```
TRICKY DICK  V-1.0                          BY T TSE

D5AA96 Y DEAA   DOS   SL-6 T-11 <00> VOL
D5AAAD Y DEAA   3.3   DR-1 S-OF <--> 254
D5AAAD O DEAAEB       PR-0 <*L> <62> <H>

00: 00 11 0E 00 00 00 00 00 :@QN@@@@@:
08: 00 00 00 12 0F 82 C3 A0 :@@@ROBC :
10: C9 A0 C1 A0 A0 A0 A0 A0 :I A      :
18: A0 A0 A0 A0 A0 A0 A0 A0 :         :
20: A0 A0 A0 A0 A0 A0 A0 A0 :         :
28: [illegible] 02 00 10 0F :     B@PO:
30: 82 C2 CF CF D4 C5 D2 A0 :BBOOTER  :
38: A0 A0 A0 A0 A0 A0 A0 A0 :         :
40: A0 A0 A0 A0 A0 A0 A0 A0 :         :
48: A0 A0 A0 A0 A0 A0 A0 07 :         :
50: 00 10 [illegible] C4 :@PHDLOAD:
58: C5 D2 A0 A0 A0 A0 A0 A0 :ER       :
60: A0 A0 A0 A0 A0 A0 A0 A0 :         :
68: A0 A0 A0 A0 A0 A0 A0 A0 :         :
70: A0 A0 03 00 10 05 84 D0 :  C@PEDP:
78: C9 C3 D4 D5 D2 C5 A0 A0 :ICTURE   :

DATA: NORMAL HEX     ALL COMMANDS:C
```

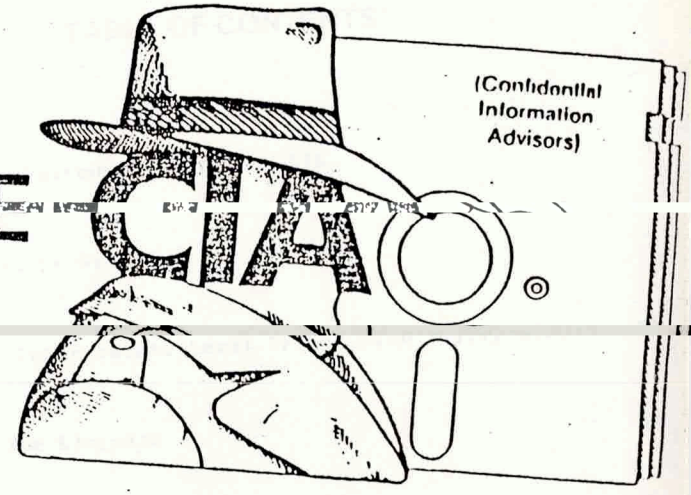
```
TRICKY DICK  THE LINGUIST                   BY T TSE

D596AA Y AADE   DOS   SL-6 T-11+HALF VOL
D6ACAF N 0000   3.3   DR-1 S-OF <--> 254
D5AAAD O DEAAEB       PR-1 <IL> <44> <H>

4000-96969696 96969696 96969696 96969696
4010-96969696 96969696 96969696 96969696
4020-96969696 96969696 96969696 96969696
4030-96969696 96969696 9696DEAA EBEADFFF
4040-FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
4050-D5AA96FF FEAABBAF AFFAEADE AAEEFBFF
4060-FFFFFFFF FFFFD5AA ADA7ACAD AF9D9696
4070-96969696 97EFEFD9 9BD99AB6 B6969696
4080-96969696 96AEAE96 A7A796A7 9DAEA7AE
4090-AEA79696 96979796 F2EFD6EF B49BB6F2
40A0-B596B4D6 96D6B496 F7A79D9D 96969D9D
40B0-96969696 96969696 9B9B96B4 B496DADA
40C0-9DA69B96 96969696 96969DA6 DAB4BDBF
40D0-BFBDBD96 96969696 96969696 96969696
40E0-96969696 96969696 969696DF 969DA6DA
40F0-[illegible]

4053: FE 11 OF E0    ALL COMMANDS:C
```

Edmund Ma



5 sophisticated disk utilities with which you can:

* edit normal or protected disks
* quickly find and recover any intact file, however badly the disk is corrupted
* list programs directly from any disk - protected or not
* examine textfiles directly from any disk - protected or not
* analyse the formatting of normal or protected disks
* decrypt commercial software - or encrypt your own
* rapidly auto-search normal or protected disks for anything you like
* understand & use the latest copy protection methods
* use your Apple as a powerful document retrieval system
* make use of an exhaustive knowledge of disk lore

Includes "THE CIA FILES"

INDEX

CHAPTER ONE - AN INTRODUCTION TO THE CIA 1

TRICKY DICK 1
THE LINGUIST 1
THE TRACER 1
THE CODE BREAKER 2
THE TRACKER 2
THE CIA FILES 2
THE CIA MODULES 2
GOLDEN DELICIOUS GOES BARE 2
HOW TO USE THIS BOOK 3
DROP US A LINE 3

CHAPTER TWO - TRICKY DICK 5

A FIRST WORD ABOUT DISK DATA 5
THE RAW NIBBLE DUMP 5
THE RWTS READ 5
TRICK DICK'S INSTRUCTIONS 7
THE TRICKY DICK DISPLAY 7
HELP SCREEN (/ OR ?) 7
SELECT DOS VERSION (^D) 7
SLOT DRIVE AND DEVICE SELECT (^O) 7
TRACK AND SECTOR SELECTION (. < > ARROWS) 8
READING A SECTOR (^R) 8
VOLUME NO. 8
CURSER MOVEMENT (IJKM AND ^I^J^K^M) 8
EDITING SINGLE BYTES IN THE DISPLAY 8
DATA ENTRY MODES (^A ^ ") 9
THE DATA DISPLAY 9
FLIPPING DATA DISPLAYS (^F) 9
SECTOR FILLING (^Z ^X) 10
DISASSEMBLING SECTOR DATA (L) 10
LISTING APPLESOFT AND INTEGER CODE (^L, L) 10
WRITING TO THE DISK (^W AND Y) 10
ERROR MESSAGES 11
DEALING WITH NON-STANDARD SECTOR HEADERS (^S) 11
PRINTING HARD COPY FROM TRICKY DICK (^P, P) 12
MODULE CHECK-OUT (SHIFT M) 12
EXITING TRICKY DICK (RESET, ^C) 12
JUMPING TO A MODULE (^E) 12
THE TRICKY DICK TUTORIAL 13
THE VOLUME TABLE OF CONTENTS 14
THE BIT MAPS 15
FREEING UP TRACK $23 17
GETTING EXTRA SPACE ON TRACK $02 18
SNATCHING SPACE FROM THE CATALOG TRACK 20
GETTING RID OF DOS 21
REPAIRING A CLOBBERED VTOC 22
UNDELETED PROGRAMS 22
FILE TYPE FLAGS 24
ELIMINATING HIDDEN CONTROL CHARACTERS 25
FINDING AND CHANGING A BINARY FILE'S ADDRESS AND LENGTH 25
LISTING APPLESOFT OR INTEGER PROGRAMS
DIRECTLY FROM THE DISK 26

CHAPTER THREE - INTERMEDIATE TRICKS WITH TRICKY DICK 29

AVOIDING DOS LANGUAGE CARD CLOBBER 29
SWITCHING THE HELLO FILE 29
USING A BINARY OR EXECABLE HELLO FILE 29
CHANGING THE "DISK VOLUME" CATALOG MESSAGE 31
PUTTING HEADINGS ON THE CATALOG TRACK 31
HIDING THE HELLO FILE ON THE CATALOG 33
CHANGING DOS ERROR MESSAGES 33
SOME IDEAS FOR ADVANCED PROGRAMMERS 34
MOVING CLOSER TO THE DISK 35

CHAPTER FOUR - THE LINGUIST 36

HOW TO INSTRUCT THE LINGUIST 37
THE LINGUIST NEEDS TRICKY DICK 37
GETTING FROM ONE TO THE OTHER (^E,^C) 37
THE HELP SCREEN (/ or ?) 37
THE LINGUIST'S DATA DISPLAY 38
ENTERING COMMANDS 38
TRACK SELECT (< : >) 38
SEEKING THE DISK ARM TO TRACK $00 (^S) 38
READING A TRACK (^R) 38
PAGING THROUGH THE BUFFER (ARROWS) 38
JUMP TO THE BEGINING OR END OF THE BUFFER (^B ^N) 38
CURSER MOVEMENTS (I,J,K,M AND ^I^J^K^M) 38
DECODING THE ADDRESS FIELD INFORMATION (CURSER CONTROLS) 39
CHANGING THE DECODING MODE (^D) 39
TRANSLATING A SECTOR (^T) 39
CHANGING THE DRIVE NUMBER 40
GETTING HARD COPY FROM THE LINGUIST 40
THE APPLE DISK LANGUAGE SCHOOL - A LINGUIST TUTOTIAL 40
DATA STORAGE AND THE BUFFER 42
THE SYNCH BYTES 43
THE ADDRESS FIELD 44
DATA ENCODING CONSTRAINTS 45
THE 4&4 ENCODING TECHNIQUE 46
HOW TO TRANSLATE THE ADDRESS FIELD HEADER 47
LOGICAL VS. PHYSICAL SECTORS 48
THE DATA FIELD 50
DIGGING THROUGH THE RAW DATA 51
DOS 3.2 AND 3.3 ENCODING TECHNIQUES 52
MORE ON FOUR 'N FOUR 53
GOODIES OR GARBAGE? 54
TRANSLATE RAW DISK DATA WITH THE LINGUIST 54

CHAPTER FIVE- THE SECRETS OF SOFTWARE 58

PROTECTION 58
FIRST STEPS IN READING PROTECTED DISKS 59
USING THE LINGUIST AND TRICKY DICK IN TANDEM 61
HOW TO EDIT HALF TRACKS 64
EDITING DISKS WITH WEIRD TRACK NUMBERS 65
PROTECTING YOUR OWN DISKETTES 66
[illegible] 66
[illegible] 67
[illegible] 69
WHAT WAS ON TRACK $22 72

CHAPTER SIX - THE CODE BREAKER 75

UNRAVELING THE RWTS TRANSLATE TABLES 75
THE NIBBLE TRANSLATE TABLE 75
THE BYTE TRANSLATE TABLE 76
DOS 3.2 TRANSLATION 77
INSTRUCTING THE CODE BREAKER 79
TRICKY DICK PREPARATION 79
THE CODE BREAKER DISPLAY 79
THE HELP SCREEN (/ OR ?) 79
EDITING THE TRANSLATE TABLES 79
SAVING THE CHANGED TABLE (^S) 79
ERASING A MISTAKE (^R) 79
RESTORING THE STANDARD DOS TABLE (^D) 79
THE CODE BREAKER TUTORIAL 80
ENCRYPTING YOUR PROGRAMS 80
HOW TO EDIT ENCRYPTED SOFTWARE 84
THE D5 - D6 SWITCH 86

CHAPTER SEVEN - THE TRACER 87

THE TRACER'S INSTRUCTIONS 87
TRICKY DICK AND THE TRACER 87
GETTING READY FOR THE TRACER 87
THE MENU 88
VERIFY FORMATTING 88
T&S LISTS 88
CATALOG SECTORS 89
VTOC 89
STRINGS 89
SPECIFYING THE RANGE 89
MENU VERIFICATION 91
EDITING THE MENU 91
JUMPING OUT OF THE MENU 91
HOW THE TRACER TRACES 92
HOW THE TRACER FLAGS WHAT IT HAS FOUND 92
CONTINUEING THE SEARCH 93
HANDLING THE SECTOR OVERLAP 93
ALTERING THE SEARCH RANGE FROM TRICKY DICK 94
WHEN THE SEARCH IS COMPLETED 94
ABORTING A SEARCH 94
TRACER ERROR HANDLING 95
DEFAULTING TO THE PREVIOUS SEARCH ARGUMENTS 95
CHANGING THE SECTOR MARKS 95
TRACER SCAN TIMES 96
THE TRACER TUTORIAL 96
TRACING LOST FILES WITH THE TRACER 102
USING THE TRACER FOR TEXT RETRIEVAL 106
SEARCHING DISKS WITH MODIFIED DOS MARKS 107
SEARCHING DISKS WITH MODIFIED SECTOR NUMBERS 109
SEARCHING HALF - AND STRANGELY NUMBERED TRACKS 109
FINDING THE BYTE AND NIBBLE TRANSLATE TABLES 110

CHAPTER EIGHT - THE TRACKER 112

MAKING THE TRACKER TRACK 112
THE TRACKER STANDS ALONE 112
THE TRACKER PREFERS 3.3 112
GETTING THE TRACKER ON THE DOS TRAIL 113
THE TRACKER'S MENU 113
THE TRACKER'S DISPLAY 114
THE TRACKER'S USE OF RESET 115
THE TRACKER LIVES ON 115
THE AMPERSAND AND CONTROL-Y OPTIONS 115
THE TRACKER TUTORIAL 116
THE AMPERSAND AND CONTROL-Y COMMANDS 118
ON THE TRAIL OF THE WILD DOSSIT 119

APPENDIX A 120

FUNNY NUMBER SYSTEMS 121
MAKING IT WITH BINARY 122
SOME NECESSARY HEX AND BINARY JARGON 125
BACKWARD ADDRESSES 125
EXCLUSIVE ORING 126
HEX-BINARY-DECIMAL CONVERSION CHART 127

CHAPTER ONE — An Introduction to The CIA

Welcome to the world of the CIA!

The CIA is a group of 5 powerful disk espionage utilities that will allow you to investigate, analyse, edit, protect, decrypt, encrypt, locate, list, translate, trace, verify, and examime programs and datafiles on normal and protected disks. It contains many new features never before offered to the Apple computing public. What's more, much of the disk lore that appears in this book has never found its way into print until now.

Here's what the CIA lineup looks like.

TRICKY DICK is an all-purpose disk editor with a difference - it can be used both on normal and most protected diskettes. Not only that, it enables you to read in a sector directly from a disk and list the sector's contents on your screen or printer in Integer or Applesoft BASIC, assembler, hex, and ASCII. Among may other uses you will find Tricky Dick's tricks invaluable for patching and customizing your own disks, protecting your software, and reading and altering normal and most protected programs.

THE LINGUIST dumps a track of raw nibbles directly from any disk - protected or not. It also decodes address data found on the disk and translates raw disk nibbles into hex. The translated nibbles can then be listed by Tricky Dick in BASIC, assembler, and ASCII. This means that The Linguist makes it possible for you to list the programs and textfiles from any disk, normal, corrupted, or locked. You can use it to have a closer look at your favorite games, examine disk formatting, and recover lost programs no matter how badly the disk they're on is corrupted.

THE TRACER automatically searches normal and most protected disks sector by sector for any 1 - 6 strings you specify in ASCII or hex. It will also sniff out the VTOC, catalog sectors, and track/sector lists on normal and protected disks. You can choose one, some, or all of its 9 search options, and The Tracer will carry them out simultaneously, scanning an entire disk in 19 - 135 seconds. When it finds something you're looking for it jumps back in Tricky Dick and places a cursor on the item it's located. This allows you to thoroughly search and edit a disk in record time.

THE CODE BREAKER makes use of a special table in DOS 3.3 and 3.2 to decrypt the most popular "secret code" used to hide commercial programs from prying eyes. What's more, this book's chapter on The Code Breaker tells you how to protect and encrypt your own files using the same method.

THE TRACKER closely shadows the disk arm, reporting its every move as DOS LOADS, SAVES, RENAMES, or does anything else to a disk. The Tracker leaves on your screen or printer a list of every track and sector visited, and every read or write operation carried out during any disk access. You can use this utility to find out exactly where a disk is crashing, make a permanent record of your files, and learn more about how DOS works.

