KRACKER JAX

REVEALED - VOL III

TABLE OF CONTENTS

INTRODUCTION.....PAGE 2

GEOS v2.0.....PAGE 5

SNAPSHOT GEOS v1.3,2.0.....PAGE 12

DEATH SWORD V1/V2.....PAGE 15

RAD WARRIOR.....PAGE 17

SPIDERBOT.....PAGE 19

TRACKER.....PAGE 21

STARGLIDER.....PAGE 24

WWF WRESTLING.....PAGE 27

MAVIS BEACON TYPING.....PAGE 29

1942/GHOSTS & GOBLINS.....PAGE 31

L.A. CRACKDOWN.....PAGE 33

V-MAX v1.? INTRODUCTION.....PAGE 38

INTO THE EAGLE'S NEST.....PAGE 40

PAPERBOY.....PAGE 42

XEVIOUS.....PAGE 44

PROTECTION SCHEME #1.....PAGE 46

PROTECTION SCHEME #2.....PAGE 50

HACKER'S UTILITY KIT INSTRUCTIONS.....PAGE 54

DISK DOCTOR COMMANDS.....PAGE 67

BOOKS FOR FURTHER READING.....PAGE 67

LIMITED WARRANTY.....PAGE 67

ADDITIONAL PRODUCT INFORMATION.....PAGE 69

INTRODUCTION

<< 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 purchased Kracker Jax Revealed Vols I & II and that you've performed many of the procedures in those manuals. 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 M/L programmer, but you MUST have a cursory knowledge of M/L 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 techniques 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 M/L. 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 v2.0. 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 section. 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.

Lastly, for those of you who don't have many of the programs we've worked on in this manual, we have an answer. Rent-A-Disk, located in West Virginia, has consented to stock multiple copies of these titles. They are a mail order disk rental outlet and can fill your needs very nicely. Please understand that neither Rent-A-Disk or Kracker Jax advocate piracy in any form. We suggest this company only for educational purposes, and, if you do work on a title you happen to like, they will be glad to sell it to you on a discount basis. Please do NOT write to their diskettes and please do NOT retain a copy of the program after breaking it. More details on Rent-A-Disk can be found in the back of this manual.

This is definitely the last volume in the KJ REVEALED series, but it's definitely NOT the last word from us on copy protection. If there is enough interest, we might consider publishing a quarterly newsletter to fill in the gaps. We propose to publish approximately 6 tutorials every quarter. We would have to charge $13.50 per issue, which would include shipping and handling (No C.O.D.). Ownership of Kracker Jax Revealed III or the Hacker's Utility Kit would be assumed. Again, drive knowledge would be mandatory because we'd be in advanced user territory. Write to us, and voice your opinion.

<< 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 SOFTWORKS : 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 $64A0 to "JMP $64A0". 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 $64A0 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 first segment is loaded into $9000. 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 KERNAL. 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 first-time 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 re-activate "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 "GMON'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 $13/$0D and is stored at $0528/$0529 for use by other subroutines.

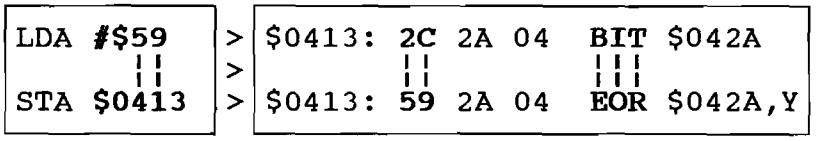
Let's stop here. Using a sector editor or "GMON" drivemon, look at the first sector of the GEOS KERNAL. This is a block of track/sector pointers (GEOS VLIR file). Our GEOS shows 3 file chains starting at $13/$0D (!!!), $14/$11, and $14/$0F. WRITE THESE DOWN! (Your GEOS may have slightly different values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



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 the eight header bytes it jumps to the 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.

$04BA: JSR $0502 (see above) to check this tail gap.

$04BD: Count $0A (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 done, otherwise branch back to $04B0.

