protection has progressed to the point of excellence (in some cases) and is often better than the programs that it protects! Most programs today are protected in very individual styles. In this edition of Kracker Jax Revealed, we try to hit the highlights and prepare you for your trek ahead.

Please understand that we can't be responsible for the machine language training that must be done before you can thoroughly understand the procedures and principles set forth in this manual. You don't have to be a fluent MIL programmer, but you MUST have a cursory knowledge of MIL and a strong natural curiosity. Don't expect to discover (as some beginners do) a generic method of de-protection. It just doesn't exist. We can and will give you hints, tips, and technique8 that can be applied to other programs, even if they are a completely different protection type than discussed in this manual.

Finally, many protection schemes are based on the fact that no standard or nybble copier on the market can duplicate the program data. This protection becomes even harder to back up. No longer are we dealing with a sector or track of special protection; EVERY byte on the disk is protected. These programs must be either broken from memory or have a special copier developed to duplicate that program's format. Both of these methods are far too complicated to discuss within this manual. As you become more and more proficient at the patch method, the memory break method will become obvious. Writing copiers is in the realm of DOS experts that have a complete knowledge of NIL. Leave the special copiers to them.

Kracker Jax Revealed Vol III has many features worth mentioning. Berkeley fans will really enjoy Bob's new work on GEOS. He shows us exactly how to use Super Snapshot to obtain a working copy of GEOS that may be booted from ANY drive. Also, for those of you more inclined to know the internal workings of GEOS protection, Bob has done a great job on GEOS v2.O. We know you'll love this one.

After trying out the many break routines throughout this manual, you'll want to check out the Protection Scheme Bection. We show you how to create and use disk protection. Learning by doing is a great way to expand your knowledge.

For those with the courage, we suggest the V-Max! Section. Be warned, a good knowledge of the 1541 is mandatory.

Also, as promised, we have included the Hacker's Utility Kit on your work disk. We have done a slight re-format to allow those with PAL (European) systems to load this software. Because the PAL System is very different from the U.S. Commodore, we can't guarantee that all the features will work properly. Sorry.

« Author's Notes »

When I first started breaking copy protection routines, there was no such thing as "too much" information. I spent a fortune combing BBS's across the country looking for hints and tips. Every publication that even hinted at protection information eventually found its way to my door. I first became associated with Kracker Jax after they had released KJ REVEALED VOL I, which filled in several gaps in my copy-protection education and confirmed that I was on the right track in other areas. I gained enough confidence to submit a parameter to Kracker Jax that was eventually published. I was subsequently asked to contribute several pieces to REVEALED II, which I was glad to do.

If some of the tutorials in Revealed III are over your head, don't be discouraged. There is no "easy" way to learn protection removal. It takes the patience of a saint and a willingness to spend long, backbreaking hours at the console, oblivious to the hole being burned in the back of your neck by your spouse's disgusted stare. Most of all, it takes a thirst for knowledge and a competitive nature that will not bend to the will of the PUZZLEMASTER.

The software protection war is not a myth: there is plenty of evidence that protection programmers ARE paying attention to what we are doing and ARE taking steps to make it harder.

Bob Mills
Programmer

< < < BERKELEY SOFTNORKS : GEOS 2.0 > > >

Warning: Trying to understand this chapter may be hazardous to your mental health. If you haven't read "Inside Commodore DOS", "CSM's Program Protection Manual Vol. 2", and "The Official GEOS Programmer's Reference Guide" at least twice, cover-to-cover, then turn the page ....

The copy protection routine in GEOS has been a thorn in the side of everyone who ever needed a working backup of their original. A backup copy of GEOS only boots and loads properly when all of the several layers of protection checks have been satisfied perfectly. We found this out the hard way with our first GEOS 1.3 parameter. What appeared to be an ideal way around the protection check turned into a nightmare. Customers complained that file selection dialogue boxes acted strangely; that the dreaded "SYSTEM ERROR NEAR $XXXX" appeared at odd times; and that; sometimes, the GEOS System files would inexplicably disappear.

That we had failed was obvious. What was not obvious was the subtle complexity of the protection scheme. It took almost a week of sleepless nights to come up with a satisfactory solution to the problem. If you're still game, let's analyze exactly what GEOS BOOT does and how it does it.

Prepare a fast copy of your ORIGINAL GEOS 2.0. It should contain little or no modifications to the disk structure and directory, especially the System Boot Files "GEOS", "GEOS BOOT", and "KERNAL". Make sure you have a work disk ready so you can save code to it. You will also need a reset button and the "GMON" drive monitor on the Revealed III utility disk to conveniently follow the boot routine from its humble beginning to the bitter end. "GMON" is a modified "Kracker Mon" and is NOT relocatable. It was assembled to occupy C-64 memory from $2000 - $3FFF, which GEOS ignores until the inevitable entrance of "DESK TOP". It may be activated from BASIC with the command "SYS 8192".

If you use the included Disk Logger, you will find that "GEOS" and "GEOS BOOT" (GB) load respectively from $0110 - $0206 and $6000 - $64A9. Using "GMON", load and examine "GEOS" in memory. No funny stuff here. Its only purpose is loading and executing GB. You may safely ignore this file and directly load GB with "GMON".

The next step is to browse through the program code. You'll find a lot of areas that don't disassemble properly because the code is encrypted. The decryption routine is actually fairly simple. It may be seen near the bottom of the GB file at $6483 in memory. The program code from $6140 to $6440 is encrypted with the value $C9: we'll need this piece of info later. To view the program in an executable state, change "JMP $6140" at $64AO to "JMP $64AO" This creates an infinite loop from which we can safely press the reset button.

Start the decryption process from GMON with the command "G 6000". The familiar "BOOTING GEOS ..." message appears on the screen, the drive whirs for a few seconds, then ... nothing. Press the reset button and re-activate GMON from BASIC (SYS 8192). Again browse through the program code. Things look a little less confusing now.

It's not immediately obvious where the call to the decryption routine takes place. We do know that our infinite loop at $64AO did not happen until AFTER the disk drive was accessed. Lets start from the top:

$6000: JMP to $60A8

$60A8: C-64 KERNAL system and non-maskable interrupt vectors initialized. Sprites are turned off. Screen memory is cleared, color memory filled, and the text "BOOTING GEOS..." is written directly to screen memory.

$60EB: Check if GEOS BOOT should load from disk or RAM Expansion Unit (REU).

$612A: Prepare for loading the fast loader (turbo) and protection code to the drive. The JSR to $6081 at $613A should be examined closely - this is where the decryption routine is called after the drive is initialized. Notice that the values $64 and $82 are placed into the C-64 Stack area ($0100 - $01FF). When the RTS at $60A2 is executed, the microprocessor will pull these two values from the stack and add 1 to get the return address ($6482 + 1 = $6483).

$6140: This is the entry point after the decryption is complete. Here, the turbo code is being transmitted to the drive in a convoluted way - appropriate because the drive code itself is scattered in pieces throughout the program. As if fragmenting wasn't enough (it eventually wasn't), the turbo code is also BACKWARD! Backward and in pieces, the turbo code is eventually reconstructed in the 1541 drive RAM and finally activated at $6192.

$61A1: Begin receiving data from the drive. Three separate program segments are loaded using zero-page indirect addressing mode ($04/$05 contain the current address being loaded). The firstsegment is loaded into S9000. GEOS keeps its disk turbo code here, regardless of the drive type. Without an REU, GEOS programs must swap the turbo code for different drive types (1571 or 1581) in and out of this reserved area as needed. Desk Top does this (rather poorly sometimes).

$61B1: Get random value from the C-64 VIC raster interrupt and store it to $02FE. This becomes the seed value for the GEOS serial number generated when the original disk is first booted (installed).

$61B7: Load second segment to $5000: This is the cold start routine to activate the GEOS KERNAL. If an REU is present, the code at $C000 is copied here (see $60EB above).

$61C2: Load last segment from $BF00 to $FFF9. This is the actual GEOS RERNAL. The first protection check by the drive is executed prior to this. If the check fails, no KERNAL code is sent. The computer checks $05 (the load address high byte) for any change from its initial value ($BF). If it still equals $BF, the protection check failed and GEOS BOOT resets the computer (JMP $FCE2).

$61D6: The protection passed and a second VIC raster value is stored to $02FF for serial number generation if this is a firsttime load. Any open drive channels are closed and GEOS BOOT jumps to $5000 (KERNAL cold start) indirectly through the jump address stored at $C003.

Now that we have a better idea of the protection's strategy, let's take a peek inside the drive. Reload "GEOS BOOT" and again create the infinite loop at the bottom of the decryption routine. When the computer freezes up, press your reset button and reactivate "GMON". Using the "M" (monitor) command, look for "M-E" (Memory-Execute) text in memory between $6000 and $64A9. When you find it (at $61FB on our version), remember the execution address: $0457.

To trap the drive code in a viewable state, we need to make the drive shut down without resetting. Drive memory is normally wiped out during a reset. We'll change the M-E address to a DOS routine that will exit gracefully and allow us into the drive. Fairly reliable is TURNOFF (turn off drive motor) at $F98F. Because the M-E command is encrypted, we'll add a short routine to change the drive address to the correct value. Reset the computer, activate "GMON" and reload "GEOS BOOT" (sigh) again.

```
At $64A0, enter:  A 64A0 JMP $64A9
At $64A9, enter:  A 64A9 LDA #$8F   ;change M-E
                  , 64AB STA $61FE  ;address to
                  , 64AE LDA #$F9   ; TURNOFF
```

```
64B0 STA $61FF   ; ($F98F)
64B3 JMP $6140   ;continue...
```

Start up the boot again (G 6000), but this time, as soon as you hear the drive motor turn on, UNPLUG THE SERIAL CABLE FROM THE BACK OF THE COMPUTER. DO NOT TURN OFF THE DRIVE! Reset the computer, activate "GMON", THEN reconnect the serial cable to you computer. Using "GNON's" drive monitor, enter drive memory and IMMEDIATELY transfer the drive code from $0300 to $07FF in drive memory to a safe area of memory in the computer. How about $8300 - $87FF ?

After the transfer has completed, reset the drive and save the drive code from computer memory to your work disk. Now that it's safely stored, print a disassembly of the code.
Look through it carefully before you read any further. Ready? Nervous? Do you have 'Inside Commodore DOS" open and waiting? Lets DO IT!

$0457: Disable interrupts, save stack pointer, and signal computer that the data will be coming soon.

$0466: JSR to MAIN LOOP of loader.

$0483: Set up buffer pointer for data buffer at $0600.

$048B: Read and send first segment (turbo code). First track/sector is $l3/$0D and is stored at $0528/$0529 for use by other subroutines.

Let's stop here. Using a sector editor or "GMON" drivemon, look at the first sector of the GEOS KERNAL. This is a block of track/sector pointers (GEOS VLIR file). Our GEOS shows 3 file chains starting at $13/$0D (!!!), $14/$11, and $14/$0F. WRITE THESE DOWN! (Your GEOS may have slightly different values but the concept is the same).

JSR $04CF: Main subroutine to read and transmit the data. Tracing it through reveals a fairly standard fast loader. I won't go into detail about these subroutines unless they're directly related to the protection scheme. If you want to understand how each of the DOS and Floppy Disk Controller routines work, READ THE REFERENCE GUIDES MENTIONED ABOVE AND TRY ALL OF THE EXAMPLES!

The data transmission routine from $03FF - $0456 is VERY significant. Stay tuned ...

$0490: Here's where the nastiness really starts. A value of #$59 is stored to $0413. Big deal,right? Look what effect it has on the transmission routine:

```
LDA #$59   >   $0413: 2C 2A 04   BIT $042A
      ||   >          ||         |||
      ||   >          ||         |||
STA $0413  >   $0413: 59 2A 04   EOR $042A,Y
```

The innocuous BIT instruction has instantly been transformed into EOR - the favorite scrambling tool of copy protection programmers everywhere. Every sector transmitted from this point on will be EOR'd with the drive code before it's sent to the computer. Consider what happens if just ONE byte of the drive code from $042A - $0529 is altered: the main GEOS KERNAL, excluding work areas and disk drivers, is approximately 16384 bytes. If 1 byte of every 254 is wrong, we have 64 bytes with unknown values occupying our operating system, a system error for every occasion!

$0495: The next few instructions should seem familiar if you've been reading closely. They start the load of the second segment - the GEOS cold start routine at $5000. Look again at the VLIR block of the GEOS KERNAL: the third set of track/sector pointers reads - you got it - $14/$0F, consistent with what we've learned so far.

$049F: Something different is happening here. If you've done your homework, you'll recognize the 1541 SEARCH subroutine $F510. This searches the current track for the specified sector header GCR bytes, the first eight of them significant and the rest as filler preceding the sector data block. If SEARCH fails to find a sync mark and 1541's normal error handler instead of returning to the fast loader.

$04A7: And here's the main attraction, ladies and gentlemen. Read two GCR bytes with JSR $04F3.

$04C2: Congratulations! You've just entered the BYTE COUNT ZONE. The protection check is checking the tail gap of every header and data block on the current track ($14) for 2 precisely located bytes. The .X register contains the sector count ($13 = 19 dec). The protection check loops as follows: JSR $0502: This routine waits for either a GCR $55 or $67 in the current header/tail gap. If neither byte appears, the return address is pulled off the stack. The protection has failed and is getting ready to call it a day.

$04B0: Count $100 (256) GCR bytes on the track.

$04B5: Count $45 (69) GCR bytes on the track. We've just counted to the end of the data block.

$O4BA: JSR $0502 (see above) to check this tail gap.

$O4BD: Count $OA (10) bytes on the track. This is the next header block.

$04C2: We're back to the top of the loop. JSR $0502 (see above) to check this header gap. Decrement the sector count. If zero, we're done1 otherwise branch back to $04B0.

$04C8: We've passed the protection check. Read and send the third and last segment at track/sector $14/$ll (remember the KERNAL VLIR sector ?).

The drive code has done it's job and exits. Now how do we disable the protection check without scrambling the data?.
You might have noticed that the drive's BAM buffer from $0700-$07FF is totally unused by the drive code. If we copy the block of drive code that's being used as the decryption key to $0700 and change the BITIEOR address at $0413 to look there instead, we can freely alter the protection check. Change the LDY $lC00 at $0502 to read JMP $O4FD and the 2 bytes ($55 and $67) will never be checked.

Getting inside the drive during the loading process presents a problem, however. Remember that the drive code is stored in pieces in GEOS BOOT. Alterations there would be tedious and mistake-prone.
But if our code was already waiting inside the drive, all we have to do is change the M-E address that GEOS BOOT sends (the same one we changed in the first place) and we're in-like-Flint. When GEOS BOOT starts, the disk BAM (track/sector $12/$00) is sitting at $0700. There is empty space in the BAM from $07A0 - $O7FF: a great place for extra code.

But how can we copy the drive into $0700 if we're there? We would destroy ourselves. The answer is to make our BAM code load our copy/alter routine into drive buffer $0600. We then jump to THAT code, which copies the drive code to $0700, alters the protection check, and jumps to $0457 (fast loader entry point).

If this sounds complicated, it's because it IS. Use the provided GEOS 2.0 parameter on your backup copy and examine thE BAM code. It will clarify what we've been discussing.

We're still not finished with GEOS BOOT! There is ANOTHER protection check that drove us crazy until we found it. The last sector of the KERNAL that's loaded remains in the drive at $0600 when the drive code exits. The sector's last byte pointer is set at $3D. But PAST that code, at $4E is ANOThER check for the $55/$67 byte pair. This is called from the turbo code (first load segment) during the KERNAL cold 8tart. Place an RTS ($60) at position $4E to kill this little terror.

And then there's the matter of the TROJAN HORSE routine in Desk Top that will delete the SYSTEN BOOT files from your disk if it detects any changes in GEOS BOOT. To date, we have found four versions of Desk Top containing this check, all slightly different and very hard to pinpoint if you don't have a sound working knowledge of the internal workings of GEOS. Again, use the provided Desk Top parameter to explore this further.

As a final exercise, use the included GCR editor tolook at the header and tail gap bytes we discussed above. They can be found at position $OA in ANY header block and position $145 in ANY data block on your ORIGINAL GEOS boot disk.

In closing, we hope you have a better understanding~of what kind of effort can go into finding and disabling a protection scheme as complex as this one. It's easy to complain about copy protection... but doing something about it is a whole new ball game.

< < < HOW TO SNAPSHOT GEOS 1.3 & 2.0 > > >

If you've ever tried using Super Snapshot's (SSS) excellent archiving talents on GEOS, you know that any interruption of GEOS, even with a hardware device, will ultimately produce a total system freeze or crash. There are several minor reasons this occurs but only one major reason: GEOS uses custom drive "turbo" code to speed up disk accesses. It is almost always "talking" to the currently active drive via the serial port at $DDO0 while the drive is checking its end of the serial bus ($1800 in drive memory) for any command signals (load, save, etc...).

GEOS keeps track of the state of the drives through 4 status bytes (called TURBO FLAGS) located at $8492 - $8495 in computer memory. Each of these 4 bytes corresponds to GEOS drives A through D or DOS devices 8, 9, 10, and 11. If the status byte contains $00, the drive is either inactive or is not running the turbo code (i.e. available for normal DOS commands). A status of $80 indicates that the turbo code is present in the drive but not active. Finally, a status value of $c0 means that the turbo code is up and running.