THE CIA FILES is this book. It consists of well over 50,000 words, and contains a series of extensive and carefully worked out tutorials designed to turn you into a disk expert. As you learn to use the CIA utilities, they also become your personal guides, leading you step by step from a beginner's knowledge of the disk to that of a pro. Here you'll find a wealth of information relating to disk repair and file recovery, DOS patches, copy protection, disk formatting, program encryption, and many other vital topics. Much of the material appears here in print for the first time.

The CIA Modules

In order to make the CIA as flexible as possible, we've adopted a modular approach which works like this. Tricky Dick is what you might call a chief executive program, in that it controls the operations of the other CIA utilities. They are accessed through Tricky Dick, and considerable information gets passed back and forth as they work a disk over.

The first step in using the CIA is to BRUN Tricky Dick. Then when you need to call upon one of the other utilities, you simply BLOAD it. A single keystroke transfers control to the utility you have just summoned and enables it to carry out its work. During certain oprations, the module in question will pass data to Tricky Dick for special processing. For example, when The Tracer finds a string it's looking for, it automatically jumps back to Tricky Dick, displays the string, and places Tricky Dick's cursor on the first character. Now Tricky Dick is in charge and needs only a few keystrokes from you to edit the string and/or the other data in the sector where it resides.

The only exception to this system is The Tracker who works alone. Complete instructions for using Tricky Dick in conjunction with the modules are given in this book.

Golden Delicious Goes Bare

As you may have gathered from the previous discussion, The CIA programs are NOT copy-protected in any way. However, in order to give you some hands-on practice using CIA on locked disks, I have "protected" track $22 and placed a "half-track" between tracks $20 and $21. Nevertheless, all the CIA programs are copyable, listable, and modifiable. In fact, this book contains a number of special patches to customize Tricky Dick and some of the modules.

This means that before doing anything else you should make at least two backups of the CIA disk. You'll need to use FID on your Apple System Master disk for this purpose (COPYA won't work because of tracks $22 and $20 1/2). After reading part of this book, you will also be able to copy track $22 across to your backups.

How to Use This Book

Each of the CIA programs has at least one chapter devoted to it. Near the beginning of each program's chapter you'll find a concise set of instructions for using that program. If you are an experienced disk person, this will be enough to get you going. However, if you're closer to being a beginner, the way to use each chapter is to try out its instructions in order to get familiar with their effect on the program in question. The next step is to move on to the tutorial which directly follows the instructions and work your way through it.

As you work through the tutorials from beginning to end of the book, you'll find that the material becomes more and more advanced. I set it out this way so that no matter what your level of expertise, you could find a starting point in this book from which to progress to greater knowledge about the disk.

If you've already got some experience, you may find that some of the material in the first couple of chapters is not new to you, having appeared elsewhere in the literature. Its inclusion was necessary so that even an Apple newcomer would have a place to start. So if some of the tricks described in the beginning of this book look familiar, take heart and move on to the next chapter - I'm pretty sure that sooner or later you'll find plenty of things to get your teeth into.

Drop Us a Line

Our goal is to produce good, sound, usuable software for a wide range of interests, and at a low cost. In order to do that effectively, we have to know about your particular needs and interests. So if there are any improvements you'd like to see in the CIA, any similar types of software you'd like see written, or any bugs you spot in the programs or this book, let us know. Due to our hectic schedules, I can't promise you a letter in return, but you can rest assured that we'll take your complaints, compliments, and ideas seriously when we're putting together future projects.

If and when any CIA updates emerge, we'll write to you and offer them to you at a premium price. We'll also keep you posted about any other software we've got going. But in order, to keep this sort of contact with our customers, we need to know about you. To make sure we do, if you didn't get this program directly from Golden Delicious Software, drop us a post card with your name and address for our records. This is not a ploy to make you a "registered owner" of our software as many program

publishers attempt to do, but rather to enable us to keep you posted on any exciting new developments in our product line as soon as they break.

Here's hoping you get many hours of fun out of the CIA!

CHAPTER TWO — Tricky Dick

Your first CIA contact is Tricky Dick. So get out a blank floppy and get ready to enter a new domain of disk doctoring.

But first, if you have not already done so, make a couple of copies of the CIA disk, using FID, and write protect them. Then put the original away and boot up one of the copies. When the CIA menu appears, select option £1. I know you're anxious to get better acquainted with Tricky Dick, but just before you start, you might be interested in hearing....

A FIRST WORD ABOUT DISK DATA

If you are already familiar with the differences between a raw nibble dump directly from the disk and a disk read using RWTS, you can skip this explanation and jump to "Tricky Dick's Instructions" in the next section. Up to now, these two methods of reading a disk seem to have been documented in a manner comprehensible only to those progammers capable of writing machine code in their sleep. This prompted me to start things off with a detailed discussion designed clarify disk examination for you and enable you to get the maximum use and enjoyment from the CIA.

The first thing you need to know is that when you SAVE or BSAVE any program, DOS totally alters the program code just before writing it to the disk. The next time you load or run that program, DOS changes it right back again to its original form. The user, naturally enough, never notices this process, and it is carried out in a manner guaranteed not to interfere in any way with the running of your Apple.

Why should DOS go to all this extra trouble whenever any information is put on or taken off the disk? The answer lies deep in the innards of your machine. The Apple's hardware (and that of many other micros) has some innate limitations which restrict the range of byte values that can be allowed to pass between the machine and its disk drives.

Now if you type "CALL -151" into your machine and then list a large range of memory (by typing, say, "F800.FFFF") you will probably notice that almost every byte value from $00 to $FF can be seen scrolling by (see Appendix A if you feel the need of a brush-up on hexadecimal numbers at this point). Everything that is stored in RAM - programs, ASCII code, textfiles, etc. - is represented by a block of one or more hex values in this range. This means that your machine has 256 different byte values to make use of for representing information in memory (since there are 256 different hex numbers in the range $00 to $FF).

Unfortunately, when communicating with its disk drives, your Apple can't handle such a large range of values. Because of the hardware constraints mentioned above, DOS 3.3 only sends to the

disk, or receives from it, values from $96 to $FF (150 to 256 in decimal). Even within this limited range some bytes are "illegal" in that they violate the Apple's hardware rules. Others are "reserved", i.e., set aside for special disk use. In fact, DOS 3.3 has to represent all the 256 different values that appear in RAM using only 64 values on the disk. Even more amazing is the fact that earlier versions of DOS had to make do with even a smaller range of disk bytes.

A little later on, I'm going to explain these mysteries in considerable depth, but for the moment let's come back to the subject of the two different ways of reading a disk. These are (1) a raw nibble dump, and (2) an RWTS read.

The Raw Nibble Dump

The expression "raw nibbles" refers to the information exactly as it is represented on the disk - in other words, in the specially encoded form described above. When you examine this disk data, you will of course notice (if you are reading a DOS 3.3 disk) that the hex numbers that appear on your screen range in value from $96 to $FF. So, unless you have built-in boolean logic in your brain, this will hardly be recognizable as program or textfile code.

More interestingly, however, you will also see a lot of data which is NOT part of any program. In fact, you will behold dozens of hex numbers whose sole raison d'etre is to help DOS do its job in getting information on and off the disk. These "DOS marks" will tell you a great deal about the disk and will be covered in detail later on. For now, suffice it to say that when you do a raw nibble dump, you can summon up a display of anything on the disk except the label - exactly as it is written on the track you select for reading.

The RWTS Read

"RWTS" is an abbreviation for a subroutine in DOS called "Read and Write Tracks and Sectors". RWTS is the part of DOS that does all the donkey work involved in putting data on the disk and getting it back into the machine. One of its more important jobs is to translate the raw nibbles on the disk back into code which is intelligible both to the Apple ROMS and to programmers.

Another function RWTS carries out is filtering out the DOS marks referred to above. Once a program is loaded into memory, they serve no further purpose and hence are discarded just after being picked up by the read/write head. So what you get when you do an RWTS read is a block of data, neatly translated back into numbers that represent program or textfile code, and without any superfluous disk information thrown in. Another way of thinking about this is to remember that when DOS loads or runs any program, it automatically performs an RWTS read to get the program into memory. A raw nibble dump, on the other hand,

comes directly from the disk itself, bypassing RWTS and DOS entirely.

The software on the CIA disk will enable you to read a disk in either of the above ways. Tricky Dick by itself can only do an RWTS read. But if you also get The Linguist on the case, you will also be able to carry out a raw nibble dump on any disk whatsoever. Much, much more about these functions will be revealed in due course, but first let's get better acquainted with Tricky Dick.

TRICKY DICK'S INSTRUCTIONS

This section briefly outlines the commands used in operating Tricky Dick. If you already possess considerable disk knowledge and experience, these should be enough to get you going. If you have somewhat less expertise, the way to use this section is to work through it fairly carefully, trying out the various commands, and watching their effect on Tricky Dick's display. After this, you should work through the Tricky Dick tutorial which follows this section. This tutorial is designed not only to help you use Tricky Dick as effectively as possible, but also to put you well on the road to becoming a disk expert.

The Tricky Dick Display: Press any key to get out of the initial display that appears when Tricky Dick is run. This will bring up the data viewing area, but will leave intact the three lines of parameters at the top of the screen and the command line at the bottom. These will always remain in view no matter what Tricky Dick is doing. "ALL COMMANDS" at the lower right of the screen prompts you to enter the instruction you wish Tricky Dick to execute. Each command is echoed on the right of the ':' to serve as a reminder of your last input.

Help Screen (/ or ?): Hitting the '/' key brings up a list of each command along with its accompanying key stroke. '^' before a letter indicates that the CTRL key must be pressed at the same time as the letter. The help screen can be accessed at any time. Hit any key to get back to the data display.

Select DOS Version (^D): CONTROL D toggles between DOS versions 3.2 and 3.3, telling Tricky Dick which one is on the disk it's going to be working with. The current version is shown under the word "DOS" at the top of the display.

Slot, Drive, and Device Select (^O): CONTROL O brings the cursor up next to 'SL='. If your disk controller card is in slot 6, just hit RETURN; if not type in the correct slot number. Tricky Dick will check that an Apple disk controller card actually exists in the slot you specify. If a card with P5 and P6 PROMs is present, DOS 3.2 will automatically be brought into play. The cursor will move to 'DR=' prompting you for the number of the drive which contains the disk you wish to work on. After typing in this information, hit RETURN to go to 'PR='. The default value (PR=0) means that printer output is disabled. You

only need to enter your printer card's slot number when you are ready to print out some data from Tricky Dick.

Track and Sector Select (;, <, .>, arrows): Hitting ';' brings the cursor to the top of screen next to 'T='. Select the track number you need and hit RETURN. The cursor will move down next to the 'S=', waiting for you to key in the sector you wish to read. Enter single digit track or sector numbers by typing the digit followed by RETURN. A RETURN alone in response to "T=" and "S=" accepts the value shown. Alternatively, you can decrement or increment the track number shown by hitting the '<' or '>' key respectively (either with or without pressing SHIFT at the same time). Track numbers wrap around when you get to $22.

In a similar manner, you can clock the sector number forward or back by hitting the right or left arrows respectively. If you have selected DOS 3.2, sectors numbers wrap aound when you reach $0C; with DOS 3.3, wraparound occurs at sector $0F. However, the ';' command allows you to designate track numbers greater than $22 or sector numbers greater than $0F/$0C.

Reading a Sector (^R): Press CTRL R to read the sector you have selected into Tricky Dick's buffer (an area in memory specially set side for data storage). The disk you are reading should be in the drive indicated by 'DR='. The sector data will be displayed in the data viewing area on your screen immediately after a successful read.

Volume Number: The volume of the disk you have just read will appear under the letters "VOL" in the upper right hand corner of the screen.

Cursor Movement (I,J,K,M and ^I,^J,^K,^M): I, J, K, and M move the cursor around the data display in exactly the same way they do when used for Applesoft editing. When these commands take the cursor beyond the edge of the screen, scrolling or wrapping around occurs. For example, pressing 'K' when the cursor is at the end of a row causes it to jump to the beginning of the next row; pressing 'M' when the cursor is in the bottom line of the display causes the screen to scroll up one line. Scrolling can be continued until the first or last line of the sector data appears on the screen. Holding down CTRL and hitting any of these four keys increases the distance covered, causing the cursor to jump to the edge of the display.

Editing Single Bytes in the Display: This is accomplished by placing the cursor over the byte you wish to change, and typing in the new hex value. Single-digit hex numbers must be entered with a leading zero (e.g., enter '5' as '05'). If you type the first digit of an entry and then change your mind, simply hit 'J' or 'K'. This moves the cursor away and cancels the entry. Similarly, if you type the first digit incorrectly and want to change it, hit the space bar and start again.

Data Entry Modes (^@, ", ^): Just after loading Tricky Dick, the words "NORMAL HEX" will appear at the lower left of your screen next to the word "DATA". They mean that you can enter hex digits from the key board and these digits will appear in the data display under the cursor. This allows you to edit the screen display before writing it to the disk.

Keying in the " character (SHIFT 2) places you in "high ASCII" mode and causes an inverse "HIGH ASCII" message to appear in the lower left of yor screen. In this mode, each time you press any key, its corresponding ASCII code number with the high bit set will appear under the cursor, and the cursor will advance to the next space. This allows you to type alphanumeric characters into the data display without looking up their corresponding high ASCII codes. Before you can issue any other command, you'll need to hit the CTRL, SHIFT, and 'P' keys simultaneously (i.e., CTRL @). This puts you back into the "NORMAL HEX" mode and allows Tricky Dick to accept your keyboard commands again.

Inputting a ' character (SHIFT 7) will place you in standard, or "low" ASCII mode (signalled by "LOW ASCII" in the usual place). In this mode, each keypress leaves its normal ASCII code under the cursor. To return to normal Tricky Dick functioning, hold down the CTRL, SHIFT, and 'P' keys together. (CTRL @).

The Data Display: When you boot up Tricky Dick and read a sector, the data viewing area will show the first half (i.e., the first 128 bytes) of the sector data. On the far right of the screen you will also see an eight-column alphanumeric symbol display. This part of the screen gives an ASCII translation of the hex data. Control characters show up in inverse and all flashing symbols get changed to normal to minimize distraction.

To view the second half of the sector, hit either CTRL M or RETURN, bringing the cursor down to the bottom of the screen. You can now scroll through the remaining data by holding down the 'M' and REPT keys together. From there, you can scroll back to the beginning by hitting CTRL I, followed by pressing 'I' and REPT simultaneously. Alternatively, hold down the CTRL key and press 'F', 'M', and 'F', in that order. The column of hex numbers on the far left which are followed by a ':' tell you what portion of the sector data you are viewing. They are offsets in Tricky Dick's data buffer and range from $00 to $FF in 8 byte increments.

Flipping Data Displays (^F): If you are in the viewing mode described in the preceeding two paragraphs, a CTRL F will erase the alphanumeric symbols on the right of the screen and display all the data in the last sector read. The cursor will remain over the same byte during the flip. The leftmost column of numbers indicates buffer offsets, but with the trailing zero ommitted to retain clear screen formatting.

From this full sector display, you can of course get back to the partial screen with ASCII by simply pressing CTRL F again. When you do, Tricky Dick displays the next 128 bytes of data, starting with the row in the full display where you left your cursor. For example, if your cursor was in the row numbered $5, then after CTRL F you will see the 16 rows of data and symbols numbered from $50 to $C8. As before, the cursor will remain over the same byte during the flip.

Sector Filling (CTRL Z, CTRL X): In either of the above two displays, CTRL Z replaces the sector data with zeros beginning with the byte over which the cursor is placed, and extending right to the end of the sector. CTRL X fills the sector from the cursor to the end with the byte value under the cursor. Note that these commands do not write anything on the disk itself.