$04C8: We've passed the protection check. Read and send the third and last segment at track/sector $14/$11 (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 BIT/EOR address at $0413 to look there instead, we can freely alter the protection check. Change the LDY $1C00 at $0502 to read JMP $04FD 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 - $07FF: 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 start. 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 SYSTEM 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 to look at the header and tail gap bytes we discussed above. They can be found at position $0A 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 $DD00 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 **SSS** 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 DOS 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 V1/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. Obviously, 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 than load 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 $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 - $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 JMP 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 $C000 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 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. 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 $11/$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 = $1B)**) 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 "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 $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 $C000 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 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. 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 $C000 monitor and the SB boot file, 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. Most 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 (T/S) 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 $7EF1 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 $C000 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 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. 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 $09AB. 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 $90AE 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 $0E. The drive routine stores these 3 bytes into job queue $01, producing a read ($80) of track 40 ($28) /sector 14 ($0E) 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 $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 - $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 TK 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 $010E. 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 $E0 (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 - $06FF 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
, 06EA  STA $01
, 06EC  LDA $01
, 06EE  BMI $06EC
, 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 $90AB. 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 $90AE 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 $0E. The drive routine stores these 3 bytes into job queue $01, producing a read ($80) of track 40 ($28)/sector 14 ($0E) into drive memory $0400. The computer waits for this read to be completed, 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 - $0A). 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 $010E. 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 $E0 (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 - $06FF 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
, 06EA  STA $01
, 06EC  LDA $01
, 06EE  BMI $06EC
, 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.

< < < MicroLeague : WWF 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, U1, etc...". You should find a "U1" (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 $0A (10). If the $73 byte is found or .Y equals $0A, 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 06DF CPY #$02      ;this was "CPY #$0A"
, 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 "U1" command text is located at $03D6) then a JMP to $0F00 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 $0F00. The routine from $0F00 - $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 0C3C 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 MBTT**, and initialize the drive. Use the drive's job queue to read in T/S $24/$10 (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/$0E) 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/$0E) from $24/$10 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.

< < < CAPCOM : 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 $02BB - $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 $CC00.

Load "1.0", which resides from $C900 - $D000. The entry point at $CC00 JMPs to $CC71, which calls a subroutine at $C900. This subroutine sends protection check code to the drive. If you look at memory in the range $CA00 - $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 $C9E0. 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 $CA31 to read:

```
A CA31 LDA #$FF
, CA33 STA $CFFF
, CA36 RTS
```


Now execute the loader (G CC00). 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 $CC71 at $CC00 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 MON1000, 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 $CA00.

Load the "(C) 1988 EPYX" file. It resides from $C74F - $CA19. Disassemble from $CA00, 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 $C9F1. 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 $CA16. 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 $CA00. 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

(SYS4096). 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 $C9F1 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 "MON1000", 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 $0A (**T**rack) and $0B (**S**ector). 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 $E0 (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 $06FF by the routine. At $01AD, a JSR $03BE 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 $01AD. To continue execution of the code, place a 'JMP $0160' at $0500 and place $E0 in drive job queue $02. After the drive motor shuts down, disassemble the code at $0300 - $06FF.

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 ($0F - $12). Using the drivemon, place the following bytes into job queue $06 - $0D: 12 0F 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 $E0 in drive job queue $02 to decrypt the code. Transfer the decrypted code from $0160 - $01FF 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: LDX #0          ; move code to a safe place
], 0503: LDA $0500,X
], 0506: STA $0700,X
], 0509: INX
], 050A: BNE $0503
], 050C: JMP $070F       ;continue execution
```

Transfer the code from $0500 - $05FF to $0700. Continue entering code at $070F:

```
]A 070F: LDX #$0D        ;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 $00,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 $01AD       ;fire up the fast loader

]:0740 80 80 80 80 00 00 12 0F    ;DOS job queue data
]:0748 12 10 12 11 12 12
```

Transfer the code at $0700 - $07FF 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 $11/$0C, totals 106 blocks!

Load the BAM into drive memory $0500. Use the monitor to enter the following data:

```
]:0504 15 FF FF 1F 15 FF FF 1F
]:050C 15 FF FF 1F 15 FF FF 1F
]:0514 15 FF FF 1F
```

This makes tracks 1 - 5 available. Now fill $0518 - $058F with $00 to allocate the rest of the available sectors. To free-up the sector at $11/$0C enter:

```
]:0544 01 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 $11/$0C. 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 02EC 02A7  ;copy load routine to $02A7

A 02ED 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 #$F0   ;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 abnormal 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 **minor 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 so 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 T/S $12/$0D into buffer $0700 by entering:

```
:000E 12 0D
: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 - $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 #$01
], 0790 JMP $F969
```

This will return control to the drive-mon when the code has finished execution. To execute it, enter

```
:0004 E0
```

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/$0D like we did above.