When the SSS button is pressed, the entire state of the computer is preserved. But the drive(s) running the turbo code are still waiting for a command signal from GEOS. At this point, any attempt to communicate with the drive through DOS is fruitless - unless the drive is turned off and on again. Now the drive can be accessed normally and the Snapshot process can be completed. However, when the Snapshotted GEOS is re-booted, it will continue no further BECAUSE THE TURBO FLAGS STILL SHOW THAT THE DRIVES ARE RUNNING THE TURBO CODE ! GEOS assumes that the turbo code is active and will try to signal the drives, GEOS-style. The drives will, of course, not respond properly (if at all) and the operating system, by now totally confused, heads for the remote island of Catatonia to sort it all out.

Fortunately, GEOS Desk Top allows us to get our foot in the door through the RESET option located in the SPECIAL menu. This option clears the screen, re-initializes the drive(s), and opens the current disk(s). Perform steps 1 through 9 EXACTLY as described to properly Snapshot GEOS.

1) Boot GEOS to the Desk Top. Your system should be configured to your liking (number and type of drives, etc..). If not, do it now.

2) Format a disk to contain the Snapshotted GEOS files. This will become your new boot disk.

3) Copy the following GEOS files to your new boot disk:
   a) "DESK TOP"
   b) "CONFIGURE"
   c) "Preferences" [optional].
   d) "Pad Color Pref" (GEOS 2.0) [optional].
   e) Your current input driver file (Ex: "COMM 1351")
   f) Your current printer driver file (Ex: "MPS-801")
   g) Any other desired files, as long as you leave at least 58 kbytes (237 disk blocks) free.

4) Place your new boot disk into the drive from which the Snapshotted GEOS will be booting.

5) Open the SPECIAL menu and click RESET. You now have exactly 1.6 seconds to press the SSS button (for stopwatch buffs) -OR- press it before the screen clears completely. It's a good idea to practice a few times (pretend to press the button) until you feel confident enough for the real thing.

6) Confident, eh? Repeat step 5 but actually press the button.The SSS sub-system menu should appear. If it doesn't, Keep trying step 1 and steps 4 through 6, until step 6 is completed properly.

7) Turn off all drives for at least 5 seconds. Turn them back on.

8) Enter the 555 ML Monitor. Type the following exactly:

```
:8492 00 00 00 00
```

Now press RETURN. This resets the TURBO FLAGS to reflect the new status of the drives - no disk turbo; normal nos active.

9) Exit the monitor (X   RETURN) to return to the SSS sub-system menu and select the SNAPSHOT option. Save the program to your new boot disk. When the sub-system menu returns, select RESUME EXECUTION.

When Desk Top reappears, there will be a slight delay as GEOS uploads the turbo code to the drives. The RESET sequence will then continue as if nothing happened. When the RESET is complete, use the Desk Top to place the first file that SSS saved (the boot file) to the top of the directory for easy loading.

* * N~O T E * *

The only limitation to the method in this article is the lack of automatic drive type detection and configuration that occurs when booting from the GEOS SYSTEM boot disk. If you use different combinations of drives for various applications, create a boot disk for each of these unique combinations. For example: if the new boot disk was created while using a 1571 as the boot drive, don't copy the Snapshotted GEOS file(s) to a 1541 or 1581 and expect it to boot properly from that drive. GEOS only has enough space in the operating system code to handle 1 drive type at a time. This is not the case if a Ram Expansion Unit (REU) is detected. Up to 3 disk drivers are automatically stored in and accessed from the REU by the CONFIGURE utility and Desk Top. The CONFIGURE utility was added later to allow GEOS to support new drives as they appeared. Beginning with GEOS 2.0, it is the application's responsibility to move the appropriate disk driver code in and out of the reserved area.

< < < EPYX : DEATH SWORD Vl/V2 > > >

If you have studied the procedures set forth in the Rad Warrior section, you'll find Death Sword protection to be very similar. At this time, we have found two almost identical versions of Death Sword protection on the market. Both versions are identical to each other except where noted as V2.

You will need the following:

1) An original "Death Sword" (DS) diskette.

2) A backup copy of both sides DS using any good nybbler.

3) A disk log of the DS disk to get the load addresses.

4) An error scan of the original DS disk.

5) A reset button that will reset the screen.

Examining the disk map shows that the disk appears to be completely normal. This is common to most Epyx releases. They have an impressive fast loader routine that requires a slight modification to the sector headers. A fast copier will ignore these eccentricities, but a nybbler can reproduce them well enough to fool the fast loader. obviously1 this isn't where the protection lies.

Load the nybbled copy of DS and observe what happens. When the fancy "EPYX" screen appears, the disk drive stops and the computer takes a permanent time-out. This, then, is where the protection check occurs.

The DS boot file resides from $02A7 - $0303. The program start address can be found in the BASIC warm start vector in $0302 - $0303. The entry point is $02C1. This routine does little more thanload the only other file in the directory "(C) 1987 EPYX" and then jumps to $0600. The file resides from $0409 to $0618: SCREEN MEMORY! This makes it a little tougher for us to examine. A software based monitor like "Kracker-Mon" has to use screen memory to display. Anything loaded there will be immediately destroyed. We must relocate the file as we load it.

Load the $COOO monitor and relocate the file by entering:

```
   L"(C)*" ,08,1409
```

The file will now reside at $1409. Begin disassembly at the entry point of $0600 (for consistency's sake, I'll refer to the actual address. Just add $1000 to any address within $0409 - $0618). You should be looking at a short routine that ends with a JMP to $67E9 at $0614. Examine the other subroutine calls to $05F1 and $05F4. These are the initialization routines that start the drive code and fast loader. A logical place to stop the loading process is the JMP $67E9, but its location (screen memory) requires us to use the supplied File Tracer utility to patch this JKP on the nybbled backup disk so that it JMP's to itself (JMP $0614). Then we'll reset the computer and check the code at $67E9.

After applying the patch to your backup, boot it. The program should freeze up. Press your reset button and load the $COOO monitor. Disassemble the code at $67E9. The subroutine call to $68CA (V2 $68E6) reveals several calls to the load routines in screen memory, followed by a comparison to a byte value at $68E6 (V2 = $6902). If the byte doesn't match, the code branches to $68EF (V2 = $690B), where it executes an undocumented opeode ($02) that sends the computer into an infinite loop. What would happen if we just bypassed this code altogether? Again, we'll have to patch the backup disk.

But where is this code? Try to find it with the Byte Pattern Searcher. You won't find it. Epyx' fast load routine requires the disk data to be written a special way that Commodore Dos doesn't understand. But we CAN patch the code in memory, after it's loaded.

Use the drivemon (see Rad Warrior elsewhere in this manual) to load the last sector of the "(C) 1987 EPYX" file
(T/S 17/4 or $l1/$04).

Change the JMP $67E9 at position $13 (V2 $14) to read:

```
   LDA #$60                ;An "RTS"
   STA $68CA (V2   $68E6) ;is placed at top of
   JMP $67E9               ;of protection check
                           ;and then JMP
```

You also must alter the last-byte pointer at position 1 in the sector to reflect our added code (from $16 to $1A (V2 $lB)) so that it loads properly. Write the sector back to the nybbled backup and boot it. It worked! The protection check is bypassed. You may apply the same procedure to the other side of the disk.

< < < EPYX : RAD WARRIOR > > >

Epyx, like many other major software producers, uses many different protection schemes in their program releases. The complexity of the protection is apparently related to anticipated sales of the release. Hence, their "U.S. Gold" and "Maxx Out" (bargain division) series are easily nybbled, with only a few requiring a (usually) short parameter. "Rad Warrior" falls into this group - it appears that the protection on this title was designed to thwart only software based nybblers. The actually protection is easy to disable - once you find it.

You will need the following:

1) An original "Rad Warrior" (RW) diskette.

2) A backup copy of RW using any good nybbler.

3) A disk log of the RW disk to get the load addresses.

4) An error-scan of the original RW disk.

5) A reset button that will reset the screen.

Examining the disk map shows that the disk appears to be completely normal. This is common to most Epyx releases. They have a VERY fast loader routine that requires a slight modification to the sector headers. A fast copier will ignore these eccentricities but a nybbler can reproduce them well enough to fool the fast loader. Obviously, this is not where the protection lies.

Load the nybbled copy of RW and observe what happens. When the "Maxx-OUT" screen appears, the disk drive hangs. If you listen closely to the drive when this happens~ you will hear the drive head move a long way across the disk before it goes into a coma. This, then, is where the protection check occurs.

The RW boot file resides from $02A7 - $0303. The program start address can be found in the BASIC warm start vector at $0302 - $0303. The entry point is $02C1. This routine does little more than load the only other file in the directory ("(C) 1987 EPYX") and then jumps to $0600. The file resides from $0409 to $0626: SCREEN MEMORY ! This makes it a little tougher for us to examine. A software based monitor like flKracker~MonN has to use screen memory to display. Anything loaded there will be immediately destroyed. We must relocate the file as we load it.

Load the $C000 monitor and relocate the file by entering:

```
L"(C)*",08,1409
```

The file will now reside at $1409. Begin disassembly at the entry point of $0600 (for consistency's sake, I'll refer to the actual address. Just add $1000 to any address within $0409 - $0626). You should be looking at a short routine that ends with a JMP to $67E9 at $061E. Examine the other subroutine calls to $05F1 and $05F4. These are the initialization routines that start the drive code and fast loader. A logical place to stop the loading process is the JMP $67E9, but its location (screen memory) requires us to use the supplied File Tracer utility to patch this JMP on the nybbled backup disk so that it JMP's to itself (JMP $061E). Then we'll reset the computer and check the code at $67E9.

After applying the above patch to your backup, boot it. The program should lock up. Press your reset button and load the $COOO monitor. Disassemble the code at $67E9. The subroutine call to

$6909 reveals several calls to the load routines in screen memory, followed by a comparison to a byte value at $6925. If the byte doesn't match, the code branches to $692E, where it executes an undocumented opeode ($02) that sends the computer into an infinite loop. What would happen if we just bypassed this code altogether? Againg we'll have to patch the backup disk.

But where is this code? Try to find it with the Byte Pattern Searcher. No Go, Joe! Epyx' fast load routine requires the disk data to be written a special way that Commodore DOS doesn't understand. But we CAN patch the code after it's loaded into the computer. Use the drivemon to load the last sector of the "(C) 1987 EPYX" file (18/5 or $12/$05). With the Kracker-Mon in drive mode, initialize the drive and place a $12 in location $06 and a $05 in location $07. By placing an $80 in location $00 and pressing RETURN, you can read the sector into the $0300 buffer in the drive.

Change the JMP $67E9 at position $031D to read:

```
   A9 60      LDA  #$60    ;An "RTS"
   8D 09 69   STA  $6909   ;is placed at top of
   4C E9 67   JMP  $67E9   ;of protection check
                           ;and then JMP
```

You must also alter the last-byte pointer at position $0301 in the sector to reflect our added code (from $031F to $0324) so that it loads properly. Write the sector back (place a $90 in position $00 and press RETURN) to the nybbled backup and boot it. It worked!

The protection check is bypassed.

< < < EPYX : SPIDERBOT > > >

Epyx, like many other major companies, uses many different protection schemes in their software releases. The complexity of the protection is usually directly related to anticipated sales of the release. Hence, their "U.S. Gold" and "Maxx Out" (bargain division) series are easily nybbled, with only a few requiring a (usually) short parameter. "Spiderbot" is one of these: it appears that the protection on this title was designed to thwart only software-based nybblers. The actual protection is easy to disable -
once you find it.

You will need the following:

1) An original "Spiderbot" (SB) diskette.

2) A backup copy of SB using any good nybbler.

3) A disk log of the SB disk to get the load addresses.

4) An error scan of the original SB disk.

5) A reset button that will reset the screen.

Examining the disk map shows that the disk appears to be completely normal. This is common to many Epyx releases: they have an impressive fast loader routine that requires a slight modification to the sector headers. A fast copier will ignore these eccentricities but a nybbler can reproduce them well enough to fool the fast loader. Obviously, this is not where the protection lies.

Load the nybbled copy of SB and observe what happens: when the "Maxx-OUT" screen appears, the disk drive hangs. If you listen closely to the drive when this happens, you'll hear the drive head move a long way across the disk before it gets spindizzy. This, then, is where the protection check occurs.

Load the $COOO monitor and the SB boot file1 which resides from $02A7 - $0303. The program start address can be found in the BASIC warm start vector at $0302 - $0303. The entry point is $02C1. This routine does little more than load the only other file in the directory "(C) 1987 EPYX" and then jumps to $7F06. This file resides from $7D09 to $7F73. Nost of this routine is the fast loader initialization code and drive-to-computer transfer routines.

At $7D2C, you can see the text for the Block-Execute (B-E) command that starts up the drive code on track/sector (TIS) 18/6 ($12/$06).

The drive code is interesting to study (see "L. A. Crackdown" elsewhere in this manual for all the gory details) but, if there's an easier way, why bother?

Begin disassembly at the entry point of $7F06. You should be looking at a short routine that ends with a JMP to $67E9 at $7F24. Examine the other subroutine calls to $7EFl and $7EF4. These are the initialization routines referred to above. A logical place to stop the loading process is the JMP $67E9. Change this instruction so that it JMP's to itself (JMP $7F24). Execute the code at $7F06 (G 7F06).The program should freeze up. Press your reset button and load the $COOO monitor.

Disassemble the code at $67E9. The subroutine call to $6909 reveals several calls to the load routines we saw earlier, followed by a comparison to a byte value at $6925. If the byte doesn't match, the code branches to $692E, where it executes an undocumented opcode ($02) that sends the computer into an infinite loop. What would happen if we just bypassed this code altogether?

Again, we'll have to patch the backup disk.

But where is this code? Try to find it with the Byte Pattern Searcher. Good luck! Epyx' fast load routine requires the disk data to be written a special way that Commodore DOS doesn't understand. But we CAN patch the code after it's loaded. The best place is at the end of "(C) 1987 EPYX" file, which ends at $7F73. Use the drivemon to load the last sector of the "(C) 1987 EPYX" file (T/S 18/5 or $12/$05). Change the JMP $67E9 at position $23 to read: JMP $7F73 ($4C $73 $7F). See the Rad Warrior section elsewhere in this manual for details on the use of the drivemon for this purpose.

Then add the following at position $72:

```
   LDA #$60       ; An "RTS"
   STA $6909      ;is placed at top of
   JMP $67E9      ;of protection check
                  ;and then JMP
```

You also must alter the last-byte pointer at position 1 in the sector to reflect our added code (from $72 to $7A) so that it loads properly. Write the sector back to the nybbled backup and boot it.

It worked! The protection check is bypassed.

< < < RAINBIRD : TRACKER > > >

Examination and analysis of the protection code in "Tracker" (TK) is a frustrating process: there are many, MANY code transfer and decryption routines. It is very easy to get lost and eventually one gets tired of tracing this nonsense. There must be an easier way.

There is. But first, make a FAST COPY of your original TK and then boot it several times in a row so you are familiar with the sequence of events that occur during the load. It's especially important to listen carefully to the drive while the program is loading so you get the "feel" or sense of rhythm of the loading process. Timing~is critical to discovering the protection check.

Let's examine the loading process. The auto-boot routine blanks the screen, there is some disk activity1 then nothing for about 5 seconds. The title screen appears and the load continues. After about 45 seconds the screen again blanks and the drive shuts off. Thirty seconds later the drive activates and you can hear the drive head swing a long distance across the disk and back again. If you are loading from the original disk, the first game screen will appear. Otherwise, a backup copy will produce garbage. So we can, for now, assume that the protection check occurred sometime during that long head swing.

The next step is to find the protection code. Repeat the loading process and wait for the long head swing we discussed above. When it starts to move back, hit your reset button. Load the $8000 monitor and start searching for drive command text (B-E, M-W, M-E, etc...). Often, these drive command strings are stored in memory in reverse, so keep trying. You should find a reversed 'M-W' and 'M-E' stored respectively at $09A6 and $O9AB. These commands write to and execute code at $0300 in the drive. Disassemble the code at $0900. Careful study will reveal what the drive is being told to do. First, the drive routine at $9OAE is sent to $0300 in the drive by a Memory-Write. Then, the routine is Memory-Executed after sending 3 additional bytes: $80, $28, and $OE. The drive routine stores these 3 bytes into job queue $01, producing a read ($80) of track 40 ($28) Isector 14 ($OE) into drive memory $0400. The computer waits for this read to complete then stores the sector of data at $9600 - $96FF, not caring if the read was successful or not. It assumes all the needed data is in place and starts up the game.

Use the drive monitor and the original TK disk to look at this sector. Initialize the disk and place $28 and $OE into job queue $08 and $09. Then place $80 into $01. When the drive shuts off, check $01 for a successful read: if it contains a $01 then the job completed successfully (a backup should produce an error code ($02-$0A). Disassemble the data at $0400. This is the code the protection is trying to load at $9600 in the computer. A bad read attempt will not produce the correct data, therefore whatever is loaded into $9600 will be executed, whether its valid code or not. This results in a system crash.

To produce a copyable backup we must relocate this sector to a normal DOS track. We prefer to use directory sectors when possible.