Disassembling Sector Data (L): Hitting 'L' disassembles the sector code beginning with the byte under the cursor and continuing until the screen is filled. The middle column of the display gives the ASCII translation of the hex data to its left. Repeated pressing of 'L' carries on the disassembly until the end of the sector is reached. After disassembling a screenfull of hex you can return to the previous hex display by hitting the space bar or some other noncommand key. The cursor will be positioned next to the last byte that was disassembled.

Listing Applesoft and Integer Code (CTRL L): If the sector data contains Applesoft or Integer BASIC code, a listing can also be displayed. To accomplish this unique feat, Tricky Dick requires only that you select the language you wish to list by pressing CTRL L, then typing an 'A' for Applesoft, an 'I' for integer, or a '*' for assembler. Finally, hit 'L' for a listing which begins with the byte under the cursor. Keep typing 'L's' until you have listed all the code in the sector buffer.

Writing to the Disk (CTRL W and Y): CTRL W, followed by the 'Y' key writes the contents of the sector buffer to the sector whose address shown at the top of the screen. WARNING: Be sure that all the information shown on the screen is correct before executing a write - once you do it, the die is cast! If you foul up here, you will clobber the disk you are working on, perhaps irreparably. This means checking to be certain you have selected the correct drive, track, and sector numbers - and that the data displayed on the screen is the data you want written on the disk.

To help prevent accidents, a unique safety factor has been built in here. When you hit CTRL W, you will hear a series of 6 short tones over a period of about 3 seconds. Keying in a 'Y' during this sequence writes to the disk and stops the tones. However, if you have pressed CTRL W by mistake (not hard to do, since most commands involve the CTRL key), you can simply elect to do nothing, and when the tones cease, the write instruction is automatically cancelled. If you don't want to hear 3 seconds of sound effects, hit any key except 'Y' and they will stop immediately.

Error Messages: When an error occurs during the operation of Tricky Dick, a tone is sounded and a flashing error message occurs inside the '<-->' mark in the upper right corner of the screen. Just above, Tricky Dick displays the accompanying DOS error code inside the '<00>'. A subsequent normal read or write operation clears both the error message and its code from the screen. The chart below shows the type of error, its flashing designator, and its DOS code.

Type of Error	'<-->' designation	DOS code
Write Protect Error	<WP>	10
Drive Error (Read or Write)	<IO>	40

Dealing with Nonstandard Sector Marks (CTRL S): Here, we come to the features of Tricky Dick which allow you chuckle, chuckle, to read from and write to disks whose formatting has been altered - either by accident or design. In order to do this with the least amount of work, you'll need to use The Linguist to determine the exact nature and extent of any such alterations. The instructions for The Linguist are in a later chapter of this book.

Hit CTRL S and the cursor will jump up to first byte of the sector marks. You can then move the cursor along this data with the left and right arrows. To replace any digit, simply position the cursor over it and type in the new digit. The change will appear and the cursor will move to the next symbol. A RETURN gets the cursor back to the data display.

In the top line, "D5AA96" is the standard DOS 3.3 address field header and "DEAA" is the address field trailer. If you have changed the DOS version to 3.2 the top line will read "D5AAB5" which is the 3.2 address header. the next line, "D5AAAD" refers to the data field header and "DEAA" is that field's trailer. **Important:** by replacing one or more of the header or trailer bytes with a '00', you can tell Tricky Dick to accept any value in that position. For example, 'DE00' in the first line causes the second byte of all address field trailers to be ignored during reading or writing.

The third line shows the data field header and trailer (the latter with an added 'EB') once again. The header and trailer you select here will appear in the data field of the next sector you write to the disk. This line is used only for writing and allows you to alter the values in the data field header and trailer of any sector you write to the disk.

Finally, by changing the 'Y's' in the DOS mark section to 'N's', you can tell Tricky Dick to ignore the address field checksum (the first 'Y') and/or the data field checksum (the second 'Y'). The '0' just below the two 'Y's' indicates that Tricky Dick will always write a sector of data to the disk with a data field checksum of $00.

Printing Hard Copy from Tricky Dick (CTRL P): Typing CTRL P allows you to select the form in which you wish the sector data in Tricky Dick's display to be printed. The cursor will jump to the print select parameter, prompting you to type in one of the following instructions: 'H' for a hex dump with ASCII translations; 'A' for an Applesoft listing of the sector code, 'I' for an Integer listing and '*' for a disassembly. The next step is to type CTRL O, followed by two RETURNs. This brings the cursor in position next to the "PR=" for you to key in the slot number of your printer interface. Finally, after making sure your printer is turned on, type 'P' for the action to begin. If you select the wrong slot, you may find that Tricky Dick hangs, or that other strange things happen.

If you are in the '*', 'A', or 'I' modes you will get a listing from the last cursor position to the end of the sector data. In the 'H' mode you get a full sector dump irrespective of the cursor position.

Module Check-out (SHIFT M): Typing a SHIFT M displays a list of the Tricky Dick coresident modules, their names, and a brief description of their functions. If one of the modules is already in memory, its name will be displayed at the top of the screen just to the right of the inverse "Tricky Dick". If no module is in memory, Tricky Dick's version number will be displayed.

Exiting Tricky Dick (RESET, CC): Tricky Dick does not tamper with RESET, so you can use this old standby to jump out any time (except while writing to a disk, of course). CTRL C also exits to BASIC and reminds you that you can restart Tricky Dick by typing a '&' (or use CALL 2051). CTRL Y (or 803G) from the monitor also gets things going again.

Jumping to a Module (CE): To load a module while running Tricky Dick, type "CTRL C" followed by BLOAD (name of module). When the disk drive stops spinning, key in a '&' and you are ready to roll. When you get back into Tricky Dick you will notice that the name of the module presently in memory replaces Tricky's version number in the banner at the top of the screen. To go into a module you have loaded into memory, type CTRL E and the module's introductory display will appear on the screen. Then hit any key and you're ready for action.

If you hit CTRL E with no module in memory, Tricky Dick will let you know by sounding a rather pleasant tone. When you jump to one of the modules, the words "TRICKY DICK" at the top of the screen get changed from inverse to normal. At the same instant, the module's name switches from normal to inverse. This lets you know at glance which program you're "in" - it's always the one whose name is in inverse.

IMPORTANT NOTE: The DOS modifications which are explained in the following tutorial and elsewhere in this section of the book ONLY apply to a standard DOS 3.3. slave disk. If you have a disk with any other kind of DOS (including 3.2 or one of the speedy

DOS versions) do not attempt the patches that follow without making sure the code being written over is the same as in DOS 3.3. The locations for many of the patches are bound to be different, and you could end up ruining a disk.

THE TRICKY DICK TUTORIAL

This tutorial represents the first leg of your journey towards disk expertise. Along the way, you'll pick up a lot of new information, not only about DOS and disk formatting, but also about the Apple II in general and how to get even more enjoyment out of it. I sincerely hope this trip is as much fun for you as it was for me when I first began exploring the inner world of DOS.

Boot up the CIA disk, select Tricky Dick from the menu, and when the initial display comes up, press the '/' key. This brings up the help screen, a source you can always appeal to if you need to recall any of the commands described in the last section.

Before doing anything else, hit the space bar to bring up the data display. This should show a screenfull of $00's indicating that Tricky Dick is empty. When you read a sector off the disk, it is deposited in a specially reserved area of memory (called the "buffer"), and immediately copied to the screen.

An important feature is the column of numbers on the far left. This starts with a '00' and ends with a '78' at the bottom of the screen. Each of these numbers is an "offset" into the buffer, meaning that it is used to count the bytes found there. The way this works can best be illustrated by the second row which starts with '08'.

byte numbers:		$08	$09	$0A	$0B	$0C	$0D	$0E	$0F
		/	/	/	/	/	/	/	/
2nd row -	08:	00	00	00	00	00	00	00	00

The '08' in the second row tells us that the first byte in that row is the 8th byte in the buffer; the next byte is the 9th, and so on.

Since you can change bytes simply by placing the cursor over them and typing in the new value, you will rarely have to think about offsets. However, in this tutorial, I will be using the numbers that appear in the leftmost column as line numbers. So, for example, I would refer to the row above not as "the second row", but rather as "row $08". This notation will ultimately save you a lot of counting.

As you can see, the numbers referred to above are in hexadecimal. This a policy I will be following throughout the book, so if you are a bit rusty on the hex number system, now is the time to turn to Appendix A for a bit of a brush-up. If you

are completely hexed by hex, take heart and dive in to the appendix anyway. In both the US and England, 10 year old school kids are learning alternative number systems as part of their normal class work. If they can do it, so can you!

Now let's do some disk espionage. Start by making a FID copy of your entire original Apple System Master diskette and leave it un-writeprotected. Take the CIA disk out of your drive and put the copy in its place. If you have two drives, leave the CIA disk in drive 1 and put the practice copy in the other drive. In the latter case, select the drive the copy is in by keying in CTRL O, RETURN, the drive number, and RETURN in that order. If everything is working O.K., the cursor should have hopped up to 'SL=', then down to 'DR=' after the first RETURN. The drive number should have been echoed after 'DR=', and the last RETURN should have caused the cursor to jump back into the display.

If you have just booted up, the sector address at the top of the screen should read "T=0" followed by "S=0". So select track 11, sector 0 by typing in the following sequence.

; 11 0 RETURN

During this keying-in sequence, the cursor will perform another series of gazelle-like leaps, finally returning to the data display. The track and sector parameters at the top of the screen should now read "T=11 S=0" and the screen should be filled with '0's'. The last 8 columns on the right of the screen should be filled with inverse "@" signs. This section of the display gives the ASCII translations of the sector data.

The Volume Table of Contents

Now press CTRL R and watch the action. After about one second of disk whirring, some scattered bits of data will appear among the screenfull of zeros. You are now looking at the notorious Volume Table of Contents (alias the VTOC), a part of the disk which, sooner or later in every Apple enthusiast's life, gets overwritten with random data.

We're not going to worry too much at the moment about fixing a clobbered VTOC, but rest assured that before finishing this part of the book, you will be able to perform masterful surgery on this vital diskette organ, even in its most mutilated state. For now, let's take a closer look at the very top line of data which should read:

04 11 0F 03 00 00 FE 00

The first number, '04', has no function and can always be ignored. The next two numbers give the location where the first catalog entries have been written. They should read "110F", telling us that the catalog starts at track $11, sector $0F (it starts at sector $0C on a 3.2 disk). The '3' tells you that the disk's DOS version is 3.3 (a '2' = 3.2). The 'FE' two bytes along

is the volume number of this disk (decimal 254).

The next 3 rows are pretty uninteresting, since they contain nothing but zeros. The fifth row (numbered with a '20' in the leftmost column) contains a '7A' as its last byte. This indicates that each file's track and sector list (we'll talk about these a bit later on) is allowed to hold up to $7A (decimal 122) track and sector addresses.

After another row of zeros, we come to the following line:

12 01 00 00 23 10 00 01

The first number, '12', indicates that track $12 was the last track allocated by DOS for file storage. The '01' following it tells us that the next track DOS will attempt to write to is $12 + $01 or $13. Now these two numbers may be different on your practice copy of the System Master disk, since any small variation in the way FID has written out the files could alter its selection of the next track to be used. For example, instead of a '1', you may see an 'FF'. This tells you that DOS will search in a negative direction, i.e., 12 - 1, for the next available track. In many implementations, a hex number whose high bit is set, in other words, whose value is $80 or greater (see Appendix A), is taken to be a negative number by the system (in which case it qualifies to bear the impressive title, "signed 8-bit binary number").

Skipping over the two unused '00's' brings us to '23 10'. This tells us that DOS has formatted $23 (decimal 35) tracks on this disk and that each track contains $10 (decimal 16) sectors. The final '00 01' on this line indicates that there are $100 (decimal 256) bytes per sector on the present disk (remember that the 6502 usually handles two byte numbers in reverse - i.e., with the high byte last and the low byte first).

The Bit Maps

We now arrive at the "bit map" field, the most important area of the VTOC. This begins on the next line, which should be numbered $38 on your display, and extends up to line $C8. The bit maps are a block of data which tells DOS which tracks and sectors have not yet been written on, and hence are available for storing files. In order to avoid clobbering your precious programs, DOS is obliged to take a look at the bit map field each time you ask it to save any new information on the disk.

The bit maps on the work disk you are now examining will consist mainly of '0's' interspersed with a few higher values. This is not the most helpful configuration for you to learn on, so let's try a little experiment which should prove more educational. Here's what to do.

(1) Take your work disk out of the drive and put an unitialized diskette in its place.

(2) Hit "CTRL C".

(3) Type "CALL-151" to go into the monitor.

(4) Type "BEFE:24".

(5) Now "INIT HELLO" the unitialized disk.

(6) Key in CTRL Y, jumping back into Tricky Dick.

(7) Make sure the display reads "T=11", "S=00", and check that D = the drive which contains the initialized disk.

(8) Hit 'CTRL R', reading in the disk's VTOC.

Right away you will notice, if all has gone according to plan, that the bottom of the screen contains a couple of dozen 'FF's' that were not present in the practice disk's VTOC data. This state of affairs makes explaining things much easier.

Now hit RETURN, followed by a series of 'M's, until line $38 reaches the very top of the screen. This scrolls up some more of the VTOC, exposing another batch of 'FF's'. Since the bit maps start with the first byte in row $38 (a '00'), this brings the largest possible number of them into view. It also brings me to the meaning of all the '00's, 'FF's', and other hex numbers scattered around the place.

Whoever wrote DOS employed a simple and elegant way of recording the status, full or empty, of every single sector on the disk - and squeezing this vital information into as little space as possible. He (or she) did this by assigning a two byte "map" to each track and structuring DOS to vary the maps' values in a way which signals the track's available sectors. The illustration below shows how the maps are linked to their respective tracks.

```
        trk $00          trk $01
38:     |00 00| 00 00   |00 00| 00 00

        trk $02          trk $03
40:     |00 00| 00 00   |FF FF| 00 00

        trk $04          trk $05
48:     |FF FF| 00 00   |FF FF| 00 00
  .
  .
  .
        trk $1E          trk $1F
B0:     |FF FF| 00 00   |00 00| 00 00
```


As you can see, the first '00 00' in row $38 is the bit map for track $00. DOS skips the following two '00's' and assigns the third byte-pair to track $01. This process continues throughout the bit map field.

But how do we translate this succession of seemingly meaningless data into information about each sector's status? The answer lies in the fact that all hex bytes are composed of 8 individual bits (see appendix A, if you are unclear on this). Now if we allot each track two bytes for its bit map, we immediately get 16 bits to play with. This works out quite nicely, since each track contains 16 sectors. And what is even more convenient, each bit can take on one of two values - a '0' or a '1'. So if we assign a single bit to each sector in the track, we can show that a sector is free by setting its bit to a '1'. A '0' value for any sector's bit would signal us (and DOS) that it contains data. The way DOS links bits to sectors is shown in the example below.

sector numbers:	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
bit values:	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
byte values:				3	F							F	F			

Assigning '0's' to sectors $0F and $0E shows that these are in use; the '1's' paired with sectors $0D - $00 tells us that they are free. Since your Apple automatically tranlates the binary number '0011111' into the hex number '3F', and '11111111' into 'FF', the bit map shows up as '3F FF' in the display.

In fact, if you look at the first two bytes in line $80 on your screen, you will probably see this very same bit map. Counting up the bit maps, starting with the one assigned to track $00 in line $38, would tell us that '3F FF' represents the status of the sectors in track $12 (decimal 18). Track $12 is where the HELLO program was placed when you initialized the practice disk and it occupies sectors $0E and $0F. Note that when DOS writes data on a track, it always starts with the highest numbered available sector and works down.