Directory sectors $04, $07, $0A, and $0C will contain the code from $0300 - $06FF. Beginning at $0700, re-write the drive code as follows:

```
0700 SEI        ;disable interrupts
0701 LDX #$0D
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 0A 12 0C
```

Make sure all of the original code from $078E - $07FF 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
]:000E 12 0D
]:0000 90 90 90 90 90
```

There is the third level protection present on this title. To remove it, enter the following:

```
]:0006 18 0D
]:0000 80
]:0362 6B
]:036E 45
]:0000 90
```

That's all there is to it! Enjoy your backup copy.

< < < MINDSCAPE: 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/$0D into buffer $0700 by entering:

```
:000E 12 0D
: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 - $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 #$01
], 0790 JMP $F969
```

This will return control to the drive-mon when the code has finished execution. To execute it, enter

```
:0004 E0
```

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/$0D like we did above.

Directory sectors $04, $07, $0A, and $0C will contain the code from $0300 - $06FF. Beginning at $0700, re-write the drive code as follows:

```
0700 SEI        ;disable interrupts
0701 LDX #$0D
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 0A 12 0C
```

Make sure all of the original code from $078E - $07FF 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
]:000E 12 0D
]:0000 90 90 90 90 90
```

There is the third level protection present on this title. To remove it enter the following:

```
]:0006 19 01
]:0000 80
]:035C 60
]:0368 6F
]:0000 90
```

That's all there is to it! Enjoy your backup copy.

< < < MINDSCAPE : 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 so 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/$0D into buffer $0700 by entering:

```
:000E 12 0D
: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 - $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 #$01
], 0790 JMP $F969
```

This will return control to the drive-mon when the code has finished execution. To execute it, enter

```
:0004 E0
```

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/$0D like we did above.

Directory sectors $04, $07, $0A, and $0C will contain the code from $0300 - $06FF. Beginning at $0700, re-write the drive code as follows:

```
0700 SEI        ;disable interrupts
0701 LDX #$0D
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 0A 12 0C
```

Make sure all of the original code from $078E - $07FF 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
]:000E 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 - $1F
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 drive, 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 $0x, $1x, $4x, $5x, $8x or $Cx;
; (x = any hex number from $0 - $F)
; The following code must be written to
; drive memory $0500 and can be executed
; from BASIC with the following statement:
;
; OPEN 15,8,15,"UC:"+CHR$(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
     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  $07
     lda  #$B0         ;seek track/sector
     jsr  waitjob
     lda  #$80         ;read track/sector into $0300
     jsr  waitjob
     lda  newid        ;setup new data block id
     sta  $47
     lda  #$90         ;write tr/se 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
     lda  $00          ;wait for job to finish
     bmi  wjloop
     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 for 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,"U1:2 0 01 00":REM READ TRACK/SECTOR 1/0
50 GET#15,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 $c000

       lda #$00 ;open cmd channel
       jsr $ffbd ;SETNAM
       lda #$0f
       ldx #$08 ;to drive 8
       tay
       jsr $ffba ; SETLFS
       jsr $ffc0 ; OPEN
       lda #$01     ;open buffer channel
       ldx #<pound
       ldy #>pound
       jsr $ffbd
       lda #$02
       ldx #$08
       tay
       jsr $ffba
       jsr $ffc0
       jsr $ffcc  ;clear channels CLRCHN
       ldx #$0f   ;output "u1" command
       jsr $ffc9  ;CHKOUT
       ldy #$00
loop   lda u1cmd,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    "#"
u1cmd   .byt    "u1: 2 0 01 00"
        .byt    $0d
```

This M/L routine can be stored at $C000 (49152) and called from BASIC as follows.