Track/sector 18/6 ($12/06) is available so use the job queue to write our data to it. Insert your backup copy, initialize the drive and place a $12 into $08, $06 into $09 and $90 into $01. Our sector is now easily accessible - to us. The protection routine will still look for it on track 40. We must find a way to re-direct the sector read to our new location.

There might be a simpler way, however. The nature of the 1541 DOS is that a sector header error (which will occur with a backup copy of SG) will NOT corrupt the current contents of the drive buffer. That is, the data residing in the buffer will still be intact after a header error. If we can read our sector at the appropriate time, the protection check will not destroy the data, assuming it doesn't find a valid header in track 40. One way is to "wedge" ourselves into the drive code.

One of the first things the auto-boot routine does is to execute the custom loader routine in the drive. This code reads in a sector of data and transmits it to the computer. What if we modified the routine to read our sector at $12/$06 AFTER it has completed its other duties? This would leave the data in $0400 as described above and the protection check would be satisfied. Reboot TX and allow it to load until the drive motor turns off. Press the reset button and load in the $8000 monitor. Examine the auto-boot code at $0l0E. This routine outputs a block-execute command (backwards at $0191 - 'B-E 2 0 18 02') that starts up drive code located on T/S 18/2 ($12/$02).

Insert your backup copy of TK, initialize the drive and use the drivemon to load this sector into drive buffer $0300 using the job queue. Disassemble the code in the drive at $0300. This code, when executed, loads T/S $12/$12 (18/18) into drive buffer $0600 and decrypts it. Control is then passed back to the computer, where a memory-execute (M-E) command of $0693 is sent to the drive. This initializes the drive side of the loader. To view the decrypted code at $0600, insert your backup copy of TK and do the following:

1) Use the job queue to read T/S $12/$12 into drive memory $0600 (T/S $12/$02 should already be present at $0300).

2) Assemble the following at $0400:

```
A 0400 JSR $0314
  0403 JMP $F969
```

3) Execute our routine at $0400 by placing the value $12 into drive memory $08 and $09, then place the value $EO (job queue execute command) into $01.

After a short period of drive activity, you may disassemble the decrypted code at $0600. The entry point of the loader is $0693, where some setup is done. Then a loop is executed to load and transmit each sector. After the load is completed, the code exits by JMP'ing to $D048, which re-initializes the drive. This is the ideal place for us to "wedge" ourselves into the loader. We can execute a job queue read of our sector at $12/$06 THEN jump to $D048. The drive code from $06E0 - $O6FF is filled with zeroes and is available for our use. Assemble the following code at $06E0:

```
A 06E0  LDA #$12
  06E2  STA $08
  06E4  LDA #$06
  06E6  STA $09
  06E8  LDA #$80
  O6EA  STA $01
  O6EC  LDA $01
  O6EE  BMI $O6EC
  06F0  JMP $D048
```

And the following at $06C4:

```
      A 06C4 JMP $06E0
```

This "patch" will load our sector into drive buffer $0400 and exit the same way as the original code.

Because the loader is encrypted we must also re-encrypt the code containing our patch. To do this, re-execute step #3 above. Rewrite the re-encrypted code at $0600 back to T/S $12/$12 by placing the value $90 into drive memory $03. When the drive LED turns off, reset the computer and try out your newly broken backup.

< < < RAINBIRD: STARGLIDER > > >

Examination and analysis of the protection code in "Starglider" (SG) is a frustrating process: there are many, MANY code transfer and decryption routines. It is very easy to get lost and eventually one gets tired of tracing this nonsense. There must be an easier way.

There is. But first, make a FAST COPY of your original SG and then boot it several times in a row so that you're familiar with the sequence of events that occur during the load. It's especially important to listen carefully to the drive while the program is loading so that you get the "feel" or sense of rhythm of the loading process. Timing is critical to discovering the protection check.

Let's examine the loading process. The auto-boot routine blanks the screen, there is some disk activity, then nothing for about 5 seconds. The title screen appears and. the load continues. After about 45 seconds the screen again blanks and the drive shuts off. A few seconds later, the drive activates and you can hear the drive head swing a long distance across the disk and back again. If you are loading from the original disk, the first game screen will appear. Otherwise, a backup copy will produce garbage. So for now, we can assume that the protection check occurred sometime during that long head swing.

The next step is to find the protection code. Repeat the loading process and wait for the long head swing we discussed above. When it starts to move back, hit your reset button. Load the $1000 monitor and start searching for drive command text (B-E, M-W, M-E, etc...). Often, these drive command strings are stored in memory in reverse, so keep trying. You should find a reversed 'M-W' and 'M-E' stored respectively at $90A6 and $9OAB. These commands write to and execute code at $0300 in the drive. Disassemble the code at $9000. Careful study will reveal what the drive is being told to do.

First, the drive routine at $9OAE is sent to $0300 in the drive by a Memory-Write. Then, the routine is Memory-Executed after sending 3 additional bytes: $80, $28, and $OE. The drive routine stores these 3 bytes into job queue $01, producing a read ($80) of track 40 ($28)/sector 14 ($OE) into drive memory $0400. The computer waits for this read to be completed6 then stores the sector of data at $4200 - $42FF, not caring if the read was successful or not. It assumes all the needed data is in place and starts up the game.

Use the drive monitor and the original SG disk to look at this sector. Initialize the disk and place $28 and $0E into job queue $08 and $09. Then place $80 into $01. When the drive shuts off, check $01 for a successful read: if it contains a $01 then the job completed successfully (a backup should produce an error code ($02 - $OA). Disassemble the data at $0400. This is the code the protection is trying to load at $4200 in the computer. A bad read attempt will not produce the correct data, therefore whatever is loaded into $4200 will be executed, whether it's valid code or not.

This results in a system crash.

To produce a copyable backup, we must relocate this sector to a normal DOS track. We prefer to use directory sectors when possible.

Track/sector 18/6 ($12/06) is available, so use the job queue to write our data to it. Insert your backup copy, initialize the drive, and place $12 into $08, $06 into $09 and $90 into $01. Our sector is now easily accessible - to us. The protection routine will still look for it on track 40. We must find a way to re-direct the sector read to our new location.

There might be a simpler way, however. The nature of the 1541 DOS is that a sector header error (which will occur with a backup copy of SG) will NOT corrupt the current contents of the drive buffer. That is, the data residing in the buffer will still be intact after a header error. If we can read our sector at the appropriate time, the protection check will not destroy the data, assuming it doesn't find a valid header in track 40. One way is to "wedge" ourselves into the drive code.

One of the first things the auto-boot routine does is to execute the custom loader routine in the drive. This code reads in a sector of data and transmits it to the computer. What if we modified the routine to read our sector at $12/$06 AFTER it has completed its other duties? This would leave the data in $0400 as described above and the protection check would be satisfied. Reboot SG and allow it to load until the drive motor turns off. Press the reset button and load in the $1000 monitor. Examine the auto-boot code at $OlOE. This routine outputs a block-execute command (backwards at $0191 - 'B-E 2 0 18 02') that starts up drive code located on T/S 18/2 ($12/02).

Insert your backup copy of SG, initialize the drive, and use the drivemon to load this sector into drive buffer $0300 using the job queue. Disassemble the code in the drive at $0300. This code, when executed, loads T/S $12/12 (18/18) into drive buffer $0600 and decrypts it. Control is then passed back to the computer, where a memory-execute (M-E) command of $0693 is sent to the drive. This initialize the drive side of the loader. To view the decrypted code at $600, insert your backup copy of SG and do the following:

1) Use the job queue to read T/S $12/$12 into drive memory $0600 (T/S $12/$02 should already be present at $0300).

2) Assemble the following at $0400:

```
A 0400 JSR $0314
  0403 JMP $F969
```

3) Execute our routine at $0400 by placing the value $12 into drive memory $08 and $09, then place the value $EO (job queue execute command) into $01.

After a short period of drive activity, you may disassemble the decrypted code at $0600. The entry point of the loader is $0693, where some setup is done. Then, a loop is executed to load and transmit each sector. After the load is completed, the code exits by JMP'ing to $D048, which re-initialize the drive. This is the ideal place for us to "wedge" ourselves into the loader. We can execute a job queue read of our sector at $12/$06, THEN jump to $D048. The drive code from $06E0 - $O6FF is filled with zeroes and is available for our use. Assemble the following code at $06E0:

```
A 06E0 LDA #$12
  06E2 STA $08
  06E4 LDA #$06
  06E6 STA $09
  06E8 LDA 1$80
  O6EA STA $01
  O6EC LDA $01
  O6EE BMI $O6EC
  06F0 JMP $D048
```

And the following at $06C4:

```
A 06C4 JMP $06E0
```

This "patch" will load our sector into drive buffer $0400 and exit the same way as the original code.

Because the loader is encrypted, we must also re-encrypt the code containing our patch. To do this, re-execute step #3 above.

Rewrite the re-encrypted code at $0600 back to T/S $12/$12 by placing the value $90 into drive memory $03. When the drive LED turns off, reset the computer and try out your newly broken backup.

< < < NicroLeague : WF wrestling > > >

"WWF Wrestling" uses a protection scheme that takes its sweet time before making the protection check, which leads you on until you're convinced that the backup you made is sound. Then, SURPRISE!, it fails. Fortunately, there are two ways to create a working backup of this piece. You can disable the protection check or you can use the included GCR EDITOR to reproduce the physical disk protection. Let's explore the protection check first.

Use any fast data copier to make a copy of your ORIGINAL WWF. Boot it and let it (oh so slowly) make its way towards the protection check, which occurs during the disk access preceding the actual beginning of the wrestling match. Reset the computer and load the $1000 monitor. Search memory for disk commands such as "M-E, B-E, Ul, etc...". You should find a "Ul" (read sector) and "B-E" (Block-Execute) command referencing track/sectors $12/$03 and $12/$04 (18/3 & 4).

Use the drivemon to load and disassemble these two sectors in drive buffers $0500 and $0600, respectively. You are now looking at the (fast?) loader drive code. If you're familiar with a normal read of GCR data, you'll notice something funny about the read routine in T/S $12/$04 at drive memory $0695. This code swings out to track 35, waits for a data block, and counts $144 bytes to the end of the data block, placing us in the tail gap (this is an effective protection technique because software-based nybblers will seldom copy tail-gap bytes).

Then the scheme looks for a GCR byte equal to the value of $73. If it's not found, the Y register is incremented and loops back to try again until Y is equal to $OA (10). If the $73 byte is found or .Y equals $OA, the current value of .Y is stored to $0300 in drive memory. The protection scheme is using this odd GCR byte ($73) to set a different byte to a certain value. We can break this protection check if we know the proper value of the Y register.

Load the included GCR editor and read track 35 of your ORIGINAL WWF diskette. Read in each data block and look for a $73 byte starting from position $144 on the GCR (left) side of the display. You should find the $73 byte on sector 0 at position $146. $146 minus $144 equals 2, giving us the value of the Y register. You can satisfy the protection check right here by reading this same sector on your backup copy, editing the data block so that it contains the $73 byte at position $146, and then writing the sector back to the backup copy. This duplicates the physical disk protection on the backup.

If you want to completely disable the protection check, reload drivemon and read track/sector $12/$04 (18/04) into drive buffer $0600. Enter the following:

```
   A O6DF CPY #$02   ;this was "CPY #$OA"
     06E1 BEQ $06E6  ;this was "BNE $06A2"
```

This "patch" will let the code execute normally but exit at the proper time with the correct value in .Y (2). Write the sector to your backup copy and you'll have a completely unprotected backup!

Note: This same patch will have to be applied to each WWF "Match" diskette because the drive code in track/sectors $12/$03 and $12/$04 is present on each of these releases - including SIDE 2 of the "Game" diskette.

< < < SOFTWARE TOOLWORKS : MAVIS BEACON > > >

"Mavis Beacon Teaches Typing" (MBTT) is another in a class of protection schemes that depends upon a sector of data located on a non-standard track. The mechanism is simple: critical data is placed on a track that is not used by standard DOS (36 - 40). A routine is called to read in the sector and transmit the data to the computer. Without this data the program will either crash or function improperly - sometimes in very subtle ways.

Before proceeding, use any good nybbler to copy side A, tracks 1 through 36 of an original copy of MBTT. Then use the provided File Logger to determine the start and end addresses of the files on MBTT Side A and error scan it to get an error map of the original. Try to boot your backup copy. It will fail, due to some subtle, deliberate alterations to track 36.

The error map shows us that valid sectors ARE present on track 36. The next step is to find the code that reads that track. Lets look at the auto-boot file. Load the $2000 monitor then insert your backup copy of MBTT and load "MAVIS". The file resides from $032C to $0400. The first 2 bytes are the KERNAL "close all files" vector (CLALL), which now contain $34 and $03. This is the entry point of the auto-booter ($0334). Analysis of this code at $0334 reveals that a series of Block-Reads are made of track 35 (the "Ul" command text is located at $03D6) then a JMP to $OFOO at $03C5 continues the loading process.

Change the code at $03C5 to JMP $2000 and execute the code at $0334 (G 0334). The screen will turn black, the disk drive will activate, and after a short time, control will return to the monitor. Disassemble the code at $OFOO. The routine from $OFOO - $0F22 copies the freshly-loaded code from $0C3C - $123B to $033C - $093B, then JMP's to $0623. This makes viewing the code in its proper location more difficult. By locating and executing the protection code in screen memory ($0400- $07F7), MBTT protects itself from a monitor like the one we are using. In addition, a normal reset of the computer will destroy ALL of this code. We can relocate it ourselves to a more convenient area ($733C) by using the monitor's (T)ransfer command:

```
   T OC3C 123B 733C
```

When disassembling this relocated code, remember to add $7000 to all address references in the program and the following text. The entry point here is at $0623 ($7623 - remember: add $7000).The routine at $0633 copies the drive fast loader code to $5000 - $52FF, then calls the subroutine at $0342 to send it to the drive, execute it, and change the KERNAL LOAD vector to point to the fast loader. The next step at $064F is the key to the protection scheme: what appears to be a normal load routine is actually reading the protected sector into $0C00. The KERNAL SETNAM call at $0654 is pointing to a rather odd file name consisting of 4 hex bytes at $0690 with the values $01 $24 $10 $01. Hex 24 ($24) 36 decimal and $10   16. Track/sector (T/S) 36/16 is the sector containing the protected data! The data is then decrypted and moved to $C002, where it is executed to continue the loading process.

The easiest way past a protection scheme like this is to capture the data ourselves, write it to a safe place on our backup copy, and change the protection code to look at our new location. This will be especially easy because the code is not encrypted. To do this, enter the drivemon, insert an ORIGINAL NBTT, and initialize the drive. Use the drive's job queue to read in T/S $24/$l0 (our protected sector) and write it to your backup copy. An unused directory sector is usually a good bet, so we'll use T/S $12/$12 (18/18).

The last step is to change the reference to the original protected sector to our newly relocated sector. Recall that the code we've been analyzing was loaded from track 35. Use the provided Byte Pattern Scanner to search for the 4 hex bytes ($01, $24, $10, $01) that we discussed earlier. Enter 35 for the starting AND ending tracks. The scanner should report the bytes' location on T/S 35/14 ($23/$OE) at position $54 (84). Use any sector editor or the drivemon to change the 2 bytes at position $55 on T/S 35/14 ($23/$OE) from $24/$l0 to $12/$12 and rewrite them to your backup copy. Now the protection scheme will look for our relocated data on track/sector 18/18 ($12/$12), load it in, and continue on its merry way. After you copy sides B through D of MBTT using any nybbler (tracks 1 through 35) you'll have a fully functional, unprotected backup of your valued typing tutor.

< < < CAPCON : 1942 V2 & GHOSTS & GOBLINS > > >

It was quite a surprise when CAPCOM released these titles using a protection scheme other than RapidLok. This scheme is as different from RapidLok as it is easy to trace and defeat. Please note that both programs are identical in their protection check routines except as noted.

You will need the following:

1) An original "1942" or "Ghosts & Goblins" diskette.

2) A backup copy of both sides of 1942 or Ghosts & Goblins using our "C-64 Fast Copier".
3) A disk log of the 1942 or Ghosts & Goblins disk to get the load addresses.

One thing is obvious when you boot the copy of these programs: they check protection immediately! Load the $8000 Kracker-Mon then the boot file "1942 or GHOSTS & GOBLINS". They load from $O2BB-$0305. The BASIC cold/warm start vectors at $0300/$0302 show the entry point to be $02D6. The boot file loads the "1.0" file, then jumps to $CCOO.

Load "1.0", which resides from $C900 - $DOOO. The entry point at $CCOO JMPs to $CC7l, which calls a subroutine at $C900. This subroutine sends protection check code to the drive. If you look at memory in the range $CAOO - $CAFF, you will see numerous BACKWARDS "Memory-Write" (M-W) commands. The drive code is at $CA8D. This code looks for some special bytes on the disk and stores them in drive memory. When its finished, the computer Memory-Reads them into memory and stores them.