The table at the end of Appendix A allows us to translate any bit map into a hex value since it gives hex (and decimal) equivalents for binary numbers. A glance at the table quickly informs us that if we want to show that all 16 sectors on a given track are free, we need to set all its 16 bits to '1', giving us an 'FF FF' byte-pair. Similarly, to reserve an entire track, we need to shove a '00 00' into its bit map.

Freeing up Track $23

All of which brings me to the reason for asking you to type in that 'BEFE:24' a couple of pages back. That was a patch to DOS which told it to initialize 36 (hex $24) tracks instead of its usual 35. This means you can now make use of track $23

(decimal 35), gaining an extra 4K of program storage (remember that track $23 is the $24th track on the disk because DOS starts with track $00). Here's how to do it.

Hold down the 'M' and REPT keys until the rest of the VTOC has scrolled into view and line $F8 appears at the bottom of your screen. The bit map for track $23 can be found in line $C0 as shown below.

```
                        trk $23
C0:   FF FF 00 00 ¦00 00¦ 00 00
                    --
```

Since DOS isn't used to formatting track $23, it writes '0's' in the track's bit map to make sure it doesn't get used. You are now going to make short work of this trivial obstacle to greater disk storage.

Using the 'I', 'J', 'K', and 'M' keys, place the cursor on the byte indicated by the two '00's' in the above illustration and type in two 'FF's'. Your display should now look like this.

```
C0: FF FF 00 00 FF FF 00 00
```

Next, type CTRL F to flip screens, CTRL-I to jump to the top of the page, and CTRL F again; this brings you back to the first part of the VTOC. Now place the cursor over the '23' in line $30,

```
30: 12 01 00 00 23 10 00 01
                ^^
```

and change it to a '24' to tell DOS that it now has $24 (decimal 36) tracks to contend with.

Finally, check the display to make sure that track $11, sector $00 is selected, and that 'D=' the drive of the newly initialized disk. Then hit CTRL W, followed by 'Y' before the tones stop. Finish off by reading back the VTOC with a CTRL R to make sure your alteration took. You now have an extra track at your disposal.

Getting Extra Space on Track $02

At this point you may be wondering if it's possible to reclaim even more disk space. If so, read on because I'm going to show you how to wrest another 16 sectors from DOS' grasping clutches, bringing the grand total to 32. Let's begin by looking at one of the 3 tracks, which, according to page 135 in the DOS Manual, is exclusively reserved for DOS. Start by selecting track $02, sector $0F, and reading it in. Follow this with CTRL F to get the entire sector's data on the screen at once. Notice anything funny?

That's right - you are now staring at a sector full of nothing. Proceed by hitting the left arrow once to decrement the

sector number to $0E and then read this sector in. Again you'll see a screenfull of '0's'. If you continue by alternately pressing the left arrow followed by a CTRL R, you will be able to do a rapid sector by sector scan of the entire track. This should reveal that every sector from $0F down to $05 is devoid of data.

In spite of the fact that DOS only uses 5 sectors ($00 - $04) on track $02, a quick look at its bit map tells us that the entire track is reserved. Apple may have done this to leave room for further expansion of DOS, but for our purposes, the remaining sectors are going to come in mighty handy for file storage. So let's grab them right now by reading in the VTOC and altering the track $02 bit map as follows.

Start by reading in sector $00 of track $11. You should still have the full sector display, so press CTRL I, causing the cursor to jump to the top edge of the screen. Follow this with CTRL F which takes you back to the display of the first half of the sector with ASCII. When switching displays in this manner, remember that the shorter ASCII display will start with the line in the full display where the cursor was last placed. The cursor itself will always remain over the same byte through any number of screen flips.

We are going to work on the bit map for track $02 which is located in line $40.

```
                trk $02
40:   ¦00 00¦ 00 00 FF FF 00 00
        --
```

But before we do, we need to figure out which byte values must be substituted in. Since we want to free up sectors $0F - $05, the binary equivalent of the required bit map should end up looking like the following.

sector numbers:	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
bit values:	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
byte values:				F	F							E	0			

The table in Appendix A tells that us that the '11111111' and '11100000' in the bit map translate into $FF and $E0 respectively. After shoving these bytes in the track $03 bit map, row $40 should look like this.

```
40: FF E0 00 00 FF FF 00 00
```

If it does, and the drive, track, and sector values are correct, write it to the disk and read it back for verification. This brings the grand total of liberated sectors to 27.

Snatching Space from the Catalog Track

The final 5 sectors of the 32 I promised you are going to come from the catalog track. This is possible because DOS has provided us with more catalog space on track $11 than we could ever fill with file names. Even with the disk completely full of short programs, the catalog hardly ever extends below sectors $07 or $06 (DOS places file names on the catalog starting at sector $0F and progressing downward). Therefore, it is possible to safely make use of sectors $01 - $05 for program storage. As usual, it is necessary to indicate this in the track $11 bit map which is displayed in line $78.

trk $11
78: FF FF 00 00 00 00 00 00

The new bit map which signals the availability of sectors $01- $05 is: 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0. Changing this into hex gives us '00 3E' which should be plugged into the track $11 bit map to produce:

78: FF FF 00 00 00 3E 00 00

Check everything and then write this to the disk. But don't leave yet. You have one important additional detail to attend to. Each sector on track $11 including the VTOC contains the address of the next sector DOS is supposed to access during any operation involving the catalog. Let's have a look at sector $06 as an example of this. Clock the sector count forward by holding down the right arrow and the REPEAT keys simultaneously. When you reach $06, do a read and take a look at row $00. It should look like this:

$00: C0 11 05 00 00 00 00 00

The '11 05' is a link pointer which tells DOS that the next catalog sector is track $11, sector $05. If you look at the same two bytes on sector $07, you will see '11 06' pointing back to sector $06, and so on throughout the track. In any disk operation, the VTOC is accessed first, and its link pointer reads '11 0F', telling DOS to begin searching downward from sector $0F for the requested file name.

Now the one thing we *don't* want is for DOS to write file names over any program code that may be stored on track $11. Fortunately, all we have to do to prevent this is to write two '00's' over the link pointer in sector $06. So you should change row $00 from the form shown above to the one below.

00: 00 00 00 00 00 00 00 00

Writing this back to sector $06 insures that DOS will never look at any sectors lower than $06 for free space when placing file names in the catalog. If all the sectors from $0F to $06 are completely filled (an extremely unlikely possibility), and

you try to save yet another file, you will simply get a DISK FULL ERROR", leaving sectors $01 - $05 untouched.

Before making use of this space, however, there is one last change that must be made. To protect the catalog, DOS has an internal safeguard against saving files on track $11. This must now be disabled, so read in track $02, sector $01, and get line $90 up on the screen.

90: D3 18 69 11 8D EB B3 8D

Change byte $92 from a '69' to an 'A9' and write the sector back to the disk. You now have an extra 8K of disk storage.

Getting Rid of DOS

In a final frenzy of greed, you could free another 21 sectors by writing 'FF's' in the bit maps for tracks $01 and $02 leaving the VTOC display looking like this:

38: 00 00 00 00 FF FF 00 00
40: FF FF 00 00 FF FF 00 00

Since this infringes on the disk space occupied by DOS, eventually part of DOS will get overwritten by your files, and the disk will no longer boot. You can still access the files, though, if you first boot DOS from another disk.

Note: freeing track $00 in this manner is of no benefit, since DOS will refuse to write on this track even if its bit map contains 'FF's' and all the others are zeroed out. Patching DOS to write on track $00 is not advisable because if a file's track and sector list happens to end up there, the file is likely to be inaccessible.

Repairing a Clobbered VTOC

I think you can now see that if a VTOC on one of your disks accidently gets corrupted, it is a trivial matter to rescue the files. To demonstrate this, let's start by blowing away a VTOC. Take the newly initialised disk out of your drive and put your practice copy of the System Master diskette in its place. Next, read in its VTOC, place the cursor on the first byte in row $00 (probably a $04), and type in '0F'. Place the cursor back over this byte and press CTRL X. The sector will instantly be filled with '0F's'. Write this information back to the VTOC of your practice disk. If you now try to boot it or load a file, you will get a FILE NOT FOUND error (DOS now thinks that the catalog begins with sector $0F on track $0F).

To fix this, read in the VTOC from a good disk and place the cursor on the first byte in line $38. Now press CTRL Z, filling the display with '00's' from the cursor to the end of the sector. This zero's out the entire bit map field and thus makes sure that no files on the disk get overwritten during future use. Finally,

write this information onto track $11, sector $00 of the clobbered disk. It is now perfectly usable again, its only fault being that no more data can be stored on it. If this were not a practice diskette, the solution would simply be to FID (not COPYA because it also transfers the zeroed-out VTOC) all the files across to a newly initialised disk and reinitialise the one that got corrupted. However, don't bother doing this with your practice copy.

Undeleting Programs

The first thing you need to know is that DELETEing a file does not automatically remove it from the disk. The only thing that happens is that a couple of bytes on the catalog track get changed and the bit maps are adjusted. The file's name in the directory and its program code still remain intact. The only exception to this may occur if you have saved other programs to the disk since DELETEing the file in question. If so, DOS may have overwritten part or all of it.

Put your practice copy of the System Master diskette in a drive and read in track $11, sector $0F. Rows $08 through $28 should look pretty much like this:

```
                    first byte of file name space
                   /
08: 00 00 00 13 0F 82 C8 C5 :@@@SOBHE:
10: CC CC CF A0 A0 A0 A0 A0 :LLO      :
 .
 . (2 more rows of 'A0's' here)
 .
28: A0 A0 A0 A0 06 00 14 0F :    F@TO
                   /
          last byte of file name space
```

The '130F' in row $08 indicates that HELLO's track and sector list (more about these later) can be found on track $13, sector $0F (but see the next paragraph); the '82' tells us that HELLO is a locked Applesoft file. The next five symbols are the screen codes (i.e., ASCII numbers with the high bit set - remember?) for the letters H-E-L-L-O, and the 25 trailing 'A0's' are screen codes for blanks. Their purpose is to fill in the unused part of the 30 spaces allotted by DOS to each file name. The fourth 'A0' in line $28 is the last byte of file name space. '06' indicates that HELLO's length is 6 sectors.

IMPORTANT NOTE: If your System Master is different from mine, or if your version of FID allocated space in some other way than mine did when you made your practice copy, the various track and sector numbers that appear in this book's illustrations may not agree with yours. This is no problem, however, since their position on the catalog track or elsewhere will always be the same. So later on when we start tracking down files from addresses recorded in various places on the disk, all you need to do is substitute the information on your disk in place of the tutorial's instructions.

Now I want you to do the unthinkable. Get out of Tricky Dick with a CTRL C, then UNLOCK and DELETE the HELLO file on your practice diskette. Jump back with a CALL 2051 and hit the space bar. The display of the HELLO catalog entry, just as it was before you DELETEd it, will reappear on the screen. Now stare hard at the two numbers on the screen which were designated in the illustration above as the first and last bytes of the file name space. Having fixed these firmly in view, hit CTRL R and watch the change.

You should have seen the first byte ($13) jump down to replace the last byte ($A0), and a 'FF' appear where the first byte was. This is all that actually happened during the DELETE. So here's how your screen should now look.

```
                    first byte of file name space
                   /
08: 00 00 00 FF 0F 02 C8 C5 :@@@SOBHE:
10: CC CC CF A0 A0 A0 A0 A0 :LLO      :
 .
 .
28  A0 A0 A0 13 02 00 14 0F :    SB@TO:
                   /
          last byte of file name space
```

All we have to undelete HELLO is to put things back the way they were. So just type an 'A0' over the '13' in the last byte position and put a '13' in the first byte where the 'FF' is. Lines $08 and $28 should end up looking like the following illustration.

```
08: 00 00 00 13 0F 02 C8 C5
 .
 .
28: A0 A0 A0 A0 06 00 14 0F
```

If they do, write the sector back to the disk.

HELLO will now load, run, and appear during a CATALOG just like before. However, there is still one small detail we need to attend to. Read in the VTOC sector and bring line $80 into view. If you zeroed out the VTOC during the first part of this tutorial, you will probably now find a 'FC' in that line instead of a '00'. This is because DOS has altered the track $13 bit map to reflect the fact that sectors $0F and 0E, previously occupied by HELLO, are now available.

One way to fix the present VTOC is simply to write a '00' over the 'FC'. However, changing bit maps is not advisable with most files, since you are unlikely to know which sectors they occupied before they got DELETEd. So the best thing to do is to load the file in question into memory and then save it back again

to the disk. When this is done, DOS will automatically adjust the bit maps for you.

File Type Flags

Remember that '82' in the third byte position? Notice that it got changed to '02' after the DELETE. This is the file type flag; the first value signals a locked, and the second, an unlocked Applesoft program. Here's the complete list.

File Type	Unlocked	Locked
Text (T)	00	80
Integer (I)	01	81
Applesoft (A)	02	82
Binary (B)	04	84
S-type (S)	08	88
Relocatable (R)	10	90

Eliminating Hidden Control Characters

Sometimes control characters are embedded on purpose (a rather dated protection measure) or accidently in program names in the catalog. If this happens, the name will appear when you CATALOG the disk, but you won't be able to access the file. To see how to deal with this minor nuisance, jump out of the Tricky Dick. Then UNLOCK and DELETE ANIMALS on your copy of the system master disk. Now type SAVE followed by these keystrokes:

T CTRL A E CTRL B S CTRL C T

plus RETURN. This saves a file called "TEST" with control characters hidden between each letter. Now CATALOG the disk. You will find "TEST" second from the top of the list. However, if you try to LOAD or RUN TEST you will end up with a FILE NOT FOUND error.

Incidently, a neat trick for quickly spotting file names which contain control characters is to type INVERSE, followed by CATALOG. Do this with your practice disk, and have a close look at the right edge of the white field in which the inverse catalog entries appear. Directly opposite TEST you will see a telltale notch telling you that some hidden characters have used up screen space without actually appearing on the screen.

Now go back into Tricky Dick, read in track $11, sector $0F, and look at line $30. In the ASCII section on the right you will see the file name "TEST" with an inverse 'A', 'B', and 'C' in between its letters. These are the control characters you typed in when you saved the file. You can still RUN or LOAD TEST by typing in these extra symbols when you access the file, but this would be rather troublesome.

Next, place the cursor over the second byte in row $30 (a 'D4'). Next type in SHIFT 2. You will see the inverse message

SHIFT 2

CTL-SHIFT-2

"HIGH ASCII" appear next to the word "DATA" at the bottom of the screen. This inidicates that anything you now type will be in screen ASCII with the high bit set. Key in the letters T-E-S-T and follow this with 3 taps on the space bar. You will see the letters echoed in the ASCII section, and the spaces will appear in the hex dump as 'A0's'. Now press down the CTRL, SHIFT, and 'P' keys all at the same time. The display at the bottom of the screen will change to "NORMAL HEX" and you will be back in normal mode. Finally, write the altered sector to the disk. All those nasty control characters are gone and the file will now answer to its real name.

Finding and Changing a Binary File's Address and Length

This exercise will take us further afield on our jouney along the Apple disk, and we will be leaving track $11 to trace the whereabouts of some actual program code.

Read in track $11, sector $0F. Then hit CTRL F, followed by CTRL M to get to the bottom of the sector, and finish off with another CTRL F to flip back to the ASCII display. Lines $D8 and $E0 should contain the following information.

```
D8: A0 A0 A0 11 00 19 0F 84 :   QPYOD:
E0: C2 CF CF D4 B1 B3 A0 A0 :BOOT13  :
```

The '19 0F' in line $D8 tells us where to find the first sector of BOOT13. The '84' in the same line tells us that we're dealing with a binary file. So let's read in track $19, sector $0F and have a look.