```
10 OPEN15,8,15,"I":CLOSE15
20 SYS49152
30 IF PEEK(251)<>ASC("2") AND PEEK(252)<>ASC("2") THEN PRINT
   "DATA BLOCK NOT FOUND!":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 involves 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 (GCRED)** from the Hacker's Utility Kit. With your work disk in the drive, input 1 for 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 bytes 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 "1" 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 only 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 every software-only nybbler stops 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 physical 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 K.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  $07
     lda  #$E0      ;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  $1c00     ;The header block was found so wait
                    ;sync mark preceding the data block
     bpl  syncloop
sloop1
     bit  $1c00     ;got a sync, now wait for it to end
     bmi  sloop1
     lda  $1c01     ;throw away the sync image
     clv
     ldx  #$01      ;set up .x/.y to count $0145 bytes
```

```
     ldy  #$45       ;($0000 - $0144) to the end of the data
                     ;block
dataloop
     bvc  *          ;wait for data byte ready
     clv             ;clear ready flag
     lda  $1c01      ;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  $1c01      ;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.

K R A C K E R J A X P R E S E N T S

THE HACKER'S UTILITY KIT

Programmed by:
Mike Howard / Joe Peter
Paul Rowe / Jeff Spangenberg
Designed by: Les Lawrence

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 Hacker's Utility Kit disk in your disk drive. 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 error in 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.
M : 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.
F1: 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 $04FF, 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 U/D/R/L: 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 ".." 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 U/D 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.

Notepad Feature: At times when using the GCR 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.

Nibble 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 nibbler, that nibblers are limited in their abilities. The following key strokes represent the user options.

F1/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).
S/D : 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:

F1 : 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 : $8D,$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:

F1 : Directory of disk in drive.
F3 : Prompt to reboot Hacker's Utility Kit main menu.

Kracker Jax Parameter/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.

F1: 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 Nibbler. We recommend the data copier in most cases. Occasionally only a Nibbler 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-Mon 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=$X000" is highlighted will also increment the monitor to the desired Hex address.

F1 : 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",8,1 to autoboot other save files. The op-codes listings will be saved as "OPS". The monitor will be saved as "X0" .
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).
HOME : 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,C000 (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+1
Example : S "FILENAME",08,C000,D001
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 $XXXX (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 O not a zero. O 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. O 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.
$36 = Bank out BASIC.($A000-$BFFF)
$35 = Bank out Kernal & BASIC.
$30 = Bank in RAM under $D000.
$31 = Bank in character ROM under $D000.

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.
   F1/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 Logger

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:

F1: 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.

< < < DISK DOCTOR COMMANDS > > >

@ = Change Byte	t = Text Mode
+ = Scan Forward	- = Scan Back
n = Next Block	N = Previous Block
j = Jump to Link	J = Previous Link
b = New Block	B = Last Block
r = Rewrite Block	c = Copy Block
s = Swap Disks	p = Print Block
Q = Quit	

Clear = Renew the current sector display.
Home = Position the cursor over position 0.
Cursor Keys = Position the cursor R/L or U/D.
Return = Position the cursor over the first byte of the next line.

<==========>

< < < BOOKS FOR FURTHER READING > > >

K J Revealed Vol I .....(c)K.J.P.B.

K J Revealed Vol II .....(c)K.J.P.B.

Commodore 1541 Disk Drive Owners Manual .....(c)C.B.M.

Commodore 64 Programmer's Reference Guide .....(c)C.B.M.

1541 Internals.....(c)Abacus Software

Inside Commodore Dos .....(c)Reston Publishing Co.

M/L For Beginners .....(c)Compute! Books Publication

Mapping The Commodore 64 .....(c)Compute! Books Publication

Program Protection Manual (Vol 1 & 2) For The C-64.....(c)CSM Software Inc.