Disassemble the code from $C900 and keep scrolling down to $C9EO. This is the M-R routine. At $C9EC, it reads in 3 bytes and over-writes them into $CA87 - $CA89. It checks $CA87 for a zero value. If it's zero, the protection fails. If not, it reads 2 more bytes into $CA8A - $CA8B. At $CA36, the weak point in this protection scheme becomes readily apparent. It checks the 5 bytes from the drive for specific values. It even shows us the values! If all the values are correct, it stores an $FF at $CFFF. Lets store the $FF ourselves and see what happens. Change the code at $CA3l to read:

```
   A CA3l LDA #$FF
     CA33 STA $CFFF
     CA36 RTS
```

Now execute the loader (G CCOO). It should load. In fact, if you return to the subroutine call at $CC71, you can see where it checks the value of $CFFF. If it doesn't match, it goes into an endless loop. You could change the JMP $CC7l at $CCOO to JMP $CC79 for a one-byte break ($71 = $79)! Use the File Tracer utility to make any of these changes to your backup copy for a completely un-protected backup.

< < < EPYX : L.A. CRACKDOWN > > >

"L.A. Crackdown" represents state of the art disk protection caught with its pants down. It is uncopyable with software nybblers, but it CAN be had with a little persistence and ingenuity.

You will need the following:

1) An original "L.A. Crackdown" (LAC) diskette.

2) A backup copy of LAC using "C-64 Fast Copy".

3) A formatted blank work disk.

4) A printout or the results of an error-scan of both sides of the original diskette.

Examining the disk maps show that side 2 is completely normal, but tracks 1 - 5 and part of track 18 on side 1 are unreadable by normal methods. A directory shows only 2 short files with 432 blocks free on the diskette. We know from our error-scan that there are very few unused sectors on side 1. So where is the program coming from? Use the file tracer to determine the files' beginning and ending addresses. Boot MONlOOO, and let's examine these 2 files. The first file loads at $02A7 - $0304. Disassembly shows that it does nothing more than load the second file, followed by a JMP to $CAOO.

Load the "(C) 1988 EPYX" file. It resides from $C74F - $CA19. Disassemble from $CAOO, which is the entry point. The first few instructions do some initialization of the system, followed by 2

JSR's and then a JMP to $4000. Look at the code in the first subroutine at $C9Fl. Careful tracing will reveal that this routine boots the fast loader code in the drive by issuing a 'Block-Execute' command to the drive. The command string is located at $C955 and the drive code is stored on track/sector (T/S) 18/6 ($12/$06). We'll look at that in a moment. The second subroutine is the computer side of the loader that communicates with the drive and retrieves the data. After the load has completed, the JMP to $4000 is executed.

Let's stop the program after the load. Replace the JMP to $4000 with JMP $CAl6. This creates an endless loop that we can interrupt with RUN/STOP-RESTORE. Then, fill memory from $4000 - $BFFF with an oddball value (I use $99). Make sure the ORIGINAL LAC disk is in the drive and then execute the code at $CAOO. The screen should blank, followed by a flurry of disk activity. When the screen re-appears (full of garbage) press RUN/STOP-RESTORE and re-enter the monitor (5Y54096). Switch in the RAM underneath BASIC (place a $36 at location $02 if you are using Kracker-mon) and look for the start of your filler bytes. You should find them at $A900. The data loaded from $4000 to $A8FF.

If you try to execute the code at $4000, the computer will lock up. Why? Because the fast loader in the drive is still running and it polls the serial bus constantly, waiting for the next load command. Only a complete reset of the drive will re-establish communication. What we must do is start up the drive code before executing the code at $4000. Recall that the routine at $C9F1 was the routine that activated the drive code. Turn the drive off for three seconds, then back on. Place a JSR $C9Fl at $3FFD and save the code from $4000 - $A900 to your work disk. Re-insert the ORIGINAL LAC diskette and again load the "(C) 1988 EPYX" file, then execute the code at $3FFD. If the title screen appears after a moment, you've done everything right. The code from $4000 - $A900 CAN be saved from memory, reloaded and started back up if the "(C) 1988 EPYX" file is also loaded.

Now let's look at the drive code on T/S 18/6 ($12/$06). Reload "MONl000", insert the ORIGINAL LAC, and initialize the drive. Use the drive monitor to load the sector into drive buffer $02 ($0500 in drive memory) so we can disassemble it. Please refer to the Rad Warrior section elsewhere in this manual. The $0500 buffer is accessed at drive locations $OA (Track) and $OB (Sector). Use location $02 to execute the command byte $80. The code from $0500 - $051F is a decryption routine. It then JMP's to $0160. If we let it JMP, we will lose control of the drive to the fast loader. To view the decrypted code at $0160, place a 'JMP $F969' (job completed) at $0522 and $EO (execute) in drive job queue $02. After the drive motor shuts down, disassemble the code at $0160. This routine reads and decrypts the drive code located in the protected sectors on track 18. How are we going to trap that drive code so we can use it on an un-protected disk?

Clearly, we must let the routine continue and interrupt it at the right moment. Study the code. The protected drive code is stored from $0300 through $O6FF by the routine. At $O1AD, a JSR $O3BE is executed. Since this is the first call made to the newly loaded drive code, this seems a good place to stop it. Again, place a 'JMP $F969' at $OlAD. To continue execution of the code, place a 'JMP $0160' at $0500 and place $EO in drive job queue $02. After the drive motor shuts down, disassemble the code at $0300 - $O6FF.

Now we need to save it. Insert your backup copy and initialize the disk (@I). The error-scan shows that there are several unused directory sectors on side 1 so we can safely save our newly-captured code to these - we'll use sectors 15 - 18 ($OF - $12). Using the drivemon, place the following bytes into job queue $06 - $OD: 12 OF 12 10 12 11 12 12. Then place $90 (write job) into job queue $00, $01, $02, and $03. Wait until the drive motor shuts off. The needed drive code is now stored on your backup disk.

The next step is to trap and save the decrypted code on T/S $12/$06 and write a short routine to load up our four drive code sectors. Again, read T/S $12/$06 into drive memory $0500 and place "JMP $F969" at $0522. Place $EO in drive job queue $02 to decrypt the code. Transfer the decrypted code from $0160 - $OlFF to $0560. Our new start-up routine at $0500 will load the four drive code sectors using the DOS job queue. Use the assembly capability of the monitor to enter the following into drive memory:

```
     ]A      0500: SEI
     ],      0501: LDX 10        ;move code to a safe place
     ],      0503: LDA $0500~X
     ],      0506: STA $0700,X
     ],      0509: INX
     ],      050A: BNE $0503
     ],      OSOC: JNP $070F     ;continue execution
```

Transfer the code from $0500 - $05FF to $0700. Continue entering code at $070F:

```
     ]A      070F: LDX 1$OD      ;load up the job queue with T/S
     ],      0711: LDA $0740,X   ;numbers and read commands ($80)
     ],      0714: STA $00,X
     ],      0716: DEX
     ],      0717: BPL $0711
     ],      0719: LDX #$03      ;wait until all sectors have
     ],      071B: LDA $oo,x     ;been loaded
     ],      071D: BMI $071B
     ],      071F: DEX
     ], 0720: BPL $071B
     ], 0722: SEI                ;move code at $0760 to $0160
     ], 0723: LDX #$60
     ], 0725: LDA $0700,X
     ], 0728: STA $0100,X
     ], 072B: INX
     ], 072C: BNE $0725
     ], 072E: JMP $OlAD          ;fire up the fast loader

     3:0740 80 80 80 80 00 00 12 OF    ;DOS job queue data
     3:0748 12 10 12 11 12 12
```

Transfer the code at $0700 - $O7FF back to $0500. Write it to the backup disk by placing a $90 into drive job queue at $02.

The last steps involve modifying the BAM of the backup disk so you can copy the $4000 file on your work disk to the backup. You must then alter the auto-boot to load both the $4000 file and "(C) 1988 EPYX", start up the drive code (JSR $CF91) and JMP to the entry point ($4000). The $4000 file should be 106 blocks long. Curiously enough, the tracks now available, 1 - 5 (5 * 21 = 105), plus the one unused sector on T/S $ll/$OC, totals 106 blocks!

Load the BAM into drive memory $0500. Use the monitor to enter the following data:

```
     ]:0504 15 FF FF lF 15 FF FF lF
     ]:050C 15 FF FF lF 15 FF FF lF
     ]:0514 15 FF FF lF
```

This makes tracks 1 - 5 available. Now till $0518 - $058F with $00 to allocate the rest of the available sectors. To free-up the sector at $l1/$OC enter:

```
     ]:0544 Ol 00 10 00
```

Place $90 into job queue $02 to write the BAM back to your backup. Initialize the diskette (@I) and view the directory (@$). It should show 106 blocks free.

Modifying the auto-boot file to load our $4000 file presents a problem because it resides in the directory (T/S $12/$02). Re-saving the file will use the first available sector: namely, our much needed block at $l1/$OC. What we CAN do, after modifying the auto-boot, is use the drive monitor to place the auto-boot code on to $12/$02. Return to the computer monitor and load the "L.A. CRACKDOWN" file. Enter the following commands and code:

```
   T 02D1 O2EC 02A7  ;copy load routine to $02A7
   A O2ED JMP $02A7   ;change JMP $4000 to our new code
   : 0302 CC 02       ;new entry point for auto-boot($02CC)
   A 02C3 JSR $C9F1   ;fire up the drive code
     02C6 JMP $4000   ;continue execution
   : 02F0 4C 41       ;our new file name ("LA")
   A 02B2 LDX #$FO    ;point load to our new file name
```

Together,these changes will load the "(C) 1988 EPYX" file and our new "LA" file, activate the drive code, and start-up the program. Now we must copy the routine over the original. Enter the drive monitor and load T/S $12/$02 into drive memory $0500. Copy our new code into the drive by entering:

```
   TC 02A7 0300 0504
```

Re-write the modified sector to the backup diskette. Return to computer monitor and insert the work disk containing our $4000 file and load it. Switch out BASIC (place a $36 at computer location $02 when using Kracker-Mon), insert your LAC backup copy, and save the file, naming it "LA". A directory of the diskette should show 0 blocks free. Your backup copy is now completed.

< < < V-MAX! v1.? > > >

When V-MAX! first appeared on the copy protection scene, one could stay up late at night and almost hear the endless nocturnal muttering from every protection removal expert in the country. With two, and sometimes three levels of physical disk protection, here was a formidable foe, indeed! We have identified two major versions of V-MAX!. Information on the last modifications of V1 is included in this Tutorial.

LEVEL 1:
Protection level 1 is the method of storage of the custom fast loader code on a V-MAX! formatted disk. On the master disk, the drive code is pre-processed by submitting each byte of the drive code to a routine that generates two GCR bytes for each drive code hex byte. This is then attached to a series of carefully chosen bytes and written to a track (usually track 20) on the master disk in one disk revolution. The only way to reproduce this track is with a hardware-based copier.

LEVEL 2:
V-MAX! uses only two density levels in its disk format. Instead of the two normal density levels used for tracks 25 - 40, the density level for tracks 18 - 24 is substituted. To copy the disk properly (excluding track 20), you must use a copier capable of detecting and reproducing these abnoruial densities. The quality of the copy is very important and should be made on a drive that is in excellent condition. Correct drive speed is of the utmost importance!

LEVEL 3:
Some V-MAX! titles require ninor changes to a sector or two to disable a third level of protection that looks for a hard-to-copy byte sequence on a track. Finding these little routines is actually the hardest part of making a backup copy of a V-MAX! protected program. If you don't have a modified 1541 DOS KERNAL that can trap this protection code (it executes in the command buffer at $0200), you have little hope of finding and breaking these routines. Because of this, we'll have to give you these modifications without further explanation.

The following pages contain specific instructions for making functional backups of three V-MAX!ed titles: Xevious, Into The Eagle's Nest, and Paperboy.

* V-MAX! V2 is a whole new ball game, and requires 8K of drive RAM to duplicate. Special copier routines must be written for these protection schemes. Also, for your information, we have spoken to several software publishers about V-MAX!, and their programs using it. They claim that V-MAX! is NOT a protection scheme, but a fast loader system only. We are skeptical.

< < < MINDSCAPE: INTO THE EAGLE'S NEST > > >

The entire protection removal process will take place in the drive. We are going to let the protected code on track 20 load into the drive and then re-write it to some empty directory sectors. We will then modify the code that reads track 20 50 that it instead loads our newly-filled sectors.

Prepare a work copy of EAGLE'S NEST using the MAX Copier on your utility disk, and then load the $1000 monitor. Insert your original EAGLE'S NEST, initialize the drive (~I) and enter the drive-mon. We'll be using the 1541/71's job queue to do a lot of the work for us.

Read TIS $12/$OD into buffer $0700 by entering:

```
   :OOOE 12 OD
   :0004 80
```

Disassemble the code at $0700 (D 0700). The first thing the code does is move the drive read/write head forward two tracks to track 20. It then initializes a set of pointers to start the load process at buffer $0300 and starts reading bytes from the drive. There are no sync marks on the track: the routine reads until it finds a GCR byte with the value $5A, of which there is a long series. When the $5A byte sequence ends, the code reads and EOR's each successive pair of bytes together and stores the result byte to buffers $0300 - $O6FF. This produces the custom fast loader code.

You can now understand how a normal nybbler is dead in the water if it can't reproduce this track. But we can trap the code easily. Bypass the JMP instruction at $0797 by entering:

```
   ]A 078E LDA #$0l
   ], 0790 JMP $F969
```

This will return control to the drive-mon when the code has finished execution. To execute it, enter

```
   :0004 EQ
```

When the monitor returns you will be able to look at V-MAX! in all its glory. We first must make a minor modification to the code in case your work copy is not perfect. There is a sector checksum verification routine at $03F3 that will fail if the sector checksum is not zero. This can be defeated by entering:

```
   :03F5 A9 00
```

Now we need to re-write the loader code at $0700. Start with a fresh copy by re-loading T/S $12/$OD like we did above. Directory sectors $04, $07, $OA, and $OC will contain the code from $0300 - $O6FF. Beginning at $0700, re-write the drive code as follows:

```
  0700    SEI         ;disable interrupts
  0701    LDX #$OD
  0703    LDA $071A,X ;store read data to job queue
  0706    STA $00,X
  0708    DEX
  0709    BPL $0703
  070B    CLI
  070C    LDA $00     ;wait for read to complete
  070E    ORA $01
  0710    ORA $02
  0712    ORA $03
  0714    BMI $070C
  0716    SEI         ;continue normally ...
  0717    JMP $078E

  071A 80 80 80 80 00 00 12 04
  0722 12 07 12 OA 12 OC
```

Make sure all of the original code from $078E - $O7FF is left undisturbed.

Now write all the code to the work copy by inserting your work copy into the drive and entering:

```
   ]:0006 12 04 12 07 12 OA 12 OC
   ]:OOOE 12 OD
   ]:0000 90 90 90 90 90
```

There is the third level protection present on this title. To remove it, enter the following:

```
   ]:0006 18 OD
   ]:0000 80
   ]:0362 6B
   ]:036E 45
   ]:0000 90
```

That's all there is to it! Enjoy your backup copy.

< < < NINDSCAPE: PAPERBOY > > >

The entire protection removal process will take place in the drive. We are going to let the protected code on track 20 load into the drive and then re-write it to some empty directory sectors. We will then modify the code that reads track 20 so that it instead loads our newly-filled sectors.

Prepare a work copy of Paperboy using the MAX Copier on your utility disk, and then load the $1000 monitor. Insert your original Paperboy, initialize the drive (@I) and enter the drive-mon. We'll be using the 1541/71's job queue to do a lot of the work for us.

Read T/S $12/$OD into buffer $0700 by entering:

```
   :OOOE 12 OD
   :0004 80
```

Disassemble the code at $0700 (D 0700). The first thing the code does is move the drive read/write head forward two tracks to track 20. It then initializes a set of pointers to start the load process at buffer $0300 and starts reading bytes from the drive. There are no sync marks on the track: the routine reads until it finds a GCR byte with the value $5A, of which there is a long series. When the $5A byte sequence ends, the code reads and EOR's each successive pair of bytes together and stores the result byte to buffers $0300 - $O6FF. This produces the custom fast loader code.

You can now understand how a normal nybbler is dead in the water if it can't reproduce this track. But we can trap the code easily. Bypass the JMP instruction at $0797 by entering:

```
   ]A 078E LDA #$01
   ], 0790 JMP $F969
```

This will return control to the drive-mon when the code has finished execution. To execute it, enter:

```
   :0004 EO
```

When the monitor returns you will be able to look at V-MAX! in all its glory. We first must make a minor modification to the code in case your work copy is not perfect. There is a sector checksum verification routine at $03F3 that will fail if the sector checksum is not zero. This can be defeated by entering:

```
   :03F5 A9 00
```

Now we need to re-write the loader code at $0700. Start with a fresh copy by re-loading T/S $12/$OD like we did above.

Directory sectors $04, $07, $OA, and $OC will contain the code from $0300 - $O6FF. Beginning at $0700, re-write the drive code as follows:

```
  0700    SEI         ;disable interrupts
  0701    LDX #$OD
  0703    LDA $071A,X ;store read data to job queue
  0706    STA $00,X
  0708    DEX
  0709    BPL $0703
  070B    CLI
  070C    LDA $00     ;wait for read to complete
  070E    ORA $01
  0710    ORA $02
  0712    ORA $03
  0714    BMI $070c
  0716    SEI         ;continue normally
  0717 JMP $078E

  071A 80 80 80 80 00 00 12 04
  0722 12 07 12 OA 12 OC
```

Make sure all of the original code from $078E - $O7FF is left undisturbed.