We are still in the lower half of the sector display (rows $80 - $F8), but need to get back to the beginning. Holding down the CTRL key while pressing 'F', 'I', and another 'F' should do the trick. The 3 rows of numbers starting on line $08 constitute BOOT13's track and sector list. As the name implies, this lists all the tracks and sectors occupied by the file. If your copy is the same as mine, this should show BOOT13 starting on track $19, sector $0E, and running down to track $19, sector $06.

```
08: 00 00 00 00 19 0E 19 0D
 .
 .
18: 19 08 19 07 19 06 00 00
```

Now hit the left arrow key to get to sector $0E, and do a read. Your sceen will display the first 128 bytes of the program code. The first 4 bytes of this data give the address and length of BOOT13.

```
                                first byte of program code
      address   length          /
00:  |00  17| |F0  08|  20  E3  03  84
```

In the usual 6502 reverse notation, this tells us that BOOT13 starts at location $1700 in memory and is $8F0 (decimal 2288) bytes long. You can relocate BOOT13 so that it BRUNS or BLOADS at another address in memory simply by changing the first two bytes. For example, change byte $01 from '17' to '16 and write the sector back to the disk. Then get out of Tricky Dick, BLOAD BOOT13, and type CALL -151 followed by '1600L'. You will see the first 3 bytes of the program, '20 E3 03' start at $1600 instead of the original $1700. Do not try to run BOOT13 at this address, however, because it is not a relocatable program.

You will now have to BRUN Tricky Dick once more, since BOOT13 has blown part of it out of memory. Do this, and read in track $19, sector $0E again. If you plan on putting your practice copy of the System Master to use later on, you had better change BOOT13's start address back to $1700.

Place the cursor on byte $04 (a '20') in row $00 and hit the 'L' key. This will give you a disassembly of the first sector's code. The first instruction should read "JSR $03E3" (meaning Jump to a SubRoutine which starts at the address $03E3). This is a kind of indirectly routed GOSUB to a DOS subroutine which tells DOS the whereabouts of some vital disk reading parameters.

Keep pressing 'L' until byte $A9 scrolls into view near the top of the screen. You should now be able to see the word "SECTOR" in the ASCII display column. The addition of this ASCII information to the disassembly enables you to quickly locate any text in a block of machine code. Keep hitting 'L's' until you have dissambled your way through the entire sector. When you reach the end, the lower part of your screen will be blank and nothing further happens in response to an 'L'.

Listing Applesoft or Integer Programs Directly from the Disk

This works pretty much the same way as the above process. Read in track $11, sector $0F and find the address of HELLO's track and sector list. It should be located on track $13, sector $0F, so increment the track number by pressing the '>' key twice and do a read. The track and sector information now on your screen reveals that the HELLO program code starts on sector $0E and occupies 5 sectors on the same track.

Now read in sector $0E and take a look at the first line. The illustration below shows how to interpret the data that it contains.

```
                                    start of program code
      length          line no.  /
00: |71 04| |19 08| |0A 00| B2 20
                \ /
                memory location of next line
```

The first 2 bytes tell us that HELLO is $471 (decimal 1137) bytes long (no starting address is given at the beginning of an

Applesoft program because it automatically loads at $801). Bytes $02 - $03 are a pointer indicating that when HELLO is in RAM, the next line will start at $819. Each line of an APPLESOFT program starts with this type of pointer, though, of course, you don't see it when you LIST the code. Moving on, the '0A 00' is the hex equivalent of the first line number (line 10). Finally, the program code itself starts with 'B2', the Applesoft token which stands for 'REM'.

Now press CTRL L, followed by 'A'. The cursor will jump to the top of the screen and change the '<*L>' to a '<AL>'. This tells Tricky Dick that you want a listing of some Applesoft code. Put the cursor over the 'B2' and press 'L' to list the first sectorful of BASIC. The continuation of this code can be found by moving on to sector $0D and typing 'L' again. You can look at the whole of any Applesoft program by noting in its track and sector list the sectors it occupies, then working through them as explained here.

By the way, when you are listing the first sector of an Applesoft program, placing the cursor anywhere to the left of the first token may result in a spurious first line number. This is because Tricky Dick interprets the length and pointer bytes as BASIC tokens. The rule to follow when LISTing any sector is to place the cursor on the first byte to the right of the first '00' in the sector.

Integer listings work almost exactly the same as those of Applesoft. If you want to try one, get the address of the track and sector list for ANIMALS from track $11 (the '81' in front of the file name tells you that it's a locked integer program). The first line in the first sector should look like the following illustration.

```
                                    first byte of program
      length     line no.  /
00: |6D 10| 08 |00 00| 5F B1 E8
                \
                length of first line
```

The preliminary information tells us that ANIMALS is $106D (decimal 4205) bytes long, the first line is 8 bytes long, and its line number is 0.

Type in CTRL L, then 'I', to switch on the INTEGER BASIC lister. Put the cursor over the first program byte (the '5F') and hit 'L' to display a listing of the first sector.

To return to the assembly listing mode, simply type CTRL L followed by a '*'. You can also get any of the above listings on your printer as explained under "Printing Hard Copy from Tricky Dick" in "Tricky Dick's Instructions" in the first part of this section.

If you have followed along with me this far, you will have put to use most of the Tricky Dick commands in realistic disk editing tasks. So by now, you should be able to use Tricky Dick pretty easily and find little difficulty in doing more advanced work. But what is far more important, you already know more about the DOS 3.3 disk than about 80% of all Apple users. This is only the beginning, however. By the time we finish our work together, I hope to elevate your standing from the top 20% to the top 1% of the DOS intelligensia. So turn off your machine for a while and take a well-deserved coffee break before we begin ....

CHAPTER THREE — Intermediate Level Tricks with Tricky Dick

These aren't really any more difficult than the things discussed in the foregoing tutorial, but I'm going to describe them in somewhat less detail. So if you already knew something about disk formatting when you bought the CIA, and found the tutorial a bit elementary, this is the place to start. But don't forget that the following techniques apply to DOS 3.3 only unless otherwise stated.

Avoiding DOS Language Card Clobber

DOS 3.3 has particularly pesky subroutine which stores a $00 in the first byte of the language card whenever we do a PR#6. This in turn makes DOS think that the language card is empty. So if you happen to have INTEGER BASIC (or some other program) there, and then boot up from the keyboard, you always have to reboot your System Master and hang around while it reloads INTEGER. Most of the time, however, a perfectly good image of INTEGER is still in the language card in spite of the LANGUAGE NOT AVAILABLE message you get when you try to call it.

Fixing this is a piece of cake. Just read in track $00, sector $09, and write 3 'EA's over the '8D 00 E0' in line $D0, leaving it looking like this.

D0: C0 A9 00 EA EA EA 4C 44

Then write it back to the disk. Any disk with this patch in its DOS will leave INTEGER in peace when booting.

IMPORTANT NOTE: I know it seems obvious, but don't forget to reboot the DOS you've just altered with the following patches if you want to see them in action.

Switching the HELLO File

To make this simple alteration, read in track $01, sector $09. Starting with byte $75 in line $70, you will see the name of the HELLO program. If you want another file on the disk to run automatically on boot-up, put the cursor over the first byte of the HELLO file's name (a 'C8' for 'H' on most disks), and key in SHIFT 2 (the " character). Now type in the new file's name and press down the CTRL, SHIFT, and @ keys together, returning to normal operation. If the name of the new file is shorter than that of the old one, there will be some unwanted characters tacked on at the end. Be sure to type 'A0's' (ASCII for spaces) over these before writing the sector back to the disk.

Using a Binary or EXECable HELLO File

Normally, when DOS finishes booting into RAM, it issues a RUN command to start the HELLO program. However, if you used the foregoing method to switch HELLO to a machine language or EXEC file, you will obviously want DOS to issue the correct BRUN or

EXEC command on boot-up. To do this read in track $00, sector $0D and change byte $42 from a '06' to:

(1) a '34' to BRUN a binary HELLO program;

(2) a '14' to EXEC an EXEC file.

Now write the sector back to the disk. You may wish to make this and the foregoing patch on a COPY of your CIA disk so that it's DOS BRUNs Tricky Dick immediately on boot-up. If you do, you should leave line $40 looking like the example below.

```
40: 03 A9 34 D0 05 AD 62 2A
            \
          binary HELLO flag
```

Loading a Program between DOS and its Buffers

Having carried out the preceeding two operations, you might decide that it would also be useful to place your program in some secure spot in memory where subsequent loading and running of other files cannot overwrite it. The best way of doing this is to move the DOS buffers down and load your program on top of them. A simple DOS patch will insure their complete safety even if DOS is coldstarted.

To set things up, read in track $00, sector $0C.

```
00: D3 1C 81 1E BD 1E 75 2A
```

The next step is to subtract the length of your program in bytes from $1CD3, the number shown in reverse at the beginning of line $00. So if your file was, say, $200 (decimal 512) bytes long, you'd have to work out that $1CD3 - $200 = $1AD3. You should now reverse the high and low bytes of this result in the classical 6502 manner, type them over the 'D3 1C', and write the whole works to the disk.

```
00: D3 1A 81 1E BD 1E 75 2A
```

What happens is, the '1A D3' gets changed to '9A D3' on boot up, moving the buffers down the required amount. This allows you to fix your program to run $9D00 - $200 = $9B00 in RAM.

Eliminating the Pause during a CATALOG

If you manage to accumulate a large number of files on a single disk, you may find it useful to have continuous scrolling during a CATALOG. If so, read in track $01, sector $0D of the disk whose DOS you wish to provide this service. Then simply change byte $34 from a 'CE' to a '60' as shown below.

```
30: 8D 20 ED FD 60 9D 33 D0
            \
          changed byte
```


Write this back to the disk and you will find, after rebooting, that the patched version of DOS will not stop after each screenfull of file names during a CATALOG, but will scroll rapidly through to the end of the list. If you have an autostart monitor, you can use CTRL S to stop/restart the listing.

Changing the "DISK VOLUME" Catalog Message

In order to personalize your disks, you might like to have some message other than "DISK VOLUME 256" appear when a CATALOG is executed. If so, read track $02, sector $02 and change the "DISK VOLUME" message (written backwards!) that begins at byte $B0. If you write over the space ($A0) at byte $AF, you can squeeze in up to 12 characters by hitting SHIFT 2 (the " sign) and typing them in backwards. So if your new heading is to be, say, "Sammy's Disk", lines $A8 - $B8 would look like the illustration below.

```
                                        first byte of entry
                                              |
A8: C9 C1 C2 D3 D2 C1 C2 CB :IABSRABK:
B0: D3 C9 C4 A0 D3 A7 D9 CD :SID S'YM:
B8: CD C1 D3 04 11 0F 04 00 :MASDQODØ:
                                              |
                                        last byte of entry
```

If you carried out the preceeding instructions, and rebooted, you should get the following heading on each CATALOG.

```
SAMMY'S DISK254
```

This looks a bit messy, so to get rid of the '254', read in track $01, sector $0C and type 3 'EA's over bytes $C0 - $C2, ending up with:

```
C0: EA EA EA 20 2F AE 20 2F
```

After writing this back to the disk and rebooting, your catalog message will blaze forth in its most pristine form.

Putting Headings on the Catalog Track

There are few more frustrating experiences than searching through dozens of disks for a program you urgently need, realizing that you have overlooked it, and then having to start the whole tedious business from scratch once again. Some order can be brought to disk chaos by inserting headings on the catalog track and making sure the type of files that they apply to are placed underneath. For example, it might be useful to get the following display upon CATALOGing a disk.

```
 T 000 GAMES
 T 000 -----
*B 062 PIRATE'S SWAG
*B 071 ROBIN HOOD'S LOOT
```

With a newly initialized disk in the drive, type in the following sequence: SAVE XXXXX SAVE YYYYY DELETE XXXXX DELETE YYYYY. Then examine track $11, sector $0F and you will see the 'X' and 'Y' strings just beneath the HELLO entry. The 'X's' will most likely be in line $30. So start by placing the cursor on the 'FF' in the line above (byte $2E) and type in '24 00 00'. This should leave your cursor on the first of the 5 'D8's' (ASCII for 'X') in line 30.

Now you can press SHIFT 2 and type in the letters G A M E S, leaving the string 'C7 C1 CD C5 D3' in place of the 'D8's'. Be sure to finish this sequence by pressing CTRL SHIFT P (CTRL @) to get back into normal mode. Lines $28 and $30 should look like this.

```
28: A0 A0 A0 A0 02 00 24 00 :    B@$@:
30: 00 C7 C1 CD C5 D3 A0 A0 :@GAMES  :
```

The next step is to press 'H' 3 times to bring your cursor over the '13' at the end of the string of 'A0's in line $48. Type in 'A0 00'. This should bring you to the beginning of the deleted entry for the 'Y' program.

Now you have only to repeat the above process. In other words, just type '24 00 00' over the 'FF 0E 02' in line $50. Then follow this with a SHIFT 2 and hit the '-' key 5 times, leaving a trail of 'AD's' over the 'D9's'. After CTRL @, move straight down to the '14 02' and replace this with an 'A0 00'. Finally, check everything and write the sector back to the disk. Reboot and do a CATALOG to make sure the heading got set up O.K.

If you want a flashing instead of a normal heading, press SHIFT 7 (the ' character) before typing in the heading's letters. Inverse characters can be obtained by changing the normal screen ASCII numbers as follows:

numbers beginning with a 'C' ..... change the 'C' to a '0'
numbers beginning with a 'D' ..... change the 'D' to a '1'
'A0' (a space) ..... change the 'A' to a '2'

If you change the ASCII for "GAMES" in this manner, you would end up with:

```
30: 00 07 01 0D 05 13 A0 A0 :@GAMES  :
```

You can now transfer your favorite games to the disk and they will automatically appear beneath the heading. If after doing this you still have some space left over, you can easily use the same procedure to shove another heading underneath the games on the catalog. Further files can be added below this, and so on.

By the way, when you SAVEd the 'X' and 'Y' files, 4 sectors were set aside by DOS to store their nonexistent data. Subsequently DELETEing them readjusted the bit maps to reclaim

this wasted space.

Another point to take note of was the '24' we put in the dummy files' track pointer byte. This was done to prevent the catalog heading from being accidently DELETEd. If you now try to access "GAMES" with any DOS command you will get an I/O ERROR, since track $24 cannot be reached on the Apple drives (the '-'s' are safe in any case because they are illegal catalog characters).

Hiding the Hello File on the Catalog

If you used the foregoing method to create headings and want to get the word "HELLO" out of the way - or you simply want to conceal the existence of your HELLO program during a CATALOG - you can make it do a disappearing act as follows. First, read in track $11, sector $0F and put the cursor over the first 'A0' after the HELLO file's name. Then type in 19 '88's' and check your work by counting up the inverse 'H's' which will have appeared in the ASCII display after "HELLO" (or whatever the file's name is). There should be 19, since $88 is the ASCII value for CTRL H.

Having done this, you now need to let DOS in on your little secret so that it can recognize HELLO on boot-up. So what you now have to do is change the DOS record of the HELLO file's name as described a couple of pages back under "Switching the HELLO File". Follow those instructions to read in track $01, sector $09 and add 19 '88's' after the HELLO program's name. The program will run automatically when you boot the disk, but of course, DOS will ignore any direct commands referring to HELLO, since HELLO now contains 19 extra control characters.

A couple of points are worth noting here. First of all, it seems that 19 '88's' is always the correct formula, regardless of the file name's length (but you can't hide files whose names are longer than 11 letters due to the 30 character maximum permitted by DOS). Secondly, the reason this method works is that CTRL H's output backspaces to the monitor. So what happens is that the HELLO file's name gets printed for a tiny fraction of a second, too quick for anyone to spot it. Then along comes the next filename to completely overwrite it.

Changing DOS Error Messages

If you feel capable of a more eloquent turn of phrase than the author of Apple DOS, you might like to change the wording of some of the DOS error messages. These begin on track $01, sector $08, byte $75, and end on the next sector ($09), byte $3D. So let's assume, for example, that you want to change I/O ERROR message to CRASH OUT (clearly a far more descriptive choice). Start by reading in track $01, sector $08. The message starts in line $C8.