Geos Programmer's Reference Guide(tm)...(c)Berkeley Softworks

< < < LIMITED WARRANTY > > >

We have attempted to ensure that this manual, along with the software, works as specified. We would appreciate receiving notice of any errors you may find. Neither the author nor any distributor of this product will be liable for any damages which may be a result of errors or omissions, use

or misuse of this product. Should there be any defects in the software provided in this package, we will replace the defective diskette within 90 days from the date of purchase. You will find our software unprotected, and are encouraged to make a backup for your own use.

Distributed by:

Software Support Int.
2700 NE Andresen Rd #A-1
Vancouver Wa 98661
(206) 695-9648

<==========>

Important Notice: This instructional material has been written for educational and archival purposes only. You are advised that the Federal Copyright Law allows you the right to back up, for archival purposes, any computer program you have purchased. Any other use could be unlawful and is not advised nor encouraged. By using this product, you agree to be bound by the terms of this notice.

THE FOLLOWING TITLES ARE REGISTERED TRADEMARKS OF THE FIRMS LISTED BELOW

Geos.....Berkeley Softworks

Starglider.....Rainbird

Tracker.....Rainbird

MicroLeague Wrestling.....MicroLeague Sports

Rad Warrior.....Epyx

Death Sword.....Epyx

Spiderbot.....Epyx

L.A. Crackdown.....Epyx

1942 .....Capcom

Ghosts & Goblin.....Capcom

Mavis Beacon.....Software Tools

V-MAX!.....Harald Seeley

Paperboy.....Mindscape

Into The Eagle's Nest.....Mindscape

Xevious.....Mindscape

< < < PRODUCTS OR SERVICES FOR YOUR CONSIDERATION > > >

Maverick

Those of you who have been purchasing our products know that we take our work very seriously. We strive for excellence in our software. Maverick represents our finest effort ever. It is more than just a copier. Those of you who have an intense interest in program protection (as we do) will find Maverick an essential friend. We have created a set of tools that aid us in our everyday work, and placed them in a format that we know you'll find enjoyable to work with.

Super Snapshot 64 V4

What are we doing recommending a snapshot type utility to you? We know you'd rather hand break programs if at all possible. What we'd like to let you know about is the M/L monitor available in this cartridge. This monitor has proven to be an invaluable tool time and time again. There is no place in memory (now with Drive-Mon) that you can't access. Even screen memory is accessible. All this from within a running or booting program. Let the program boot until you suspect protection check and stop the program cold. Today's protection is being placed in areas of memory that are increasingly difficult to access. This monitor "reveals" all.

Other features in Super Snapshot are: 1541/71/81 Turbo DOS, DOS Wedge commands, reasonably priced upgrade policy, programmed function keys, a resume feature that allows a program to be stopped, examined and restarted in most cases, totally invisible cartridge when disabled, supports multiple drives, screen dumps to printer as well as disk, and a C-128 mode switch available to make the cartridge invisible to the C-64 mode on the C-128. Allows you to use C-128 software without removing the cartridge.

Super Snapshot is available from Software Support Int. as well as many retailers across the country. This is one purchase you won't regret.

Rent-A-Disk

Well, now you own Kracker Jax Revealed Vol III and are excited about a specific protection scheme. What?! You don't own that particular program? What to do? Well, you could buy it if you were SURE you wanted to keep it anyway; or, you could borrow it from a friend, if you knew someone who had that particular program. OR, there is another simple solution to the problem.

Program renting. Our friends at Rent-A-Disk in West Virginia have consented to have the commercial programs we have covered in this manual in stock and available to you. Rent-A-Disk allows you to rent a title for your examination for a period of two weeks, including turn around time. During this time you may try the title before you buy. If you find that the title is what you want, you may purchase that title at a very attractive price.

Neither Kracker Jax or Rent-A-Disk condones or encourages piracy in any form. We recommend rental software strictly for trial or educational purposes. We realize you will have access to copyable software by using this manual, but we urge you to let your conscience be your guide. Please remember that software piracy is a crime. If you rent a piece of software and you like it, BUY IT. Support the programmers who worked so hard to bring that package to you.

For details on Disk Rentals, call or write: Rent-A-Disk at Frederick Bldg. #345, Huntington, WV 25701. (304)529-3232 9am-10pm M-TH 9am-5pm Fr-Sa.