Now write all the code to the work copy by inserting your work copy into the drive and entering:

```
   ]:0006 12 04 12 07 12 OA 12 OC
   ]:OOOE 12 OD
   ]:0000 90 90 90 90 90
```

There is the third level protection present on this title. To remove it enter the following:

```
   ]:0006 19 Ol
   ]:0000 80
   ]:035C 60
   ]:0368 6F
   ]:0000 90
```

That's all there is to it! Enjoy your backup copy.

< < < HINDSCAPE : XEVIOUS > > >

The entire protection removal process will take place in the drive. We are going to let the protected code on track 20 load into the drive and then re-write it to some empty directory sectors. We will then modify the code that reads track 20 50 that it instead loads our newly-filled sectors.

Prepare a work copy of Xevious using the MAX Copier on your utility disk, and then load the $1000 monitor. Insert your original Xevious, initialize the drive (@I) and enter the drive-mon. We'll be using the 1541/71's job queue to do a lot of the work for us.

Read T/S $12/$OD into buffer $0700 by entering:

```
   :OOOE 12 OD
   :0004 80
```

Disassemble the code at $0700 (D 0700). The first thing the code does is move the drive read/write head forward two tracks to track 20.

It then initializes a set of pointers to start the load process at buffer $0300 and starts reading bytes from the drive. There are no sync marks on the track: the routine reads until it finds a GCR byte with the value $5A, of which there is a long series. When the $5A byte sequence ends1 the code reads and EOR's each successive pair of bytes together and stores the result byte to buffers $0300 - $06FF. This produces the custom fast loader code.

You can now understand how a normal nybbler is dead in the water if it can't reproduce this track. But we can trap the code easily.

Bypass the JMP instruction at $0797 by entering:

```
  ]A 078E LDA #501
  ], 0790 JMP $F969
```

This will return control to the drive-mon when the code has finished execution. To execute it, enter

```
   :0004 EO
```

When the monitor returns you will be able to look at V-MAX! in all its glory. We first must make a minor modification to the code in case your work copy is not perfect. There is a sector checksum verification routine at $03F3 that will fail if the sector checksum is not zero. This can be defeated by entering:

```
   :03F5 A9 00
```

Now we need to re-write the loader code at $0700. Start with a fresh copy by re-loading T/S $12/$OD like we did above.

Directory sectors $04, $07, $OA, and $OC will contain the code from $0300 - $O6FF. Beginning at $0700, re-write the drive code as follows:

```
  0700    SEI         ;disable interrupts
  0701    LDX #$OD
  0703    LDA $071A,X ;store read data to job queue
  0706    STA $00;X
  0708    DEX
  0709    BPL $0703
  070B    CLI
  070C    LDA $00     ;wait for read to complete
  070E    ORA $01
  0710    ORA $02
  0712    ORA $03
  0714    BMI $070C
  0716    SEI         ;continue normally ...
  0717    JMP $078E

  071A 80 80 80 80 OD 00 12 04
  0722 12 07 12 OA 12 OC
```

Make sure all of the original code from $078E - $O7FF is left undisturbed

Now write all the code to the work copy by inserting your work copy into the drive and entering:

```
   ]:0006 12 04 12 07 12 0A 12 0C
   ]:OOOE 12 0D
   ]:0000 90 90 90 90 90
```

That's all there is to it! Enjoy your backup copy.

< < < PROTECTION SCHEME #1 > > >

Protection scheme #1 is a simple routine that creates DOS error # 22: DATA BLOCK NOT FOUND. This is accomplished by reading a sector on disk, changing the default data block ID (normally $07) in drive memory $0047 to a new value, then rewriting the data block using the new data block ID. Please note that any good nybbler can reproduce this protection type.

There are two simple ways for a programmer to use this type of copy protection. One way is to create the error, and check that the error is present at that sector. The other method is to create the error in a sector that contains data imperative to the operation of the program. Only a specialized routine can read in the data if the error is present. If the error isn't present, the routine written to pull the sector will not operate correctly and the data will be left behind. Let's start with this type.

22 Error - Data Recovery

The new data block ID is a GCR value whose high bit (bit 7) must equal zero; therefore, the new ID can have one of the following range of values:

Dec	Hex
0 - 7	$00 - $07
9 - 31	$09 - $lF
64 - 95	$40 - $5F
112 - 127	$70 - $7F
192 - 207	$C0 - $CF

Any attempt to read a sector with a non-standard data block ID will fail unless the default value in drive memory $0047 is changed to the new data block ID value.

Use the included BASIC program "DBWRITE" to rewrite the desired sector(s) with a new data block ID (creating the 22 Error). "DBREAD" can then be used to read the protected sector(s) and place it in drive memory at $0300 where it can be accessed with a "Memory-Read" command. From there you can either transfer the code down to the computer or leave it in the driveg if it's drive code. You may use the included assembly code in a machine language program if you wish.

```
DBWRITE.ASM

  This program is for educational and personal use only
  No commercial use of this program is permitted.
  All rights reserved  (C) 1989 K.J.P.B.

************************************************************;
Job:
  Rewrite a data block with a different
  data block ID code. High nibble of code
  must be $Ox, $lx, $4x, $5x, $8x or $Cx;
  (x = any hex number from $0- $F)
  The following code must be written to
  drive memory $0500 and can be executed
  from BASIC with the following statement:

  OPEN 15,8,l5,"UC:"+CHR$(new id code)+CHR$(trk)+CHR$(sec)
  CLOSE 15
************************************************************;

   org   $0500    ;code executes in drive here writdbid
   sei            ;disable interrupts
   lda  $47       ;save current data block id char
   sta  oldid
   ida  $203      ;get new id from command buffer
   sta  newid
   lda  $204      ;get track for new data block id
   sta  $06       ;will be read into $0300
   lda  $205      ;get sector for new data block id
   sta  $07
   ida  #$BO      ;seek track/sector
   jsr  waitjob
   ida  #$80      ;read track/sector into $0300
   jsr  waitjob
   lda  newid     ;setup new data block id
   sta  $47
   ida  #$90      ;write trise with new data block id
   jsr  waitjob
   pha            ;save error code ($01 = O.K.)
   lda  oldid     ;restore old data block id
   sta$47
   pla            ;get error code
   cli            ;enable interrupts
   rts            ;and exit
   waitjob
   sta$00         ;store job code to job queue
   cli            ;enable interrupts
   wjloop
   ida$00         ;wait for job to finish
   bmiwjloop
   sei            ;disable interrupts
   rts            ;return
   newid .hex  00 ;storage for new data block id
   oldid .hex  00 ;storage for old data block id

   end
```

DBREAD.ASM

```
  This program is for educational and personal use only
  No commercial use of this program is permitted.
  All rights reserved  (C) 1989 K.J.P.B.

***********************************************************;
Job:
  Read a data block with a different
  data block ID code.
  The following code must be written to
  drive memory $0500 and can be executed
  from BASIC with the following statement:
  OPEN 15,8,15,"UC:"+CHR$(new id code)+CHR$(trk)+CHR$(sec)
  CLOSE 15

  Data block can then be read from $0300 in drive memory.
***********************************************************;

   .org $0500     ;code executes in drive here readdbid
   sei            ;disable interrupts
   lda $47        ;save current data block id char
   sta oldid
   lda $203       ;get new id from command buffer
   sta newid
   lda $204       ;get track for new data block id
   sta $06        ;will be read into $0300
   lda $205       ;get sector for new data block id
   sta $07
   lda newid      ;setup new data block id
   sta $47
   lda #$80       ;read track/sector into $0300
   sta $00        ;store job code to job queue
   cli            ;enable interrupts
   wjloop
   lda $00        ;wait for job to finish
   bmi wjloop
   ldx oldid      ;restore old data block id
   stx $47
   rts            ;and exit
   newid .hex  00 ;storage fo? new data block id
   oldid .hex  00 ;storage for old data block id

   end
```

22 ERROR - ERROR PRESENT CHECK

You can make a simpler protection check by using DBWRITE to create a DOS 22 error, and then do nothing more than check the drive error channel for the proper error code. The BASIC code would read as follows:

```
         10 REM: CHECK FOR 22 ERROR
         20 OPEN 15,8,15,"I":REM INITIALIZE DRIVE
         30 OPEN 2,8,2,"#": REM RESERVE BUFFER FOR SECTOR READ
         40 PRINT#15,"Ul:2 0 Ol 00":REM READ TRACK/SECTOR 1/0
         50 GET#l5,A$:REM READ ERROR CHANNEL:CLOSE 2:ClOSE 15
         60 IF A$="2" THEN PRINT "DATA BLOCK NOT FOUND!":END:REM
         PROTECTION PASSED
         70 PRINT "DATA BLOCK WAS FOUND":REM PROTECTION FAILED
```

A machine-language routine to do the same would read as follows:

```
         .org $cOOO

         lda #$00       ;open cmd channel
         jsr $ffbd      ;SETNAM
         lda #$of
         ldx #$08       ;to drive 8
         tay
         jsr $ffba      ; SETLFS
         jsr $ffco      : OPEN
         lda #$0l       ;open buffer channel
         ldx #<pound
         ldy #>pound
         jsr $ffbd
         lda #$02
         ldx #$08
         tay
         jsr $ffba
         jsr $ffco
         jsr $ffcc      ;clear channels CLRCHN
         ldx #$of       ;output "ul" command
         jsr $ffc9 ;CHKOUT
         ldy #$00
   loop  lda ulcmd,y
         jsr $ffd2 ;CHROUT
         iny
         cmp #$0d
         bne loop
         jsr $ffcc
         ldx #$0f  ;input error code
         jsr $ffc6 ;CHKIN
         jsr $ffcf ;CHRIN
         sta $fb   ;store first error code to 251
         jsr $ffcf
         sta $fc   ;store second error code to 252
   loop1 jsr $ffcf ;read until you receive a <RETURN> character
         cmp #$0d
         bne loop1
         jsr $ffe7 ;close all channels ;CLALL
         rts
   pound byt "#"
   ulcmd byt "ul: 2 0 0l 00"
         byt $0d
```

This MIL routine can be stored at $COOO (49152) and called from BASIC as follows.

```
   10 OPENl5,8,15,"I":CLOSEl5
   20 SYS49152
   30 IF PEEK(251)<>ASC("2") AND PEEK(252)<>ASC("2") THEN PRINT
    "DATA BLOCK NOT FOUND!"1:END:REM PROTECTION PASSED
   40   PRINT "DATA BLOCK WAS FOUND":REM PROTECTION FAILED
```

< < < Protection Scheme # 2 > > >

This protection scheme is guaranteed to defeat ANY non-hardware-assisted nybbler on the market; including Fast Hack'em and our very own set of comprehensive nybblers. The physical protection invclves placing a set of GCR bytes in the tail gap of a sector on disk. Drive memory limitations prevent a software-only nybbler from copying these bytes, which are located after the end of the GCR bytes that make up the sector on disk. Only extra drive RAM and software to support it can copy these bytes. To better illustrate this, let's look at a typical sector on disk.

Format a work disk, then load the GCR Editor (GCRKD) from the Hacker's Utility Kit. With your work disk in the drive, input 1 fQr the track number and press <RETURN> twice. The GCRED will display a summary of all the header/data blocks on the track. Both sides of the screen are showing you the same information in different ways. On the right, the (hex) bytes are displayed as they were before they were written to the disk. On the left are the converted (GCR) bytes as they actually appear when reading, or writing to, the track directly.

For every four hex bvtes there are five GCR bytes. Group Code Recording ensures that there are never more than eight consecutive "1" bits or two consecutive "0" bits written to the disk. This allows the drive to use ten consecutive "l" bits as a signal that a header or data block will be read starting with the first "0" bit read. This is referred to as a sync mark. A normal sync mark is forty consecutive "1" bits (five hex $FF bytes). This is a deliberate overkill to make the disk format as reliable as possible.

Using < CURSOR UP/DOWN >, you can highlight either a header block, whose first byte is GCR $52 or hex $08; or a data block - GCR $55/hex $07. Cursor down to the last data block. This is sector $14 (20) of track 1. Press [SPACE] to read the entire data block into memory. The GCRED will display an editing screen, again with GCR on the left and hex on the right. Pressing [S] (Side) will move the cursor from the left to the right side or visa-versa. We will onlv be working on the GCR side.

Above the sector data, POS shows you the position in the data block of your cursor. Use the cursor keys to place the cursor at position $0144. This is the last byte of the data block. This is where everv software-onlv nybbler stoos reading the data block. ANY GCR bytes written past this point are ignored by the copier. Many, MANY protection schemes depend on this fact when they create their phvsical disk protection. The logical protection involves a custom drive program to look past the end of the data block for the special bytes that have been placed there. A special routine is not needed to write the physical protection: the GCRED is fully capable of such chores.

Move the cursor to position $0145. Press [SPACE] to enter EDIT mode and type the following:

```
   AA AB AC AD AE 55 55
```

then press (RETURN) to exit EDIT mode, [W] to write the sector back to disk, and [R] to re-read the modified sector. Verify that the bytes $AA - $AE are present at positions $0145 - $0149 (ignore the two $55 bytes). If not, try entering and writing them again. We have just created the physical protection.

The next thing we concern ourselves with is the logical protection. We need a special drive routine to check for the bytes that we added to the end of the data block. Below is an assembler listing of such a routine. What you do with the bytes is up to you: you could use them as a key to decrypt some data necessary to the operation of your protected program or send the drive into an endless loop so that the program could proceed no further. We'll simply place the bytes in drive memory where they can be tested by your routine.

```
   TGREAD.ASM

     This program is for educational and personal use only
     No commercial use of this program is permitted.
     All rights reserved  (C) 1989 X.J.P.B.

   **********************************************************;
     JOB:    Read 5 tail-gap bytes from a given track and sector.
     The following code must be written to
     drive memory $0500 and can be executed
     from BASIC with the following statement:

     OPEN 15,8,15,"UC:"+CHR$(track)+CHR$(sector):CLOSE 15

     The tail-gap bytes can then be read from $0300 - $0304 in
     drive memory.

   **********************************************************;

   .org  $0500 ;code executes in drive here

   ;this routine sets up READTG for execution

  setup
    sei         ;disable interrupts
    lda  #$4c   ;set up for job queue EXEC command
    sta  $0300  ; (JMP READTG)
    lda  #readtg
    sta  $0301
    lda  #>readtg
    sta  $0302
    lda  $203   ;get track for tail-gap read
    sta  $06    ;will be read into $0300
    lda  $204   ;get sector for tail-gap read
                ;from command buffer.
    sta  $07
    lda  S$EO   ;store EXEC cmd to job queue
    sta  $00
    cli         ;enable interrupts
    wjloop
    lda  $00    ;wait for job to finish
    bmi wjloop
    rts         ;exit

;This is the actual read routine.
```

```
readtg
   sei
   jsr $f510      ;search for the header block of our   sector.
                  ;If not found, this subroutine will exit
```

```
                 ;and NOT return to us.
syncloop
   bit $lcOO     ;The header block was found so wait
                 ;sync Bark preceding the data block
   bpl syncloop
sloop1
   bit $lcOO     ;got a sync, now wait for it to end
   bmi sloop1
   lda $1cO1     ;throw away the sync image
   clv
   ldx #$01      ;set up .x/.y to count $0145 bytes
   ldy #$45      ;($000O -$0144) to the end of the data
                 ;block
dataloop
   bvc *         ;wait for data byte ready
   clv           ;clear ready flag
   lda $1cO1     ;read byte from diskette $0144 times
   dey
   bne dataloop  ;
   dex
   bpl dataloop  ;
dloop1
   bvc *         ;we're now at position $0145 -
   clv           ;in the TAIL GAP
   lda $lcOl     ;read our 5 bytes
   sta $0300,y   ;and store them from $0300 - $0304
   iny
   cpy #$05
   bne dloop1
   jsr $f98f     ;turn off drive motor
   lda #$01      ;O.K.
   sta $00       ;and exit back to SETUP
   cli
   rts
end
```

A sample BASIC program named "TGREAD" is included on disk that sends the above code (stored in data statements) to the drive, executes it, and displays whether the protection passed or failed.

THE HACKER'S UTILITY KIT

programmed by:
Mike Howard I Joe Peter
Paul Rowe I Jeff Spangenberg
Designed by: Les Lawrence
(C)1987 K.J.P.B.

Welcome to The Hacker's Utility Kit. This program represents the finest set of disk examination and manipulation tools ever assembled into one package. We are confident you will find it to be one of the most useful disks in your library. Each and every module included in this package has been put through it's paces in real use. We feel you'll find them not only extremely powerful, but also user friendly. Many extras have been put into The Hacker's Utility Kit. Please be sure to read each segment of this manual before using any of the tools. This will insure that you obtain full use of each and every feature. Before we get on to the goodies, we want to thank the programers listed above for their efforts in writing this package. We are very proud to present their finest effort ever. They, just like you, are "Hackers" at heart. This program is a showcase of their real talent.

Loading Instructions

Place the Utility disk in your disk drive, reverse side up. Type < LOAD"*",8,1 > and hit RETURN. In a short time, the menu will appear. Use the cursor U/D key to move the hand-pointer to the desired feature. Press RETURN and that utility will automatically load in and self start. We'll discuss each utility in it's order of display on the menu.

Sector Usage and Error Scanner

Selecting input 1 from the main menu will automatically boot this utility. When the menu appears, you may make your selection using the cursor or number keys to position the arrow pointer. Press RETURN to activate your choice.