C8: 41 54 43 C8 49 2F 4F 20 :ATCHI/O :

Position the cursor over the '49', press SHFT 7 (the ' sign) to go into normal ASCII mode, and type "CRASH OU". Now press CTRL @, followed by SHFT 2 (the " sign) to switch to high ASCII (the last character is in high ASCII to flag the end of the message). Type in the final 'T' and write the sector back. Boot the disk, leave the drive door open, and type "LOAD HELLO". Your altered message should quickly appear.

This can be done with any of the error messages. Just remember to end up with a high ASCII character, and make sure your own message's length does not exceed the one you are replacing.

Some Ideas for Advanced Programmers

Here are a few Tricky Dick tidbits that you assembly language programmers may find useful. And even if you don't know your way around an assembler too well yet, some of these may prove helpful.

When you have one of the CIA modules in memory, hitting CTRL E causes Tricky Dick to jump to it and begin execution. This feature makes it possible for you to install your own program and access it with the same instruction. Furthermore, you can easily interface your code with Tricky Dick to call Tricky's internal routines.

The first thing Tricky Dick does on CTRL E is attempt to distinguish a CIA module from left-over garbage in RAM. In order to do this, it EORs the byte at $8000 with the one at $8001, then CMP's this with the byte at $8002. If a match is found, it JSRs to $8003 where the modules' code begins. If a match does not occur, a tone is sounded and normal operation is resumed. To use the CTRL E hook, you need to assemble your programs to run at at this address and set up the first 3 bytes accordingly.

Tricky Dick contains both a 3.3 and a 3.2 RWTS. The 3.3 version begins at $3800, and the 3.2 at $3000. They both use the same IOB which starts at $815, and share the device characteristics table which starts at $826. The information in these two lists is in exactly the same order you would normally expect. A JSR to $121F invokes a subroutine which looks at $82A to determine which DOS version has been selected, then calls the appropriate RWTS. Reading or writing with this RWTS will be done using the DOS marks shown in Tricky Dick's display. Tricky Dick stores its sector data into a buffer starting at $2E00.

If you want to BSAVE Tricky Dick with any patches or changes to it, use A$803, L$3800.

You can call each of the 3 versions of RWTS in the machine independently. Thus, you could, for example, use the DOS RWTS at $B800 to read a disk, and one of the Tricky Dick RWTS' to write it out again - or vice versa. With Tricky Dick and a module in memory, there is still free core from $4000 to $7FFF for use as a buffer or anything else. After your program has done its thing, it can return control to Tricky Dick at any time with an RTS.

Moving Closer to the Disk

Well, that just about closes the files on Tricky Dick for now, though I'll be returning to some of this utility's more sophisticated capabilities when I show you how to work on copy protected disks. However, without a little help from the other members of the CIA, there are just some jobs that Tricky can't do. You'll find that for some of the work you'll be wanting to carry out, you're going to need to delve into the most inaccessible parts both of normal and abnormal disks - and make complete sense of all the information they contain. That's why you now need to meet THE LINGUIST.

CHAPTER FOUR — The Linguist

Often during the course of your career in disk espionage, you're going to want to scrutinize a floppy at the most intimate level and obtain a precise translation of what you find there. It might be that a disk failure has seemingly rendered your most precious file irrecoverable. Or, it may just be that you are insatiably curious to discover the latest animation techniques used in your favorite games. In fact, whatever your reason for having acquired an Apple, the need to know more about the software that makes it spring to life is an inescapable part of owning and using it.

The ability to do this quickly and easily will help you out of many sticky situations, and immeasurably increase your knowledge and enjoyment of your machine. Because of the Linguist's unique talents, you'll soon be able to examine, make sense of, and translate into machine recognizable form, the raw data from ANY disk available at the time of this writing - regardless of the extent to which its formatting has been accidently corrupted or deliberately altered.

The Linguist allows you to do a raw nibble dump of an entire track, examine the data at your leisure, and then carry out a sector by sector translation of the raw disk nibbles into code in the form that your machine normally deals with. In other words, the Linguist will transform all 3 of the encoding formats used on Apple disks (6&2, 5&3, and 4&4) into a recognizable and executable form.

Incidently, I've noticed that a lot of Apple users are surprised or at least uncertain when the subject of disk encoding comes up in a conversation. Few computerists are aware that the the familiar form of program code, the one you can view by typing CALL -151 and listing it, gets unrecognizably altered when it is written to the disk. Far fewer still can tell you just how the encoding process works. You already received a fleeting initiation into the ranks of the DOS elite in "A First Word about Disk Data" in chapter two. In the present chapter you will learn a great deal more about this subject and several others. By the time you finish the book, you will be the envy of your local users' group as you nonchalantly extol the speed advantages of 4&4 over 6&2 and 5&3, and confidently hold forth on other equally esoteric disk topics.

The immediate importance of all this is that after dumping a track of raw nibbles and translating them into ordinary hex, you can immediately jump back to Tricky Dick and display this code as an Applesoft, Integer, or assembly listing. Of course, Tricky Dick is also no slouch at translating. But where a disk has been too badly clobbered, too heavily protected, made sectorless, encoded in 4&4 - or simply requires a raw nibble examination to establish the formatting - old Tricky is just not up to the job. That's when the Linguist springs into action at the touch of a

CTRL E to help out.

HOW TO INSTRUCT THE LINGUIST

Just as with Tricky Dick's instructions, if you're an expert, you can get started right away with the Linguist merely by using this section as a reference guide. On the other hand, if you would describe yourself as somewhat below the disk master level, the best thing to do is try out each instruction carefully, watching its effect on the screen. Having done that, you'll be ready for the Linguist tutorial, "The Apple Disk Language School", which immediately follows.

The Linguist Needs Tricky Dick: Tricky Dick and The Linguist have what you might call a permanent intimate relationship. That is, The Linguist will not work properly unless Tricky Dick is also present in memory. This is because the program uses some of T. D.'s internal routines in order to both conserve RAM and to maintain an efficient interface with the user. So if you try to run The Linguist alone, you'll get some rather strange effects.

Once Tricky Dick is in memory, hit CTRL C and BLOAD The Linguist. It loads in at $8000 in memory and can be accessed at any time from T. D.

Getting from One to the Other (CTRL E): When the Linguist is in memory, you can jump to it at any time from Tricky Dick by pressing CTRL E. When you first enter The Linguist, the disk arm recalibrates and emits that grating machine-gun rattle we have all come to know and hate. The sound effects notwithstanding, this does not in any way harm your drive. Once in The Linguist, you can always get back to Tricky Dick by typing CTRL C.

The Help Screen (? or /): If you happen to need a brief reminder about any of The Linguist's commands, just hit the '/' key to display the help screen. This brings up a list of instructions, together with their accompanying keystrokes. A '^' before a letter indicates that the CTRL key must be pressed at the same time as the letter. You can call up the help screen at any time.

The Linguist's Data Display: Hitting the space bar gets you from The Linguist's introductory or help screens to the data display; several of the Linguist's commands also automatically do the same. This screen shows you one memory page (256 bytes) of code in the data buffer, which starts at $4000 and ends at $7FFF in RAM. The 4-digit hex addresses on the far left of the screen correspond exactly to memory locations in the buffer. This allows you to calculate the buffer location of any block of code by simply counting up. For example, the position of the tenth byte in row $4000 would be $400A.

Entering Commands: The "ALL COMMANDS" message at the bottom of the screen prompts you enter the instruction you want the Linguist to carry out. Each command is echoed on the right of the '?'. An incorrect entry is signalled by a brief "ping".

Track Select (< : >): You can decrement or increment the track number (which appears after 'T=') a half track at a time by hitting the '<' and '>' keys respectively. The track numbers progress up to $7F, at which point wraparound takes place. This is to allow the use of The Linguist with drives capable of reading 80-track diskettes. To clock the track number forward or backward rapidly, hold down one of the two keys and the REPT key simultaneously. An often quicker way to input a track number is to type in a ';', followed by the number. This only works for whole tracks, but the next adjacent half track can always be specified by hitting the '<' or '>' keys.

Seeking the Disk Arm to Track $00 (^S): Hitting CTRL S recalibrates the disk arm, pulling it back to track $00. It comes in handy for clocking back the track number rapidly, and also to give your drive a reference point to find the selected track (during a raw nibble read your machine makes no check of the track numbers on the disk to find the current track).

Reading a Track (^R): Hit CTRL R to read the desired track into The Linguist's buffer. Be sure to select the correct drive while still in Tricky Dick before jumping to the Linguist. One buffer page of data will appear in the data display area immediately after a read. If the introductory or help screens were up at the time of this command, a switch to the data display will automatically take place.

Paging through the Buffer (arrows): Pressing the right arrow changes the data display to the next page in the buffer. So if the data from, say, locations $4000 to $40F0 are displayed, hitting the right arrow once flips to the data located from $4100 to $41F0. This can be continued until you reach the end of the buffer at $7FFF. In the same manner, the left arrow pages backward a page at a time until you get to the beginning of the buffer. Extremely fast scrolling can be accomplished by holding down one of the arrows and REPT at the same time. The controls are "locked" so that you never accidently stray lower than the beginning or higher than the end of the buffer

Jump to the Beginning or End of the Buffer (^B, ^N): You can instantly get back to the beginning of the buffer at $4000 by hitting CTRL B; CTRL N takes you right to the end (page $7F).

Cursor Movements (I, J, K, M and ^I, ^J, ^K, ^M): I, J, K, and M move the cursor around the data display just like they do when used for Applesoft editing. When these commands take the cursor beyond the edge of the screen, scrolling or wrapping around occurs. For example, pressing 'K' when the cursor is at the end of a row causes it to jump to the beginning of the next row; pressing 'M' when the cursor is in the bottom line of the display causes the screen to scroll up one line. Holding down CTRL and hitting any of the 4 keys increases the distance covered, causing the cursor to jump to the edge of the screen.

A continuous scroll can be obtained by moving the cursor to the top or bottom of the screen and pressing REPT together with the 'I' and 'M' keys respectively. This can be continued until the first or last line of the buffer data appears on the screen.

Decoding the Address Field Information (cursor controls): At the bottom left of your screen you will find a separate line of data consisting of a 4-digit address followed by 4 2-digit numbers. It displays a translation from 4&4 into normal hex of 8 consecutive disk nibbles starting with the byte under the cursor. This enables you to decode the address field information by placing the cursor on the first byte to the right of 'D5 AA 96' address field header of any sector. The data line will display the results in the following order: the buffer address of the byte under the cursor, the disk's volume number, the current track number, the number of the sector whose address field you are examining, and the address field checksum. The buffer address is always a 4-digit number and the other items are each 2-digits long. All are in hexadecimal and the sector number represents the physical sector on the disk. The meaning and interpretation of this data will be fully explained in the tutorial which follows the instruction section.

Since most disks use 4&4 nibbles in the address field only, the 4&4 translation will usually be meaningless if the cursor is placed elsewhere among the sector data. One important exception, however, occurs when entire files have been deliberately encoded in the 4&4 mode (as can be found in many current games). If you are working with such a disk, you can decode a string of 8 4&4 nibbles by positioning the cursor over the first byte of any 4&4 byte-pair.

Changing the Decoding Mode (^D): To switch between the 6&2, 5&3, and 4&4 decoding modes, hit CTRL D, followed by the number after the '&'. For example, if you are presently in 6&2, you will see a '<62>' just under the '<-->' at the upper right of the screen. To change to, say, 5&3, type in a CTRL D, causing the cursor to leap up and cover the '6'. This is your cue to press the '5' key which automatically changes the displayed mode to '5&3'. If you previously changed Tricky Dick's DOS version to 3.2, the display will already show '5&3'.

Translating a Sector (^T): After having selected the encoding mode, place the cursor over the first byte after the 'D5 AA AD' (on a normal disk) data field header of the sector you want to translate. Then press CTRL T (which evokes a high-pitched squeek), and jump back to Tricky Dick with a CTRL C. The sector's raw disk nibbles will have been translated into normal hex bytes. The translation will be in Tricky Dick's buffer and can be displayed on the screen in the usual manner by pressing

first two lines of which should look something like this:

```
                first data byte in buffer
                /
  4000-9BAFF4EF DFF5F5DF E5F6F4E7 E5F6F3E6
 /4010-E5D6F3FD D9FFBD9D 9FA6BBFD DFD3FAB3
/  /
buffer addresses
```

Please note that although the above 2 lines actually exist somewhere on track $01 (in sector $0F to be exact), you are unlikely to see the same data residing in the first two rows of your display. This is because a raw nibble dump simply turns on the drive motor, moves the arm to requested track, and then starts reading at whatever point on that track the arm happens to end up at. It keeps on reading until it fills the data storage area. Thus, each time you read the same track, although you will capture all the data, the dump will begin at a different place. This contrasts with an RWTS read which always goes directly to a specificied sector and begins reading precisely at the beginning of that sector.

Data Storage and the Buffer

The "buffer" is a block of memory ($4000 to $7FFF in RAM) in which The Linguist places the data from the track it has just read. So the '4000' and '4010' in the last illustration are addresses at the beginning of The Linguist's buffer. The numbering system works just like Tricky Dick in that the number on the extreme left is the memory address of the first byte in its respective row. In other words, the first byte in the row that starts with '4000-' (a 'D5') resides at $4000 in memory. The next byte (an 'AF') can be found at $4001, and so on.

Now let's play around with the display a bit. Hit the right arrow once and note the instantaneous screen flip. The '4100-' now at the beginning of the first line tells you that you have just moved $100 (decimal 256) bytes forward through the buffer. The display will move along one page each time you press this key. A "page", by the way, refers to a block of $100 (decimal 256) bytes of RAM beginning at an address ending in '00'. Hence the sudden progression from '4000-' to '4100-' (i.e. page $40 to page $41).

You can get the same effect in reverse by hitting the left arrow. If you do so now, you will see page $40 pop back into view. If your cursor is on the bottom line of the display each time you hit 'H', another line of 16 bytes will scroll into view. A similar effect in reverse takes place when you press 'I' with the cursor on the top line. Holding down the REPT key together with either 'I' or 'H' moves things along a bit faster.

Now since the buffer extends from $4000 to $7FFF in memory and is thus $4000 (decimal 16384) bytes long, you might like some rapid means of getting from one end to the other. The Linguist

provides a couple of ways of doing this. You can get extremely fast scrolling by holding down the REPT key together with either the left or right arrow. Remember also that CTRL N takes you right to the very end of the buffer and CTRL B shoots you back to the beginning. Go ahead and try these commands out if you like.

This seems like a good time to point out a couple of important details about the way raw disk data is placed in The Linguist's buffer. First of all, the total number of nibbles stored on any one track usually runs around 6,300 (decimal). The exact quantity depends on the speed of the drive in which the disk was initialized (a crucial factor in a couple of copy protection techniques I'll be talking about later on). Since The Linguist's buffer is over 16,000 bytes long, this makes it possible to store more than one image of the requested track.

Thus, the first image starts at $4000 in the buffer and almost always ends somewhere on page $58. The second image usually starts on page $58 or $59 and finishes on or about page $71. This leaves us with enough space between pages $71 and $7F to store close to 9 sectors of a third image.

Why bother with more than one image? Remember that during a raw nibble dump, no analysis whatsoever of the track's data is carried out. This means that, unlike an RWTS read by Tricky Dick, The Linguist does absolutely no checking to make sure the data has been correctly read. So it's not at all impossible to find a glitch or two somewhere in one of the track images. When you use The Linguist to translate a sector of raw nibbles and subsequently get Tricky Dick to list it in recognizable code, any errors in the sector's original data will make the listing more difficult to understand. If you find a few apparently meaningless bytes in the final display, you can always go back to the track dump and translate one of the other images of the corrupted sector.

The Sync Bytes

Press the right arrow a few times and watch for a longish string of 'FF's' in the midst of the other data. They should always serve as a focal point when looking through the raw nibbles, so stop flipping pages when you reach the first batch. These 'FF's' are the notorious sync bytes (or sync nibbles) which enable the machine to lock in correctly on the data it reads in.

Just to put you in the picture on sync bytes, I'm going to digress a bit here to describe them more fully. First of all, if you look in Appendix A, you'll find that 'FF' in hexadecimal translates into '11111111' in binary. However, the sync 'FF's' are somewhat special in that they have 2 extra zeros tacked on the end to give '1111111100', a kind of 10-bit byte.

These extra 2 '0's', believe it or not, enable the machine to correctly divide up into 8-bit bytes the thousands of bits that come pouring in on each read. Fortunately, the precise way

this process unfolds is of no relevance to anyone but a professional engineer, a state of affairs which allows us both to escape a detailed 3 pages or so of explanation. Nevertheless, there are a couple of things about sync bytes which need to be mentioned.

First of all, you need to know that the 2 '0's' are thrown away well before the sync bytes get stored in memory. In fact, it is this very throwing away process which makes these bytes do their job. Secondly, the sync bytes mark the beginning of the two areas into which the sector is divided - the address field and the data field. Let's have a look at the address field first.

The Address Field

To help along the discussion of these two sector parts, I have reproduced a block of data from my raw nibble dump of track $01. Don't worry about locating this same patch of data in your own dump. Just compare it with any group of bytes which begins a longish string of 'FF's'.

```
          address field sync bytes                 address field header
          /  /  /  /    /  /  /  /                          \  \  \
41A0 - FF FF FF FF   FF FF FF FF   FF FF FF FF   FF D5 AA 96
41B0 - FF FE AA AB   AF AE FA FB   DE AA EB AC   F7 FF FF FF
          \ /   \ /    \ /   \ /     \  ¦ /   \   /  ¦  ¦  ¦
       volume track  sector check-  trailer  garbage data field
                             sum                     sync bytes
```

The address field is the first part of each sector and is preceeded by about 8 - 14 'FF' sync bytes. They ensure that DOS is correctly locked in to read the bit stream just before encountering a sector's address information so as not to overlook the crucial information that follows. This information begins with the address field "header" or "prologue", whose purpose is to tell DOS, "the address field starts here - get ready to analyse it". On normal disks, this header is a string of 3 bytes which starts with the two reserved values, 'D5' and 'AA'. The third byte in a DOS 3.3 header is '96' as shown above and in a 3.2 header it's a 'B5'. These reserved bytes are never used anywhere on the disk to represent file code, which ensures that DOS doesn't mistake file data for a header.

DOS then has its hands full for the following few microseconds during which it has to carry out a surprising number of operations. It next has to read in, decode, process, and store (in locations $2F - $2D) the volume, track, and sector numbers. During all this it also has to compute the "exclusive or" product (explained in Appendix A) of these 3 values with a fourth, called the "checksum". If the result of this operation is zero, DOS carries on its work; if not, it bombs out and you get the dreaded I/O ERROR message.

This points up the function of the checksum, which DOS puts in each sector's address field whenever you INIT a disk. The checksum is calculated by combining the crucial information in a series of exclusive or's as follows: volume number EOR track number EOR sector number EOR $00 = checksum. DOS always writes this result on the disk in the spot shown in the last illustration

Whenever reading or writing a sector, DOS uses the checksum to verify that the address field data has been correctly read. To do this, DOS' RWTS routine performs another series of EOR's of the address field info. What it ends up with is a $00 after the '=' sign like this: volume number EOR track number EOR sector number EOR checksum = $00. If a result other than $00 is obtained, it means that the address field information of that particular sector has been incorrectly read or previously corrupted.

After all that work, DOS still has to verify the trailer to make sure that it reads correctly. This is a kind of extra check to be sure that the information DOS has just processed was actually from a data field. Here we arrive at a strange anomaly in DOS. Although during an INIT it always writes the 3 trailer bytes shown in the last illustration, it only checks the 'DE AA' bytes when it reads a sector. In fact you'll occasionally notice when examining disks that the second digit in the 'EB' has somehow gotten changed to something else, altering the byte's value to, say, $E9 or $E7.

The additional trailer byte may have been included to give DOS a small amount of extra time before ingesting the upcoming data field, before which it has to compare the volume, track, and sector numbers with those requested. In any case, DOS does get a little breathing room because the trailer is often followed by several bytes of garbage left over from previous recordings on the disk, plus another 4 or 5 sync bytes. Only after this interval, does the RWTS have to begin dealing with the data field (which I'll deal with myself a little later).

Data Encoding Constraints

You'll notice in my last illustration that each of the first 4 items of address information in line $41B0 consists of 2 bytes. For example, the volume number, which by default is $FE (decimal 254), is represented by 'FF FE' in the address field. But why not write it in a single byte as 'FE' and do likewise with the other items? The answer lies in some serious operating constraints imposed both by the hardware and by the necessity of locating the beginning of each byte in the bit stream.

Unfortunately, there are several such constraints. For example, one of the requirements of the sync process described above is that any byte that passes between the machine and the disk drive must have its 7th bit set to a '1'. So, as you can see from the chart in Appendix A, we are already precluded from

storing any byte on the disk whose value is less than $80.

Moreover, other DOS 3.3 hardware restrictions dictate that no byte on the disk can contain more than one pair of consecutive '0' bits. So, for example, 11111000 ($F8) or 11001100 ($CC) would not be permitted. This dictate raises the minimum byte value to $96. Furthermore, the bytes 'D5' and 'AA' are reserved for use only in headers and trailers. So after all is said and done, DOS 3.3 is left with only 64 permissable values within the $96 - $FF range with which to represent data on the disk.

The DOS 3.2 hardware was even less generous. It has the high bit rule, the 2 reserved bytes, plus its hardware's more stringent requirement that no disk byte can contain any consecutive '0's'. That means that in 3.2 there are only 32 allowable values range from $AB to $FF.

If this weren't enough hassle, DOS has the additional burden of time constrants when it reads the address field information. It must assmilate and process each byte in exactly 32 microseconds (4 per bit) - no more, no less. Moreover, extra logical and storage operations take place during the processing of this information, so DOS is placed under considerable pressure.

Since the full range of hex values can't be stored on the disk, a single byte of address field information (or any other data for that matter) needs to be translated into a representation longer than one byte before being transferred across. When this data is later retrieved from the disk, a reverse translation has to take place. Let's first take a look at how this is done with the address field values before looking at how ordinary files are handled

The 4&4 Encoding Technique

Since the address field data can conceivably take on any value, DOS needed an encoding technique for representing every byte from $00 - $FF (256 in all) using a substantially restricted range of numbers and the technique had to be lightning fast. Fortunately, the '4&4' encoding technique I have been previously talking about neatly solves these problems.

Now for the bad news. It takes 2 bytes of 4&4 to represent 1 byte of normal hex. Nevertheless this can't be avoided, so follow along with me while I do a 4&4 encoding of the value $0E to obtain the 'AF' and 'AE' bytes which represented the track number in the previous illustration.

Step 1: Use Appendix A to represent $0E in binary

$0E = 0 0 0 0 1 1 1 0

Step 2: Lift out the odd numbered bits

```
  0   0   1   1
  |   |   |   |
    0   0   1   0
```

Step 3: Shove '1's' in front of each bit

1 0 1 0 1 1 1 1

1 0 1 0 1 1 1 0

Step 4: Consult Appendix A to get the hex translations

1 0 1 0 1 1 1 1 = $AF

1 0 1 0 1 1 1 0 = $AE

Of course, DOS doesn't have to bother with Appendix A to do any translating, since microcomputers only process binary data; displaying this data in hex is merely one of their concessions to our mental frailties. This means that DOS doesn't do steps 1 and 4, so only 2 operations are involved in encoding an ordinary hex value into its 4&4 equivalent. Decoding it is equally as fast and easy, since it simply employs a two-step process which is the reverse of the one above. Other disk encoding techniques have to rely on looking up bytes in a special table, and hence use up extra valuable clock cycles (I'll be talking about these other techniques later).

How to Translate the Address Field Header

Getting the ordinary hex equivalent of the address field bytes is even easier for the lucky possessors of The Linguist than it is for DOS. All you have to do is place the cursor on the first byte after the 'D5 AA 96' data field header (an 'FF' on most disks):

```
                                                addressfield header
                                                      \  \  \
41A0 - FF FF FF FF   FF FF FF FF   FF FF FF FF   FF D5 AA 96
41B0 - FF FE AA AB   AF AE FA FB   DE AA EB AC   F7 FF FF FF
        \
      place cursor here
```

The translation will appear in the separated line of data in the bottom left of your screen as shown below:

```
buffer address of 1st byte  volume  sector
                         \     /     /
                       41B0: FE 01 0E F1
                                  \     \
                                   track  checksum
```

The 4-digit number at the beginning gives the memory address in the Linguist's buffer that contains the first byte the cursor is sitting on ('FE' in this case).. So we know immediately that the 'FF' and 'FE', shown in the illustration before the one above, are at memory locations $41B0 and $41B1. The rest is pretty self-explanatory - all but the sector number, that is. This requires some extra discussion which I'll get to in a second.

The considerable speed advantages of 4&4 over other encoding techniques have not gone unnoticed by top games programmers. In fact, a number of current games now use 4&4 to store their high resolution images on the disk. This is one of the factors which accounts for the striking speed with which pictures appear on your screen during the course of play. This speed is usually further enhanced by doing away with sectoring altogether and loading an entire track into the high-res screen "on the fly" (i.e., one byte after the other as quickly as it comes off the disk). The extra storage demands of 4&4 are usually offset by "crunching" (i.e. reducing the code necessary to represent) the high-res images by amounts claimed to be as high as 2000%.

These techniques also offer the games pros protection of sorts, since nibble copiers need considerable extra coaching to copy sectorless disks. Moreover, the additional difficulty of "cracking" 4&4 encoded files is often enough to discourage all but the most obsessional enthusiast. Furthermore, until The Linguist came in out of the cold, no commercially available utility was capable of translating 4&4 disk code. This meant that top games writers' could rest assured that their assiduously guarded programming secrets were unlikely to be rumbled by every hobbyist with a-disk patching program and a smattering of machine code.

These considerations aside, you can easily cope with such disks, and I will be coming back to them in greater detail when we work on The Linguist's data translating facilities. Now let's have a little what about DOS sector numbering.

Logical vs. Physical Sectors

In fact, there are 2 types of sector numbering systems - the physical and the "logical" . The physical sector numbers are those physically present on the disk itself. This is the type of sector that appears in the Linguist's translation of the address field data. On a 3.3 disk, these numbers are run consecutively on the disk. However, since tracks are circular, sector $0F is followed by sector $00 both on the disk and in the buffer. You'll know instantly when you're looking at sector $00 because its address field sync gap contains a much larger number of 'FF's' (usually more than 50) than do the other sectors.

This arrangement makes it possible to find a given physical sector quickly in the Linguist's display by counting up blocks of sync bytes while paging through the buffer. For example, if you

wanted to get to sector $02 from $00, paging forward through two strings of 'FF's' would bring you to its address field.

However, although this is how sectoring works in a raw nibble dump, the story in an RWTS read, and hence when using Tricky Dick, is entirely different. Here, the physical sector numbers get translated into completely different logical sectors. This means that when you select a particular track and read, say, sector $04 with Tricky Dick, you won't get the same sector data that appears in physical sector $04 in a Linguist dump. In fact, what you is the data from physical sector $07, as shown in the illustration below.

This procedure was not incorporated to annoy and confuse us, but, just like the universal use of hex instead of decimal, to make life a lot easier. In fact, the system of logical sectors adds considerable speed to all DOS operations. To see how it wbrks, hit CTRL C twice to get the BASIC prompt up and type in CALL -151, followed by BFB8.BFC7. You'll end up with the following display (without my labels,, of course).

```
logical sectors:  0  1  2  3  4  5  6  7
                  /  /  /  /  /  /  /  /
physical sectors: BFB8- 00 0D 0B 09 07 05 03 01
physical sectors: BFC0- 0E 0C 0A 08 06 04 02 0F
                        \  \  \  \  \  \  \  \
logical sectors:  8  9  A  B  C  D  E  F
```

This is the "sector translate" or "soft skew" table which is used to bring about the "sector interleaving" or "sector skewing" process. DOS translates the physical sectors it reads from the disk into logical sectors by counting up in the above table. For example, physical sector $0B occupies byte number $02 in the table, so physical sector $0B is logical sector $02.

When you start using The Tracker, you can see this process in action and compare both types of sectors as the data itself is being read off disk. This is because The Tracker allows you to select a display of either logical or physical sectors for any DOS operation.

The speed advantage of sector skewing come from the fact that after DOS reads in a sectorfull of data, it has to digest it a bit before moving on to the next sector. Trying to do this in time to read the very next physical sector on the track, is impossible, and would clearly result in a grave case of indigestion.

In fact, while DOS is processing one sector's data, several more sectors pass under the head. By staggering ("interleaving", "skewing") the sectors in such a way as to avoid accessing them consecutively, a considerable number of unnecessary disk rotations are eliminated during DOS's work. The table above helps perform this function for DOS 3.3. You can see that although the logical sectors (those specified by RWTS) run

consecutively, their physical sector equivalents actually being accessed from the disk are not in consecutive order. The physical sector order shown in the above table gives DOS time to process one sector's data before the next appears under the read/write head.

So if you can't read a particular sector number with Tricky Dick, you can translate it using the above table into its physical sector number, and use the Linguist's translation functions to have a look at the code it contains. I'll be guiding you carefully through this process later on in the chapter under "Translating Raw Disk Data with the Linguist".

DOS 3.2 used a different method of skewing its 13 sectors. Instead of resorting to a table look-up procedure as does 3.3, its initialization routine staggers the numbering of the physical sectors on the disk. So if you look at a 3.2 disk, you will see that the numbers are not consecutive from sector to sector, but run like this.

00 0A 07 04 01 0B 08 05 02 0C 09 06 03

The normal skew factors for both versions of DOS provide a kind of middle ground where all types of DOS operations receive roughly equal speed benefits. However, the advantage of the 3.3 method is that it is readily modifiable by simply switching around the values in the soft skew table, whereas with 3.2, the formatting of the disk itself would have to be altered. Sometimes, altering the sector access order speeds up certain DOS operations.

The Data Field

Let's have another look at my display from the discussion on the address field. This time I'll show you lines $41B0 and $41C0 with labels pertaining to the data field.

```
                                                  datafield sync bytes
                                                      \  \  \  \
41B0 - FE AA AB AF   AE FA FB DE   AA EB AC F7   FF FF FF FF
41C0 - FF D5 AA AD   DE DB FB FB   DE DE E7 BF   EB FE A7 BF
           \  \  \
   data field header beginning of sector's file data
```

This begins where we left off in our examination of the address field, and illustrates how the data field is structured. It starts off with far fewer sync bytes (usually 4 or 5) than does the address field. These are followed by the 'D5 AA AD' header which is the same on all standard 3.3 and 3.2 disks. The header enables DOS to identify the beginning of the sector's data field which starts in the above illustration with a 'D9' byte.