1. Scan Disk:

   P : Print output after scan (use standard Commodore printer).
   S : Begin scan.
   E : Exit to beginning menu.
   M : Modify range of tracks to scan. Defaults are 1-38.

The following characters are used in the scan to represent the condition of any scanned diskette.

   S : Sync track (1 sync, no data).

0 : Block header not found.
1 : No sync character found.
2 : Data block not present.
3 : Checksum errorin data block.
7 : Checksum error in header.
9 : Disk ID mismatch.
- : 1571 normal format with no data.
+ : 1541 normal format with no data.
. : Data in these sectors.

2. Directory : Read any diskette in the drive.

3. Quit : Reboot Hacker's Utility Kit main menu.

Density Scanner

Selecting input 2 from the main menu will automatically boot this utility. When the menu appears, you may make your selection using the cursor or number keys to position the arrow pointer. Press RETURN to activate your choice.

1. Scan Disk:

P : Print output after scan (use standard Commodore printer).
S : Begin scan.
E : Exit to beginning menu.
N : Modify range of tracks to scan. Defaults are tracks 1-38.

The following represents the values you can expect on a normal disk. Any deviation represents a non standard condition. (More than one scan may be needed to determine density on some diskettes.)

1 : Tracks 1-17.
2 : Tracks 18-24.
3 : Tracks 25-30.
4 : Tracks 31-35.

2. Directory : Read any diskette in the drive.

3. Quit : Reboot Hacker's Utility Kit main menu.

KRACKER HACKER GCR EDITOR

The GCR Editor is the most powerful tool you'll ever use to examine a disk. It will allow you to view raw data the way it was originally written to the disk. Our GCR Editor has every feature we could think of to examine and manipulate headers and data. A thorough knowledge of the makeup of Commodore format is necessary to have full use of this utility. For complete information on this subject, we suggest "Inside Commodore DOS", written by Richard

Immers. This manual contains a wealth of information on the makeup of the Commodore format and the Disk Operating System (DOS). With this manual and our GCR Editor, you can achieve a new level of understanding.

In the following instructions, we will give you all the command features available to you with the Kracker Hacker GCR Editor. Only use and study can make you proficient. Enjoy!

What is GCR?

When you load and save files from the C-64 to disk, they are not written bit for bit straight to the diskette. The Commodore 1541/71 disk drive cannot write more than three "0" bits in a row to a disk, so writing a hex byte like #$06 poses a problem! Commodore developers created the GCR coding scheme to read and write data to and from the drive. It converts each four bits of hex code into 5 bits of GCR code. For every four bytes of hex data, there are five GCR bytes. Lastly, this data is written at a standard rate, depending on its placement on the diskette. Standard Bit Rates are as follows: Tracks 1-17 = $60, Tracks 18-24 = $40, Tracks 25-30 = $20, Tracks 31-35 = $00.

Commodore DOS protection is, for the most part, simply the placement of NON-STANDARD data on the diskette. This can be created by using single bytes in non-standard locations, abnormal drive speeds, or rewriting the format (single sectors, tracks, or the entire disk). By using your GCR Editor, you can obtain exact format information. You even have the power to duplicate many protection schemes on non-working backups. Let's go through the commands available to you in this powerful utility. From the main start-up menu, choose option 3 and press RETURN.

First Screen (Header Selection)

Track Selection: Track values are entered in decimal. Values from 1-40.5 are accepted.

Bit Rate Selection: Press RETURN for default value, otherwise enter one of four bit rates ($00,$20,$40,$60).

After Scan of Track: The number of headers equals the number of syncs on a track. Left column = GCR of first 8 bytes. The right column = converted GCR bytes. The message bar just above the list of headers gives you information about the current header the cursor is on. Left hand will say: Sector: xx if the current header is part of a standard formatted track. It will give you the sector number in decimal so you can use the GCR Editor like a sector editor. The right hand will either say DATA or HEADER, depending upon whether the cursor is on the data block header (starts with a

$52) or the actual data block itself (starts with a $55).

Commands (First Screen):

Shifted H: Help screens.
T: Enter a new track.
R: Enter a new bit rate for the current track.
Fl: Directory of disk in drive.
F3: Prompt to reboot main menu.
Cursor U/D: Scroll through headers.
Space Bar: Read current selected header and go to edit (2nd) screen.
P: Print list of headers to printer (Standard Commodore printers).
+ or -: Go back or forwards one track and read.
C: Create a Track : You may access this feature after reading a track.

options Include:

1. Fill track with no-sync: wipes out entire track with $55s.
2. Fill track with full-sync: fills entire track with $FFs.
3. Create Notepad header: Wipes out an entire track with $55s, and then creates a one header/one sync track using Notepad code.

Second Screen (Header Edit Screen)

Header Info: Appears at the top of the screen. Sync is the actual length of the sync mark of this header. Length is the length in bytes of the header. Note: if the header has more than $0500 bytes, the buffer for editing will only go up to byte $O4FF, since the disk drive cannot read long blocks unless you have expanded memory.

Header and Data Tables: Rows of ten GCR bytes appear on the left. The converted eight hex bytes appear on the right. Remember, five GCR bytes equal 4 Hex bytes.

Commands (Second Screen)

R: Reread the header data.
W: Write altered data back to disk.
Z: Find zero GCR bytes and mark them.
P: Print out data to printer.
SPACE BAR: Enter edit mode.(See more info below.)
+ or -: Increment or decrement sync length by one.
CURSOR UD/RL: Move cursor around data table.
< : Delete one byte from cursor spot.
> : Insert one byte ($00) at cursor spot.
DEL: Delete bytes (from end of table)

S: Switch column editing from left to right.
A: Toggle Hex display Hex and ASCII (right hand of screen).
D: Enter disassemble mode.(See more info below.)
C: Repairs checksum of header or data block. Use before W command to prevent checksum error.
SHIFTED R: Lets you re-read current header at a different clock rate than the entire track was read at.
SHIFTED H: Help screens.
LEFT ARROW: Return to first screen.

Edit Mode: Hit SPACE BAR to enter, border will change color. Type in hex bytes, or ASCII, whichever is appropriate. DEL key will backup cursor. Hit RETURN to exit edit mode. Note: On the display screen, double dots II II mark bytes that aren't used. If you try to hit SPACE BAR to enter the edit mode on one of these bytes, it won't work (except, on the first ".." to the right of the last data byte displayed). Hitting SPACE BAR here allows you to append to the current data, the length of the header will change appropriately.

Disassembly Mode: Hit D to enter Disassembly mode. The disassembled code will appear in the GCR column on the left. Type in assembly text and hit RETURN to enter. Hit CURSOR UD to escape Assembly mode.

SPACE BAR: Enter disassembly mode.
CURSOR U/D: Scroll back and forth through the disassembly.
RETURN: Exit disassembly mode.
P: Send disassembled code to printer.

NoteDad Feature: At times when using the OCR Editor, you may want to save a header, look at another one, and later retrieve the original header without re-reading it. Our GCR Editor features a scratch pad (called the Notepad) that lets you save one header in memory. You can also edit the notepad header.

T: Toggles editing mode from current header to notepad. The border will change colors and the message "NOTEPAD" will appear in the top left corner. You can't use any disk commands like R,W,& Z in Notepad mode. Hit T to return to normal header program.
SHIFTED S: Save header to disk as a Notepad file. Save either notepad or selected header.
SHIFTED L: Load saved header from disk.
UP ARROW: Saves current header to Notepad.
CONTROL I: Only works in the non-Notepad mode in GCR editing. Inserts NOTEPAD header code at cursor position. Use to retrieve Notepad.
CONTROL A: Appends notepad header to disk at cursor spot. If

you have a long data block with extra room at the end, and you wish to add an extra sync to disk, move the cursor to the end of block, have the desired new header saved to the Notepad, and hit CONTROL A. The GCR Editor will automatically re-scan the track.

GCR Editor Hints, Tricks & Tips

Use caution when using the W command repeatedly. The GCR Editor writes each header back to the disk as perfectly as possible (ie: correct length, correct sync). If you make a header longer than it was before and write it back to the disk, it may destroy the header that follows it.

The same goes for the CONTROL A append command. Changing sync lengths and writing the header back to the disk is also dangerous. Use caution.

After you use the W command, you should verify that it wrote correctly by using the R command to re-read it.
Use the C checksum command after editing a data block before you write it back to disk. This repairs the data block checksum. Otherwise, normal Commodore DOS will get a 23 read error when it tries to read the block.

Well, there you have it. The most powerful, easiest to use GCR Editor on the market today. If you feel confused or overwhelmed, don't be put off. A little study and practice will have you feeling right at home.

Nybble Copier

Selecting input 5 from the main menu will automatically boot this feature. This utility has been designed to copy non standard material. It will in many cases, make a perfect copy of your protected diskette. Please keep in mind as you use this, or any nybbler, that nybblers are limited in their abilities. The following key strokes represent the user options.

Fl/F2 :	Increment or decrement starting track of copy range.
F3/F4 :	Increment or decrement ending track of copy range.
F5/F6 :	Increment or decrement Source device number (must be hardwired).
F7/F8 :	Increment or decrement Destination device number (must be hardwired).
SID :	Directory of diskette in source or destination drive.
Q :	Quit.
C :	Begin copy process.

File Track & Sector Linker~Tracer

At the "Filename" prompt, enter the file you wish to see linked. Press RETURN and the drive will search for that file. If the file is found on the disk, it will be visually linked on the Track/Sector map. After the file has been read in, a blinking cursor will appear on the beginning Track/Sector of that file along with the address of the first two bytes of that Sector.

1st Screen Commands:

- Fl : Directory of Diskette.
- F3 : Prompt to reboot main menu.
- RESTORE : Resets program and drive at any time except during linking.
- Cursor Down : Move forward link by link through file (slow scan).
- Cursor Up : Move backwards link by link (slow scan).
- Cursor Right : Move forward eight links (fast scan).
- Cursor Left : Move backwards eight links (fast scan).
- HOME : Return Cursor back to first link.
- Space Bar : Enter 2nd Screen (edit mode).

Notice that as you use the Cursor commands, the address counter is incremented to reflect the true address of first two bytes of the highlighted Sector.

2nd Screen Commands:

- Left Arrow : Return to first screen.
- W : Write altered Sector to disk.
- M : Toggle edit mode between Disassembly and Hex/ASCII display.

Disassembly Mode Commands:

- Home : Home Cursor back to first byte.
- Cursor U/D : Slow scroll through disassembly.
- Cursor L/R : Fast scroll through disassembly.
- Space : Enter edit mode. Type in assembly mnemonics. Be sure to use proper spacing. Hit RETURN after each change. A bad instruction will exit edit mode.

Hex ASCII Mode Commands:

- Home : Home Cursor back to first byte.
- Cursor R/L/U/D : Move cursor around display.

Type in a Hex Byte at blinking Cursor to change values. The ASCII display will change accordingly. Remember, all 2nd screen commands also apply to this screen.

Byte Pattern Finder

At the beginning prompt you may enter the bytes you are trying to loacte in any of three forms (one at a time please). Hex, Decimal, or ASCII will be acceptable. You are limited to two lines of input. An incorrect input will not be accepted.

```
   Enter Hex data as : $BD,$53,$22 (Notice the "$" and the ","
                       placements.)

   Enter Decimal data as : 200,255,36 (Notice the "," placements.)

   Enter ASCII data as : "welcome to " (Notice the quotes around
                         the string.)

   Combination  of  the  above  as : "welcome  to ",$8D,$53,$22,200
                                     ,255,36 (Notice the commas)
```

At the next prompt, choose the range of tracks (1-35 only) you wish to search. Hit RETURN to begin scan. The drive will then begin a fast search for the imputed data. Each time the data is found on the disk, the searcher will pause and report the occurrence. Press Space Bar to Continue the search, or RESTORE to return to the beginning menu and reset the drive.

Other Commands (while Cursor is blinking) are:

Fl : Directory of disk in drive.

F3 : Prompt to reboot Hacker's Utility Kit main menu.
Kracker Jax Para.eter~Copier Creator

Selecting input 8 from the main menu will automatically boot this feature. This utility will give you the ability to easily create a parameter, and incorporate that parameter into a copier utility. From the main menu, you will be presented a number of commands. The following keys represent your main input keys.

Fl: Directory of diskette in drive.
F3: Reboot Hacker's Utility Kit main menu.
F5: Fast Format a work disk. Will ask for disk name and ID number.
1. Parameter Name : Enter the parameter title (also used as it's file name).
2: Starting Track : Increment only. (Use default value of 1 in most cases.)
3: Ending Track : Increment only. (Use default value of 35 in most cases.)
4: Type of copier : Toggle between Data copier or Nybbler. We recommend the data copier in most cases. Occasionally only a Nybbler will do.
5: Enter Data : Data may be entered in Hex or Decimal. Toggle mode with left arrow key. Important : Data MUST be

entered as follows. Starting at position zero in the buffer, enter the Track. Position one, enter the sector to modify. Position two, enter the number of bytes you will be modifying. Position three, input the starting position of that change in the sector to be modified. The bytes from four on represent the actual byte changes. After the byte changes have been imputed, you have three situations. NUMBER ONE Another change in the same sector. In this case, enter one zero byte and number of bytes, position, and actual changes again. NUMBER TWO : Another change on disk. Enter two zero bytes and then enter all new information just as you did in the beginning. Remember,just continue on with your changes. Don't start over at position zero. NUMBER THREE Done. If all modifications are entered, enter three zero bytes. This will flag the utility that you are finished.

Press RETURN to lock in all changes.

S: Save copier/parameter to formatted work disk. Your modifications will be automatically executed after the created copier has been used. The created copier will contain the proper title, tracking information, and byte modifications. The user may simply load and run the copier. It will allow the use of either one or two drives.

Kracker-Non with Relocater and Op-Code Editor

From the main menu choose option 9 to access this utility. When the monitor menu screen comes up, use the cursor U/D keys or the 1,2,3,4, keys to choose an option. Press RETURN to execute that option.

Kracker-Mon is completely relocatable in memory The = and - keys will increment and decrement the monitor address. Hitting the RETURN key while the "Monitor=$XOOO" is highlighted will also increment the monitor to the desired Hex address.

Fl : Directory of disk in drive.
F3 : Prompt to re-boot the Hacker's Utility Kit main menu.
OPTION 1 : Execute chosen monitor. (See Monitor Commands).
OPTION 2 : Save chosen monitor to a work disk.
Saves autoboot file under name : "MONX000" . Just LOAD "MONX000",8,l to autoboot other save files. The op-codes listings will be saved as "OPS". The monitor will be saved as "XO"
OPTION 3 : Edit the op-code file (OPS) on any WORK DISK.
CURSOR U'D : Slow scroll through list.
CURSOR R/L : Fast scroll through list.
RESTORE : Reset to previous menu.
SPACE : Allows you to change the mnemonic.
A : Steps through the addressing modes (changes them).

```
      HONE : Returns cursor to beginning ($00 byte).
         S : Re-saves changed opcode file to a WORK DISK.
```

KRACKER-MON COMMANDS

```
         R : Displays status of A,x,Y registers and Stack pointer
         G : XXXX - Executes code starting at $XXXX
         X : Returns user to Basic
         M : FFFF LLLL - Displays in hex, memory between 2 two
             addresses. If a second address isn't specified, scrolls
             forever. RUN/STOP halts.
         @ : Sends disk command. Alone returns drive status. ~$ for
             directory of disk.
     SPACE : during directory pauses.
  RUN/STOP : abort directory listing.
         L : Load   file   from  disk.
             L"FILENAME" ,device# ,address (optional). For example-
             L "FILE",08,COOO (IF an address is given, it WILL load
             to that address.)
         V : Verify file in memory.
             V"FILENAME",device,address(optional). Same as Load
             command but Verify instead. A "?" stands for verify
             error.
         S : Save File - S "FILENAME",device,FFFF,LLLL+l
             Example  S "FILENAME",08,COOO,DOOl
         F : FFFF LLLL xx - Fills memory from $FFFF to $LLLL with
             $XX byte.
         D : FFFF LLLL ($LLLL Optional) - Disassembles memory. Use
             CURSOR U/D to scroll through listing. Editing is
             possible using mnemonic changes.
         P : Send code to printer - PD FFFF LLLL sends disassembly
             listing.
             PM  FFFF  LLLL  sends  HEX Memory  listing.
             (Commodore  1525 compatible only)
         A : XXXX mnemonic commands - Assemble code beginning at
             $XXXXX (Be sure to use proper spacing between
             characters.)
         H : FFFF LLLL PATTERN - Hunts from $FFFF to $LLLL for up
             to an eight byte pattern. Use quotes on either side of
             an ASCII pattern. ASCII and Hex may be mixed.
         T : FFFF LLLL XxXX - Transfers memory from $FFFF through
             $LLLL to $XXXX.
        TC : Use same syntax as T command. Will transfer computer
             memory to drive.
        TD : Use same syntax as T command. Will transfer drive
             memory to the computer.
        TF : Same syntax as T command. Fast command version of TC.
             Warning: $XXXX can't be between $0001 and $0147.
         O : This is the letter 0 not a zero. 0 followed by an 8,9,
             A,B (device number) will put you in the drive-mon mode
             for the specified drive.  The above commands are the
             same for the
```

drive-mon except the P feature is inactive. For printer listings of drive memory, send the code to the computer, then the printer. 0 and RETURN sends you back to the computer memory. A "]" lets you know you're in drive memory, while a "." denotes computer memory. To assemble/disassemble beneath ROMS and the VIC CHIP, change location $0002 as if it were $0001. $0001 can't be changed through the monitor.

```
      $0002: $37 = All ROMS in.
      $36 = Bank out BASIC.($AOOO-$BFFF)
      $35 = Bank out Kernal & BASIC.
      $30 = Bank in RAM under $DOOO.
      $31 = Bank in character ROM under $DOOO.
```

Single Track or Whole Disk Formatter

Selecting input 10 from the main menu will automatically boot this feature. This utility has been designed to allow you to fast format either a single track (perfect for creating 29 Errors) or the whole disk. When the menu appears, you may select an option by using the cursor or number keys to move the arrow pointer. Use the RETURN key to activate your selection. The following keystrokes represent your options.

1. Format one track.
   Fl/F2 : Increment or decrement to proper track.
   F : You will be prompted for a two character ID number.

Formatting will follow.

R : Return to format menu.
Restore : Return to menu at any input pause.

2. Standard Format.

   You will be prompted for new name and ID number. Five characters are accepted. The last two characters will become the true disk ID Numbers.

3. Directory diskette in drive.

4. Exit back to Hacker's Utility Kit main menu.

Disk File L~ger

At the "Log Which Files?" prompt, either press RETURN to accept the "*" default (which will log all files) or enter an individual filename to log that file.

Examples:

```
Log Which Files? : *  = log all files

Log Which Files? : B* = log all file starting with "B"
```

Log Which Files? : DISK = log file called DISK.

At the next prompt, "Do you want a printout?", press RETURN to accept the default value of "NO". Hit the "Y" key to send output to the printer (Commodore compatible) as well as the screen. The logger will mark files as "Bad" if they have illegal Track or Sector numbers. You can assume these are either dummy files or files that are manipulated by special DOS routines.   As a disk is logged, the disk name and ID number will appear at the top of the screen. Below, a list of each filename will be displayed with their start and ending addresses in Hex.

Other Commands:

Fl	Directory disk in drive.
F3	Reboot prompt to return to main menu
RESTORE	Reset the program and the drive back to beginning
RUN/STOP	Pause key - active only while logging.
SPACE	Continue after pause.

HESMON INSTRUCTIONS

If You've Never Used a 'Machine Language Monitor' Before

The following section is intended for people who are unfamiliar with the uses of a machine language (M.L.) monitor program. However, It is not a tutorial in the architecture of the C64 or the 6502. Nor is it intended to teach 6502 assembly language programming. In fact, some knowledge of assembler language will be most helpful. It is intended to help the beginner get started in using HESMON. Even those who know nothing about the 6502 or the C64 will find some of HESMON's commands useful (see, for example, the Interpret Memory command).

If you are familiar with the C64's screen editor, you should have no trouble entering and editing HESMON commands. HESMON commands are entered and edited just as are BASIC direct mode commands. They consist of a single character usually followed by one or more 'parameters' and a RETURN. The parameters consist of hexadecimal numbers or character strings and are separated from one another by spaces. With one exception (the 'U' command) numeric parameters must be hexadecimal and do not need to be prefixed with '$'. String parameters are identified by enclosing them in double quotes ("). If HESMON doesn't understand a command it will print'?', usually just to the right of the bad command. If the command is understood, but the result is impossible or illegal, e.g., trying to save HESMON itself on tape, HESMON prints a'?' on the following line.

To use HESMON, turn your C64 off, insert the HESMON cartrtdge into the expansion slot In the C64 and then turn the power on. You will see the HESMON version number, the programmars name, the H.ES. copyright message, and the 'cold start' register display:

```
C.

 PC   IRQ SR AC XR YR SP
0000 EA31 27 On 00 00 FA
```

The meaning of this rather cryptic display is as follows: The first line 'C" identifies a cold start of HESMON, that is, starting up from power-on. The next line identities the pseudo 6502 registers maintained by HESMON:

```
 PC = program counter
IRQ = interrupt request vector
 SR = Status register
 AC = accumulator
 XR = X register
 YR = Y register
 SP = stack pointer
```

NOTE: "6502" is used synonomously for "6510" in this document.

The register contents are shown on the third line. The quantities shown in the register display (except the IRQ) are not the actual register contents, they are the numbers HESMON will use to set the 6502 registers when instructed to begin execution of a M.L. program. IRQ is not a 6502 register, but a RAM 'vector' that points to an IRQ interrupt service routine. Beginners may ignore this location - but better not change it! The ';'at the beginning of the last line is really a HESMON command. It tells HESMON (it the RETURN key is pressed with the cursor on this line) to put the seven numbers that follow into the corresponding pseudo registers. Just before beginning execution of a M.L. program HESMON copies the pseudo register contents to the 6502 registers. so, for example, If we want the C64 to print 'HI.', we could first move the cursor up to the';' line and alter it to read:

```
   1200 EA31 27 48 49 2E FA
```

When we press RETURN, the 6502 pseudo program counter is set to $1200, while the accumulator, and X and Y pseudo registers are set to $48 (ASCII H), $49 (ASCII I), and $2E (ASCII). Now, if we write a program at $1200 to print the AC, XA, and YR it will print 'HI.' when we execute the HESMON Go command. Let's write such a program using the HESMON Simple Assembler command, 'A'. Type In the following lines:

```
   A 1200 JSR FF02
   TXA
   JSR FF02
   TYA
   JSR FF02
   BRK
```

The 'A' beginning the first line tells HESMON we wish to assemble, that is, translate assembly mnemonics into machine code. As you press RETURN after typing each of the above lines, you will see HESMON reprint the line, showing the machine code generated from the assembly language instruction. HESMON will then prompt for the next line of program by printing the 'A' command and the next available address followed by a space. So you don't have to keep track of what the next address is, just type in the assemtty language instructions. When you've finished the program, just press RETURN and HESMON will exit this mode. By the way, $FFD2 is one of the 'Kernel'routines in the C64's ROMs. It prints the contents of the accumulator to the current output tile the screen in this case. For further information on this and other useful ROM routines, consult the Commodore 64 Programmers' Reterence Guide" published by Commodore.

Now type 'G' and hit RETURN. You should see:

```
   G

   Hi.
   B*
   PC  IRO SR AC XR YR SP
   ;120C EA31 30 2E 49 2E FA
```

Notice after the 'HI' is another register display, the break entry display identified by 'B*'. This means we've reenterad HESMON by executing a BRK instruction the one at the end of our short program. Now examine the register contents. The PC points one address higher than the BRK instruction. The X and Y registers and stack pointer are unchanged. The accumulator now has the $2E transferred into it by the TYA instructional $1201. Let's play with this a bit. Type 'D1200 120B' This command instructs HESMON to 'disassemble' the program you just entered.

Now, move the cursor to the last line, at address $1208, and type the following, with the 'A' replacing the ',' (also be sure to blank out any characters left on the screen after the '8')

```
   A 1208 LDA #48
   JMP 1200
```

We now have a M.L. program that will print 'HI.' forever - or until we stop it. Type 'G1200'. When you tire of watching the stream of 'HI.HI.HI.'s, press - no, not the STOP key - the RESTORE key by itself. The RESTORE key is HESMON's super-STOP key. It will halt just about any M.L. program (except HESMON itself when HESMON is plugged in. (Exception: It you attempt to use RS232 files all bets are off. Also, correct operation of RS232 files is not guaranteed with HESMON installed.) To get back to our example: alter pressing RESTORE you should see a clear screen with the following:

```
   S*
     PC   IRQ SR AC XR YR SP
   ;XXXX  EA31 XX XX XX XX XX
```

This is the RESTORE entry display, identified by the 'S*'. The X's are not actually what you will see. The register contents will depend upon exactly when you pressed RESTORE.

It you want to enters series of bytes into memory, use the Memory Modify command (:). For example, to enter the sequence $01, $02, $03, $04, $05, $08, $07... starting at $1234, you type:

```
   :12340102030405060708
```

HESMON will respond by reprinting the line and will prompt for another line by printing the next available address. As with the Assemble command, you may exit by typing RETURN. Besides entering programs and data into memory, one of the functions of a M.L. monitor is to examine programs and data already in memory. HESMON has several commands for this purpose, including Disassembly (D), Memory Display (M), and Interpret Memory (I). These three commands are special in that the cursor-up and cursor-down key may be used to 'scroll' their displays forward and backward through memory. The action of this scrolling is easier to use than to describe. Think of the text on the screen as being on a drum which may be rolled up or down using the cursor up/down key. The scrollable display type found closest to the edge of the screen where new lines will appear is continued in the scroll direction. I said it was hard to describe. Try it. Just type 'D AAD7' and hit RETURN. Then press and hold the cursor-down key. To scroll up, go to the top of the screen and then hold down the cursor-up key.

Other commands allow you to hunt for a particular Sequence of bytes in memory (H), compare two blocks of memory for differences (C), or transfer a block of memory to a different location (1). There are also two advanced functions: N-relocate absolute memory references in a program, and E- change the external references in a program. Finally, there are number base conversion and hexadecimal arithmetic functions.

Alphabetical List and Description of HESMON Commands

The following section lists the HESMON commands in alphabetical order describing each in detail and giving example(s) of its usage.

A - The Simple Assembler The HESMON simple assembler provides en easy way to enter Short M.L. programs. It does not have all the features found in a complete assembler such as HESBAL in HES's 6502 Professional Development System for the VIC and Commodore64, but it provides increased convenience compared to POKEing from BASIC or entering hexadecimal codes using a more primitive monitor. The syntax of HESMON's Assembler command is as follows:

```
   A 1111 MMM 00000
```

where '1111' is a four digit hexadecimal address in the C64's RAM, 'MMM' is a standard three character assembler mnemonic for a M.L. operation code (opcode), such as JSR, LDA, etc. '00000' is the 'operand' of the op-code. It's beyond our scope here to discuss fully the meaning of those parameters - for a complete discussion, consult a book on 6502 assembly language programming. See Section I for a simple example of A's usage. Notice that since all numeric operands MUST be in hexadecimal notation the customary "$" preceding these numbers is Optional- as is the preceding 'X' or 'Y' in indexed instruction operands. If HESMON understands the line, if will reprint it showing the corresponding byte(s) of M.L. between the address arid tire assembly code. HESMON will then prompt for the next line of assembly code by displaying the next address followed by a space and the input cursor. If HESMON cannot interpret the line, It will print '?' instead of prompting for the next line. For example, you type:

```
   A 12OO LDA#41
```

HESMON responds by overprinting your line and then prompting for the next line as follows:

```
   A 12OO A9 41   LDA#$41
   A 1202
```

Note - HESMON ignores anything to the right of a':' on the line.

B - Breakpoint Set

There are three different methods to return to HESMON from a M.L program. The Breakpoint Set command is one of them. This command allows you to designate an address in a program as a 'breakpoint,' that is, a place where the program is to be halted and control is to be returned to HESMON. Breakpoint Set also allows you to specify the number of times the instruction at this addresS is to be executed before the breakpoint is activated. The breakpoint defined with Breakpoint Set is effective ONLY when the C64 is executing HESMON's Quick Trace command. For example, to halt a program, first starts at address $1200, on the fifth repetition of the instruction at address $1234, you would type:

```
   B 1234 0005
   Q 1200
```

The first line above sets the breakpoint at $1234 and the repeat count to five. The second line initiates the Quick Trace mode of program execution (see the Quick Trace command). When address $1234 has been reached for the fifth time HESMON will halt execution of the program, display the current values of the 6502 registers, and enter the single-step mode of execution (see the Walk command).

The second method to return to HESMON from an ML program is to insert a 6502 'BRK' instruction into the program. Obviously, since this method requires prograrn modification, it may be used only with programs in RAM. Finally, HESMON may be called by simply pressing the RESTORE key. In either of these last two cases HESMON will be re-entered whether or not the Quick Trace mode was active. If a BRK instruction was encountered, the 'break' entry register display will be printed showing the contents of the 6502 registers. Similarly, if the RESTORE key is pressed, the RESTORE entry register registry is shown in the latter case, the screen is cleared first. The RESTORE key method of HESMON re-entry will work any time the HESMON cartridge is plugged in - unless an RS232 file has been accessed or the 6502 attempted to execute an undefined opcode (one that disassembles as '???'). After an RS232 file has been attempted HESMON may be re-entered from BASIC via a BRK instruction. Type 'SYS8' to cause a break entry.

C - Compare Memory Blocks

This command compares two sections of memory and reports any differences by printing the address of one member of the mismatched pair(s). The syntax is as follows:

```
   C 1111 2222 3333
```

where 1111 is the start address of the first section, 2222 is the end address of the first section, and 3333 is the start address of the second section the one to be compared with the first sectton. This command may be stopped (in case a large number of addresses are printed) with the STOP key. For example, suppose you have two disk files containing (you thought) the same M.L. program residing at locations $1400 to $147F. However, when you used the BASIC command VERIFY, it said 'VERIFY ERROR'. Naturally, you wonder just where the difference is. VERIFY can only tell you they differ SOMEWHERE. Compare Memory Blocks may be used to find out: First use HESMON's Load command to load one of the files (See Load). Then move thit program to $1500 using the HESMON Transfer Memory Block command: T 1400 141F 1500. Next Load the other file. Now compare the two files using Compare Memory Block:

```
   C 1400 147F 15OO
```

HESMON will print a list of all the memory locations which differ between the two programs.

D - Disassemble Memory

This command is the inverse of the Assemble command. It interprets memory contents as M.L. instructions and displays the assembly language equivalent. Disassemble is used in two distinct ways. First, it may be used to disassemble a section of memory by specifying an address range, such as:

```
   D 1111 2222
```

where 1111 is the start address and 2222 is the end. This type of disassembly is convenient when used in conjunction with HESMON's Output Divert command to produce a hardcopy listing of a M.L. program. Second, the disassemble command may be started by entering a single parameter, the beginning address:

```
   D 1111
```

This mode is handy for examining a M.L. program on the screen because, once the first line is displayed, preceding or subsequent lines of code may be disassembied by pressing the cursor-up or cursor-down key respectively.

You may alter a program in RAM using the Disassemble command's output. If you move the cursor to the line you wish to alter, change the byte display (not the mnemonic), and press return, HESMON will atler the emory contents and retype the line showing the altered bytes and the corresponding disassembly. Then HESMON will prompt for the next line by printing the next address and leaving the input cursor on the same line. To exit this mode type RETURN, just as with the Simple Assembler command.

E - External Relinker

This command is rather difficult to understand, but the effort is worth it! Basically, this command facilitates the transport of M.L programs from one 8502-based computer to another (PET, VIC, etc.) by translating the system calls of one computer to those of another. Of course the capabilities of these computers are different so one cannot always achieve a perfect translation, but at least a functioning version can be made without completely rewriting the program. The heart of this command is a table of corresponding addresses. This table contains four-byte entries consisting of pairs of addresses. These address pairs are the addresses in the respective computer operating Systems that perform a given task. Typically these will be addresses in the ROM firmware of the computers. The correspondence table must be Supplied by you. Lists of common ROM routine addresses in various 6502 computers have appeared in several places, most notably in COMPUTE! magazine (e.g., "VIC Memory Map Above Page Zero", COMPUTE! Vol.4, No.1, P.181); "Butterfield on Commodore", Commodore Magazine, Oct., Nov., 1982, pp. 81 ff.; and, for the PET, in "PET/CBM Personal Computer Guide" by Osborne and Donahue.

For example, suppose you have loaded into your C64 an M.L. program intended to run in a PET with BASIC 4.0 ROMs. We will assume it is in locations $1200 to $i3FF. Many of its external subroutine calls are probably of the form JSR $FFxx. The subroutines at these addresses are all almost identical in function to those of the same address in the C64 because these entry points are in a 'jump table' set up for the purpose of standardizing System calls between the different Commodore ROM Sets. So what's the

problem? Any subroutine call in the address range $B000 to $FFOO probably also has an equivalent In the VIC, but it's at a different address. This is where External Relinker comes in. External Relinker will find such subrouttne calls end replace them with the corresponding C64 ROM routine calls - if we can identify the correct replacement (this is where the published ROM maps come in). If we already have a correspondence table constructed in an earlier session with External Relinker, we simply load it using the load command. But, if we don't have a table, External Relinker will use our answers to its queries to construct one we may have for future use. For the present example, suppose we have no table, just two ROM maps. We want to construct a table starting at $1000, so we start it by entering four zeroes (four zeroes denote the last entry in the table) using the Fill Memory Block command.

```
   F 1000 1003 00
```

Then we start External Relinker:

```
   E 1200 13FF 1000 BOOO FF00
```

The first two parameters tell External Relinker where the start and end of the program we are working on are. The third says where the correspondence table starts. The last two give the address range we're interested in relinking. At this point External Relinker will start disassembling our program in from $1200 to $13FF, looking for references to addresses in the specified range of $8000 to $FFOO. When it finds such an address it will first consult the correspondence table which starts at $1000- If no entry for the address is found, it will show the disassembled line containing the unknown address and wait for the entry of the correspondence address. We will look up the PET address in the published table, find its equivalent in the C64 table, type the VIC address over the one on the screen, and press RETURN. HESMON will add the new correspondence to its table, alter the address reference in the program and then continue its search. On subsequent occurrences of this address HESMON will automatically make the specified replacement.

F - Fill Memory Block

This command is used to Set a section of memory to a particular value. The syntax is as foltows:

```
   F 1111 2222 33
```

where 1111 and 2222 are the first and last addresses (inclusive) of the section to be filled and 33 the hexadecimal quantity to be written. See, for example, the usage in the example of External Relinker.

G - Go (execute program)

This command transfers control of the C64 to a M.L. program; that is, it starts execution of the M.L. program. It may be used with or without an address parameter. If no address parameter is given, execution is begun at the address shown in the program counter (PC) of the Register Display command. For example you may exit HESMON and 'warm start' BASIC by typing:

```
   G A414
```

The C64 will respond, "READY.". For another example, see SectIon 1.

H - Hunt for a Sequence

This command locates a specific sequence of bytes in memory. It has two forms, as follow:

```
   H 1111 2222 33 44 55....
   H 1111 2222 "ABCDF"
```

where 1111, 2222 are the first and last addresses of the range of memory to be searched and 33,44, etc., are the hexadecirnal byte(s) to be found, separated by spaces. The second form allows the bytes to be specified as characters enclosed by quotes. For esample to find all subroutine calls to the character output routine (AB47) in the C64 ROM's we would type:

```
H AOOO FFFF 2O 47 AB
```

HESMON responds with a list of all such subroutine calls. Note that, as usual, the low and then high order bytes of the address were specified.

To find all occurrences of the string 'READY' (there is only one, at $A378), we would type:

```
   H AOOO FFFF "READY"
```

I - Interpret Memory

This command displays the contents of memory as 'ASCII' characters. It is similar to the Memory Display command except that it shows 32 characters per line. It may be used with either one or two parameters and its output may be scrolled just as with the Disassemble command. For example, to see the table of BASIC's keywords and error messages, type:

```
   I A000 A300
```

L - Load 'Program'

This command 'loads' (i.e., reads) a 'program' into memory from an external device such as tape or disk. The loaded malerial need not actually be a program. For example, it may be a section of memory containing a data table for External Relinker that was saved to tape or disk using the Save command. However, the most common use of load is to retrieve M.L. programs from tape or disk. Note that HESMON's load should NOT normally be used to load a BASIC program.

The syntax of Load is as follows:

```
   L "programname"11
```

where 'programname' is the name of the file to be loaded (be sure to include the double quote marks) and '11' is the device number from which to load. If the device number is omitted, the tape drive will be assumed, if the filename is also omitted, the first file found on the tape will be loaded.

For example:

```
   L"YAHTZEE"08
```

The above loads YAHTZEE from device eight, the disk drive.

M - Memory Display

This command displays the contents of memory in hexadecimal notation. It is similar to the Disassemble command in that it may take either one or two addresses as parameters. The two-parameter from displays from the first address to the second; the one-parameter form shows eight bytes beginning with the address given. Also like the Disassemble command, the output of Memory Display may be scrolled up or down with the cursor-up and cursordown key. For example:

```
   M AOOO A040
```

shows from $A000 through $A047 in hex and in characters, eight bytes per line. To see more, press cursor-up or down.

N - New Locator

This command is a relative of the External Relinker command. It has a different general purpose, however. New Locator is designed to convert absolute address references in a M.L. program from one memory range to another. It is typically used following a Transfer Memory Block command to relocates program in memory. For example, suppose you have just moved a M.L. program from $1200-$1280 to $13O0-$1380 using T. Any address references within the program now point $0100 too low. New Locator can fix this. Type:

```
   N 1300 1380 0100 1200 1280
```

The meaning of the above line is as follows: Disassemble from $1300 to $1380 checking for addresses in the range $1200 to $1280. Add $0100 to any such addresses. If we had moved stable of addresses, for examples 'jump table' (pairs of numbers of addresses, low byte followed high byte), instead of actual machine code; we would puts 'W' following the last parameter to tell New Locator to treat the memory contents as pairs of address bytes rather than M.L The general Syntax for New Locator is the following:

```
   N 1111 2222 3333 4444 5555 [W]
```

where 1111 and 2222 specify the actual memory range to scan, 3333 is the 'offset' to add to adjusted addresses, 4444 and 5555 specify the address range of references which are to be adjusted, and W (if present) specifies that the scanned range is a table of 'words' with no opcodes. If not in the 'word table' mode, New Locator will halt and display any line of machine code it can't disassemble.

O   - Output Divert

This command is HESMON's equivalent to BASIC's CMD command. It allows HESMON's output to be printed on the C64 printer or stored in a disk file instead of being displayed on the screen. This is the preferred method to get HESMON's output on a device other than the screen. Output Divert has a number of options. The complete syntax of the command is:

```
   011 22 "filename"
```

where '11' is the device address where the output is to be sent (normally 04 for the printer), '22' is the 'secondary address' of the device (typically 02 to 0E for the disk drive), and 'filename' is the filename to be used for storing the output (See your disk drive documentation). All of these parameters are optional. If you merely type '0' HESMON will open a file to device 4, the printer, and start diverting its output. If you type '0' when the output is already being diverted, the file will be closed and the output will be directed to the screen again. That is, typing '0' 'toggles' Output Divert on and off. If you want explicitly to revert to screen output, type '03F'. The secondary address and filename default to 'none" since they are not needed by the printer. For more information about

filenames and secondary addresses, consult the documentation for the device to which you wish to divert HESMON's output.

P - Print Screen

This command is a limited version of Output Divert. It copies the current screen display to printer or disk. It's just like having a snapshot of the current screen image. The parameters of Print Screen are the same as for Output Divert, except there Is no toggling because Print Screen automatically reverts to screen output at the completion of the screen copy. Note: Print Screen wilt NOT copy high resolution graphics.

Q - Quick Trace

This command is used after the Breakpoint Set command in debugging M.L programs. It takes one or zero parameters just like the Go command. If specified, the parameter gives the address at which to hegin execution. If omitted, execution begins at the PC shown in the register display. The difference between Quick Trace and Go is that a breakpoint, defined with the Breakpoint Set command, is only recognized in the Quick Trace mode of execution - the breakpoint will be ignored if execution is begun with the Go command. Program execution is much slower with Quick Trace than with Go because Quick Trace is really just a fast version of the Walk (single step) command. Using Quick Trace, instructions are executed one at a time and HESMON is re-entered alter each. This process continues until the defined breakpoint is reached. For an example of Quick Trace usage, see the Breakpoint Set command.

R - Register Display

This command displays HESMON's current 6502 pseudo register contents as well as the current interrupt request (IRQ) RAM vector. The IRQ vector is shown as a convenience to the programmer who wishes to use this vector to run interrupt driven or 'background' routines. This vector may be altered like any of tile register contents however, extreme caution must be exercised in so doing because the replacement is made IMMEDIATELY, not at the time of execution of a Go command. Therefore, the interrupt handling routine must be inplace BEFORE the IRQ vector is altered.

There are no parameters for the Register Display command, just type 'R'. To alter the register contents, move the cursor to the line beginning with ';' and overwrite the display. Then hit RETURN and the contents will be altered. Note that the display, except as noted for the IRQ vector, shows the contents of the 6502 registers at the time HESMON was entered. These registers will be set by HESMON to the values shown in the register display just prior to beginning execution of a program using the Go, Quick Trace, or Walk commands. For a fuller discussion of the mearring of this display, see Section I.

S - Save 'Program

This command saves the contents of a specified range of memory to an external (device as a non-relocating 'program' file. The 'non-relocating' part means that the program may be reloaded from tape using BASIC's LOAD command. The syntax of Save is as follows:

```
S "filename" 11 2222 3333
```

where 'filename' is the filename to be used (don't forget the double quote marks), '11' is the device number on which to save (01, for the tape and 08 for the disk drive). '2222' is the beginning address. '3333' is the last address PLUS ONE of the memory area to be saved. All the parameters must be given, except that in tape saves the 'filename' may be null (""). For example, to save an M.L. program residing from $1500 to $1DFF to the disk as 'APROGRAM', type:

```
S "A PROGRAM"08 1500 1E00
```

Again, notice the last parameter is one byte higher than the last program address. Also, note that HESMON's Save should NOT be used to save BASIC programs because HESMON saves programs as absolute, not relocatable, files.

T - Transfer Memory Block

This command transfers the contents of a block of memory to another area. Its syntax is as follows:

```
T 1111 2222 3333
```

where 1111, 2222 are the first and last address (not last plus one) of the block to move and 3333 is the starting address where the block is to be moved to.

U - (Test Color RAM)

U has no parameters. It tests the color RAM for proper function and prints 'OK' if they are working. If there is a bad byte, it's address will be printed.

V - Verity RAM Function

This command tests a section of RAM for proper function. Its syntax is:

```
V 1111 2222
```

where 1111, 2222 are the first and last memory locations of the block to test. HESMON will keep cycling the test over the address range specified until the STOP key is pressed (it may be necessary to hold it down for a second or two). At the successful completion of each test of the memory block, HESMON will print a'.' to show it is working. If a memory location fails the test, HESMON will print its address followed by a binary number showing the data incorrectly stored. The bits of the number are shown most significant (bit 7) to least significant (bit 0) left to right. The bits of the RAM location that are different from the test data are printed in reverse field. Using the information printed on the screen, it will usually be possible to pinpoint the bad RAM IC(s). Note that if you 'test' addresses that contain no RAM, a seemingly random pattern of numbers will be printed.

W - Walk Program

This command causes single-step execution of a M.L program under user control. It, like Go and Quick Trace, may he used without a parameter to begin at the register display 'PC' location, or it can accept one parameter that specifies the starting address. To exit the Walk mode, press the STOP key. To step as rapidly as the registers can be printed, press the SPACE bar. To step at the key repeat rate, press a normally repeating key, e.g., the cursor down key. To take one step only, press a normally non-repeating key, e.g., the left arrow key. The 'J' key has a special function in Walk mode. it causes HESMON to continue execution at full speed until a return-from-subroutine instruction is executed. For example, type:

```
W AAD7
```

HESMON will begin execution at $AAD7 - the carriage return, linefeed output ROM routine. After executing the instruction at that address HESMON will halt, showing the register contents and a dissasambly of the next instruction the C64 will execute if Walk is continued. The display in the above example is as follows:

```
25 0D 00 00 FA

,AAD9 20 47 AB JSR $AB47
```

The first of the two lines above shows the 6502 register contents in the same order as the Register Display command: SR AC XR YR SR This example assumes HESMON has just been cold started, otherwise the registers - except the accumulator - may differ from those shown here. The second line shows that the C64 will next do a subroutine call to $AB47, the character output routine used by BASIC. To continue, press any key except STOP or 'J' (no need to hit RETURN). Suppose we press the left-arrow key once. HESMON will now show two more lines:

```
250D 00 00 F8
AB47 20 0C E1 JSR $E10C
```

Now we see Ihe C64 is at location $AB47 about to execute a subroutine call to $E10C. Notice the stack pointer (SP) has been decremented by two because the return address for the JSR instruction was 'pushed' on the stack before the jump to $AB47 was executed. Let's press the left-arrow once more:

```
25 0D 00 00 F6
E10C 20 D2 FF JSR $FFD2
```

Here we finally get to a place where the C64 is going to a 'Kernal' routine we can recognize: the character output routine $FFD2. Since this routine is documented in the C64 literature, we know exactly what it will do: print the character $OD in the accumulator. Therefore, we needn't single step further through that routine. So we press the 'J' key. HESMON shows (after a blank line - where the carriage return was printed):

```
20 0D 00 06 F6
E10F BO E8  BCS $E0F9
```

Now the C64 is at the point just following the JSR $FFD2 instruction. The 'carry' bit (bit 0) of the Status register (SR = $20) is clear (0), so the branch on carry set (BCS) will not be taken. At this point we may continue to single step through this subroutine by pressing left-arrow; return to the next higher level of code (SP = $F8) by pressing 'J'; or quit the Walk command by pressing STOP.

X -Exit to BASIC

This command gives control to the C64's BASIC interpreter. It has two forms. The first form 'XC' has the same effect as if the C64 were turned off and then back on without the HESMON cartridge plugged in except that HESMON may be entered by pressing RESTORE. The second form 'X' causes a 'warm start' of BASIC,

similar to pressing RESTORE when HESMON is not plugged in. Your first exit to BASIC from HESMON after turning on the C64 should be an 'XC', otherwise BASIC may misbehave. While in BASIC, to achieve the same effect as pressing STOP & RESTORE without HESMON: First press RESTORE. Then type 'X' and hit RETURN.

# - Convert Decimal to Hexadecimal

This command prints the hexadecimal equivalent of a decimal number. If the decimal number is negative it shows the two's complement 16-bit hex equivalent and the corresponding positive decimal number. For example:

```
# 1234
```

HESMON shows (on the same line):

```
#1234 =$04D2 1234
```

$ - Convert Hexadecimal to Decimal

This command prints the decimal equivalent of a hexadecimal number. For example:

```
$ ABCD
```

HESMON shows (on the same line):

```
$ ABCD 43981
```

+ - Hexadecimal Addition

This command prints the sum of two hexadecimal numbers in hex and decimal. All four digits, including leading zeros if needed, must be used. Example:

```
+    1234 5678
```

HESMON shows (beginning on the same line):

```
+ 12345678 = $68AC 26796
```

- - Hexadecimal Subtraction

This command prints the difference of two hexadecimal numbers in hex and decimal:

```
-    1234 5678
```

HESMON shows (beginning on the sameline):

```
- 1234 5678 = $BBBC 48060
```

Notice that the decimal number in this example is positive even though we would expect the result of this subtraction to be negative. This is because the two-byte number $EBEC doesn't retain the information that the result is negative. It you want to know the true negative decimal result, either type in the operands in the reverse order, or type:

```
-    0000 BBBC = $4444 17476
```

So, the true decimal value of the difference $1234- $5678 is - 17476.

Things to be careful about when using HESMON

The BASIC interpreter has control of the C64 at all times when BASIC is running. This means that the worst that's likely to happen if your BASIC program has an error is that BASIC will issue a 'SYNTAX ERROR' message and stop your program. A M.L. monitor, on the other hand, must allow its user to take complete control of the C64 to execute certain commands. So, if your M.L. program has an error and you attempt to execute it using the Go command, the likely result is that the C64 will go catatonic - that is, even Ihe RESTORE key may not bring back HESMON. In this event you will have to turn the power off and back on to get back to HESMON. You may avoid this catastrophe by using the Walk command to check out your program. Nevertheless, you can still send the C64 to never-never land by attempting to Walk through an instruction that disassembles as '???'. These instructions are 'unimplemented op-codes'. They do not have defined result. Many of them cause the 6502 to 'crash' - that is, enters a state from which it may be recovered only by powering on again.

HESMON uses 33 bytes near the bottom of the machine stack ($120-$141) for its variable storage. Most M.L. programs do not uses sufficiently large amount of the stack to interfere with this storage - but it is a possibility to be aware of. Large, complex BASIC prorgams sometimes do use enough of the stack to interfere with these locations. And finally, RS 232 files will not work correctly when HESMON is plugged in.

Acknowledgements

The seeds of HESMON are contained in the public domain monitor programs for the PET/CBM computers known as MICROMON and EXTRAMON. These programs, while not directly useful in the C64 environment, provide at least the general framework and the philosophy of user-friendliness which distinguish them and HESMON from other M.L. monitors of the author's experIence.

VIC, PET, C64 and CBM are trademarks of Commodore.

Copyright Notice

Although we make every attempt to verify the accuracy of this document, we cannot assume any liability for errors or omissions. No warranty or other guarantee can be given as to the accuracy or suitability of this software for a particular purpose, nor can we be liable for any loss or damage arising from the use of the same.

HESMON 64 is a registered TM of HES.

Appendix A

The HESMON Commands in Brief

The following is a condensed list of HESMON's commands for quick reference. Brackets ([]) denote optional parameters.

```
   A 1111 MMM 000000 - Simple Assembler
   B 1111 2222 - Breakpoint Set
   C 1111 2222 3333- Compare Memory Block
   D 1111 [2222] - Disassemble
   E 1111 2222 3333 4444 5555 [W] - External Relinker
   F 1111 2222 33 - Fill Memory Block
   G [1111] - Go
   H 1111 2222 33 44 55....or
     1111 2222 "XXXXX...." - Hunt for sequence
   I 1111 [2222] - Interpret Memory
   L "name" 11 - Load Program
   M 1111 [2222] - Memory Display
   N 1111 2222 3333 4444 5555 [W] - New Locator
   O [11[22["name"]]] - Output Divert
   P [11[22["name"]]] - Print Screen
   Q [1111] - Quicktrace
   R - Register Display
   S "name" 11 2222 3333 - Save Program
   T 1111 2222 3333 - Transfer Memory Block
   U-Test Color RAM
   V 1111 2222 - Verify RAM
   X[C] - Exit to BASIC
   # 11111 -Decimal to Hex
   $ 1111 - Hex to Decimal
   + 1111 2222 - Hex Addition
   - 1111 2222 - Hex Subtraction
   : 1111 22 33 44 55 66 77 88 - Memory Modify
   ; 1111 2222 33 44 55 66 77 - Register Modify
   . 1111 11 [22[33]]XXXX - Disassembly Modify
```