In order to gain some more familiarity with all this, you might try finding the beginning of a data field and verify the above marks. Then hit the right arrow once or twice and look for

the very next sector's address field sync bytes. Near the beginning of them, you'll see the data field trailer, which is the same as that of the address field, a 'DE AA EB'. Here's an example from one of my disks of what I mean.

```
 last bytes
 of sector     data field
 $0E's data    checksum          more garbage
 /  /  \  \    /                 /  /  /  /
96 9E 96 CD    A7 DE AA EB       E7 CF F3 FC    FF FF FF FF
                  \  \  \                       \  \  \  \
            data field trailer          sector $0F's address
                                          field sync bytes
```

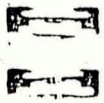
As in the address field, a few bytes of garbage may sometimes be seen among the valid marks. This is no problem and is ignored by DOS. What is important, however, is the checksum, which as you can see above is a single byte of data. It works in much the same way as the address field checksum, in that it is the product of a series of Exclusive OR's. Each byte of the raw sector data is EOR'd to the next until the end of sector is reached. The result is then written out as the last byte. This is verified by DOS just after the sector's nibbles are read into RAM.

You also need to remember that DOS treats the address and data fields entirely differently in working with any disk. Both are formatted and assigned their respective sync bytes, headers, trailers, etc., when you INIT a disk. However, each time DOS puts a new sectorfull of file data to the disk, it rewrites the data field's sync bytes, header, trailer, and checksum. This differs from DOS' treatment of the address field, which it never touches after it once INITs a disk.

Digging through the Raw Data

When you first flick through The Linguist's display, you will probably take one look at the row upon row of densly packed numbers in the various sectors' data fields, and rapidly reach for your copy of Space Zappers. What kind of weirdo could possibly make sense out of this endless field of alphanumeric gibberish? Well, let me tell you about a strange mental phenomenon, as yet unexplained by cognitive psychology, that most disk snoopers experience fairly early in their endeavors.

You will soon find, after very little practice at all, that you can merely glance at a data field of raw file nibbles on almost any disk and be able tell instantly whether it's valid code or garbage, whether it's the product of 3.2 or 3.3 DOS, and which of the 3 encoding schemes has been used to write it out. What is even more surprising, you'll find yourself doing this without being able to remember the value of a single byte that appeared in the display. What you'll develop is a kind of global perception in which the mere "look" of a screenfull of raw hex will tell you all you need to know.



As I mentioned earlier, one important sign that might lead you to look more closely for 4&4 encoding is a lack of sectoring on a track you're examining. In this case, you can often see blocks of sync bytes at regular intervals throughout the track, but no headers or trailers.

Since this is supposed to be a hands on tutorial rather than a lecture course, probably the best thing for you to do right now is to compare the above formatting and encoding differences by reading and examining both a 3.3 and a 3.2 disk with The Linguist. Now is also the time to have a look at any originals of games disks you might happen to have in order to see if you can find some 4&4 to practice on. Surprisingly little practice will develop your eye to the point where you can pick the correct DOS and encoding mode used in a matter of seconds. However, before we actually ask the Linguist to do any translating, you have to be able to tell if a sector contains ....

Goodies or Garbage?

Another knack you'll find useful is the ability to determine whether a particular track or sector is filled with viable code, or whether it contains invalid or corrupted data. Being able to do this will, among other things, save you trying to translate a useless block of disk nibbles into a program listing. As you'll discover in the next chapter, this skill will also help you find out if a disk contains any hidden "half tracks" (those written between the pathways on the disk that normal tracks occupy).

Garbage tracks and sectors can be most readily spotted by looking for illegal bytes in the data field. Certain easily recognizable illegal bytes are common to all 3 encoding modes and are shown below.

95 or less	
98 - 99	D0 - D2
9C	D5
A0 - A5	D8
A8 - A9	E0 - E4
B0 - B1	E8
B8	F0 - F1
C0 - CA	F8

A few rules of thumb can easily be deduced from these values: look for bytes smaller than $96, and those ending in '0', '1', or '8'. If you find more than a couple of such bytes in a single sector's data field, you should strongly suspect that the nibbles don't represent valid file code. If only a few are present the sector might be worth translating anyway, but if a lot are there, don't bother - the data is likely to be meaningless.

Translating Raw Disk Data with the Linguist

Now we're going to get The Linguist and Tricky Dick working together on a type of espionage that no other disk utility

available at the time of this writing can carry out. Make sure that both programs are loaded into memory, then place your original System Master diskette in your drive. Next read in track $11, sector $0F and take a look at the catalog entry for HELLO. The fouth and fiftn bytes in line $08 should tell you that the HELLO's track and sector list is on track $13, sector $0F.

Examining this list tells that HELLO starts on sector $0E of the same track and runs down to sector $0A. This is our cue to jump into The Linguist with CTRL E. Now select track $13 and press CTRL R. Our job is to find the sector which contains the beginning of the program, but we know from our previous discussion of that, we won't be looking for sector $0E. To determine the physical sector we need to translate, take a look at the sector skew table shown a few pages back. This tells us that logical sector $0E translates into physical sector $02.

To get sector $02 up on The Linguist's display, start looking for a gap of address field sync bytes at the beginning of The Linguist's display. If none are present there, you're sure to find some on the next page by hitting the right arrow. Now put your cursor over the first byte after the 'D5 AA 96' address field header. The data line at the lower left of the screen will tell you the physical number of the sector you are examining. What you have to do now is start flipping pages by pressing the right arrow some more while also counting sync gaps. Start your count on the sector whose number you just translated and increment it each time you come to an address field sync gap. When your count reaches $02, you have arrived.

Check to be sure that you're on $02 by putting the cursor on the volume number and reading the sector number at the bottom of the screen. If you have indeed arrived at the right sector, take a look at the encoding mode indicator at the upper right of the screen.

```
                                <00>
                                <--)
present encoding mode --> <62>
```

This should indicate that you're in the 6&2 mode as shown above. If not, hit CTRL D and type in a '6' (you would of course type a '5' if you wanted 5&3, or a '4' for 4&4).

Now The Linguist is ready to show off some unique translating skills. Place the cursor on the first byte after the 'D5 AA AD' data field header. This should be a '9A'. Now hit CTRL T and the entire sector will be translated and moved to Tricky Dick's data buffer instantly. A subdued squeek will be heard when this occurs.

To view the translated data, hit CTRL C followed by RETURN. Tricky Dick's first display line should look like that illustrated below.

```
00: 71 04 19 08 0A 00 B2 20
                       \
                        put cursor here
```

Place your cursor over the first byte after the '00' (the 'B2'). We have good reason to believe that the file is in Applesoft, so hit CTRL L followed by 'A' to select this type of listing. Finally, press 'L' and you should see the first 9 lines of HELLO in its original BASIC. The first 3 lines should look like like this.

```
REM  -- DOS 3.3 HELLO
20 REM
30 TEXT: PRINT
```

If you feel like some more practice, go back into The Linguist and find HELLO's next sector in the raw nibble dump. This is $0D which translates into a physical sector number of $04. You just translated $02, so you only have to count 2 sectors forward. To get some further practice, go back to the catalog track and find ANIMALS its 'B' file type marker tells us that it's an Integer file. Find the track its first sector is located on and carry out the same operations that you performed on HELLO. The only difference is that you must select 'I' for an Integer listing after pressing CTRL L.

Finally, it's probably a good idea to translate a binary file just to round off your mastery of The Linguist. APPLESOFT, whose file entry can also be found on sector $0F of the catalog track, is a good candidate. When you get to Tricky Dick this time, don't forget to select a '*' for an assembly listing

Although a sequence of several actions is called for when you want to translate a raw nibble dump, you will find that after practicing on only a few programs, that the actions will become automatic. Furthermore, this procedure has a number of important uses which can be applied to any disk.

For example, suppose you have a blown disk and you find that the formatting of some of the sectors is so badly damaged that no amount of effort with Tricky Dick enables you to read them. You can use The Linguist and Tricky Dick in concert to get a printer listing of the missing code. This will enable you to rescue any intact file from any blown disk, no matter how badly the disk itself is corrupted. Or, let's say you have a disk whose formatting has been heavily altered by its publisher, but whose code you're dying to have a look at. Again, these two utilities can work together to satisfy your curiousity about any disk presently available.

In fact, I suspect that you've been waiting impatiently throughout the last 3 chapters for an exposition of that most forbidden of all topics, spoken about only in a guarded manner in front of your local dealer, and treated like top military secrets by the users' mags. Well, you won't have to wait any

longer... If you've followed me this far, you've acquired the prerequisite disk lore to understand in considerable depth most of the principles of - dare I utter the words - Copy Protection. The Linguist and Tricky Dick form a unique and infallible team which will help you read and make sense of the code on any protected disk and enable you to edit somewhere in the region of 80 percent of them. Not to mention, of course, that you'll also learn how to put some diabolical disk locks on your own software. So all you have to do now to lift the veil of secrecy that surrounds this subject is turn the page.

CHAPTER FIVE — The Secrets of Software Protection

In the last chapter, you got a brief introduction to a few of the unique achievements attainable with the powerful combination of Tricky Dick and The Linguist. Now I'm going to show you how to use this irrepressible duo not only to carry out sophisticated software espionage, but also to sweep aside most of the counter-measures presently employed by software writers and publishers to prevent such activities. Having already learned how to read any disk you wish, you are ready to play an even more active role - that of actually editing copy protected code. This chapter will equip you to cope with most forms of nonstandard formatting so that you can repair locked disks, patch them, and investigate their software to the fullest possible extent.

IMPORTANT NOTE: DO NOT ATTEMPT TO WRITE ON THE ORIGINAL VERSION OF ANY PROTECTED DISK. DOING SO COULD IRREPARABLY CORRUPT IT AND INVALIDATE ANY ENTITLEMENT TO A REPLACEMENT FROM THE COMPANY CONCERNED. ALWAYS MAKE A COPY FIRST AND WORK EXCLUSIVELY WITH THAT.

Sorry, I felt that I had to get that boring but important disclaimer off my chest right up front. If you choose to ignore it, remember - you have been warned. Now let's get down to some more exciting issues.

Many a lonely hour and much shoe leather have been expended by expert programmers while walking the streets and riding an angle to produce an unbeatable disk lock. As a result of their Herculean efforts, a plethora of strategies, some craftily ingenious, others crudely simplistic, have been invented to corral wandering program code. Though highly numerous and varied, however, almost all copy protection tricks can be classified under one of only two categories: formatting changes and timing dependencies.

The first and by far the more common of these involves tampering with the same disk formatting components you dealt with in the last chapter. Indeed, it is mere child's play to produce a disk whose headers, trailers, checksums, sync bytes, address information, location of the VTOC and catalog, etc., have been altered. Most such protection methods can be applied in seconds by changing a couple of bytes in DOS and INITing a disk. The funny formatting is then recognized by the DOS on the INITed diskette, but not by normal DOS which is used by FID and COPYA. This means the disk will boot and run normally, but you won't be able to back it up using a standard copy program.

By the way, in case you're curious about how to actually protect your own disks with some of the above tricks, stay tuned and keep alert - as you work through my instructions, some of these secrets will be revealed to you in a most unexpected way.

The second protection category, that of making disk operation timing dependent, usually involves a lot more than just tweaking DOS a bit. Special routines are employed which either time the reading of or count up a block of nibbles (usually sync nibbles) somewhere on the disk. If these routines fail to find the quantity of nibbles they are looking for, or are unable to achieve the required timing, they abort the booting and/or running of the programs, usually wiping memory in the process. Even with the skillful use of nibble copiers, it is often quite difficult to get the number of these crucial nibbles exactly correct on a copy. This is due in major part to the substantial timing differences between Apple drives, a factor which software publishers often greatly enhance by drastically altering the speed of the drive they use to make the original.

First Steps in Reading Protected Disks

You'll know soon enough if a program you are interested in is copy-protected when you hear the familiar sound of disk suicide on your first attempt to read it with Tricky Dick. The accompanying 'I/O', flashing merrily away at the top of the display, is Tricky Dick's way of telling you that this is where the fun begins. With the supreme disdain such trivial interruptions to your enjoyment deserve, you next carry out a couple of simple checks.

The first of these is pretty obvious, but is sometimes overlooked in one's zest for investigating more intriguing possibilities. And that is simply to try toggling the Tricky Dick DOS version to 3.2 (if you are in 3.3), followed by a reread, to eliminate the possibility that the "protection" method involves nothing more than merely formatting the disk to boot under both DOS types. Such a disk usually contains a 3.2 format, though it is designed to work equally well with 3.3. Naturally, a switch to the other version should also be tried first if you originally selected 3.2 to read the disk and were unable to do so.

If a DOS toggle doesn't cut any ice, the next step is to hit CTRL S and make a few changes in Tricky Dick's RWTS parameters. This command starts things rolling by puting the cursor at the beginning of the DOS sector mark display. After your work in the previous chapter, these should now be like old friends to you. Just to make absolutely sure that we're both in the same ball park, however, I'm reproducing Tricky's DOS mark display below for reference.

```
 cursor comes here after CTRL S
 /
D5AA96 Y DEAA --- address field: header, checksum status, trailer
D5AAAD Y DEAA --- data field: same as above, but for READING only
D5AAAD 0 DEAAEB - data field: same, but for WRITING only
```

Start out by using the right arrow to skip over the 'D5AA96' address field header and place the cursor on the 'Y' directly

following it. Any time this letter is visible, Tricky Dick will compute the address field checksum in the normal way before attempting to read the data field following it. So let's type an 'N' over the 'Y', eliminating this check during any subsequent RWTS read. Now hit the right arrow once more and change the 'DEAA' address field trailer to '0000'. This tells Tricky Dick to ignore it from now on also.

Pass over the 'D5AAAD' data field header and change the next 'Y' to a 'N'. As you might suspect, this makes Tricky Dick's RWTS forget about the data field checksum. Finally, move on and change the 'DEAA' data field trailer to '0000', finishing off by hitting RETURN. These adjustments should leave the upper left corner of Tricky Dick's display looking like this.

```
D5AA96 N 0000
D5AAAD N 0000
D5AAAD 0 DEAAEB
```

What sort of reliability might you expect if you attempt to read a disk without the benefit of the verification these marks provide? In fact, 3 of them represent a prime case of overkill by the author of DOS in his (her?) effort to give you error f

Chances are that once you find a sector accessible to Tricky Dick, there will be several more near it which can also be read. The best place to start is track $00, sector $00, and then move to the other sectors on the same track. After that, try tracks $01 and $02 before looking around the rest of the disk.

By the way, although I'm only talking for the moment about getting some protected code into Tricky Dick's screen and eyeballing it, this skill will pay off later when it will enable you to do important work with protected diskettes - such as finding and fixing a clobbered VTOC and/or catalog, hunting down the start of a program, discovering and changing the various levels and options of a game, etc. In addition, it will even allow you to protect your own disks. But let's not move ahead too rapidly just now.

All other considerations aside, the foregoing few simple changes will enable you to read from (and write to a copy of) a surprising number of locked disks. Of course, these changes represent only the preliminaries; the CIA utilities have a number of other methods up their sleeves for extracting the information you require in order t

Of course, the CIA has even more tricks for reading the unreadable, and I'll be talking about these in due course. In the meantime, I need to describe briefly how you go about writing an altered sector back to a nonstandard disk (that is, a copy of such a disk, if it happens to be protected). The only further adjustment you need to make concerns the data field marks shown in the third line of the sector marks display. This line, if unchanged, looks like the illustration below.

D5AAAD O DEAAEB

Every time a sector is written on the disk by RWTS, its data field header, checksum, and trailer are rewritten. These 3 marks are repeated in the third line of the display so that you can read a sector in and write it out with a different header or trailer. Let's say you want to copy a [illegible] drive one to another disk in drive two. Let's assume the drive one disk's data field header and trailer are D5BBAD and DDAA, respectively, and you want to change these to D5CCAD and DFAA. The appropriate data mark display should look like the following, minus my comments.

```
D5AA96 Y DEAA    (normal address field header)
D5BBAD Y DDAA    (data field header values of drive 1 disk)
D5CCAD O DFAAEB (data field header values to go on drive 2 disk)
```

Your only remaining task would be to read the sector from drive one and write it to drive two. This technique is quite handy for protecting your own disks, a topic which I will be discussing at length a little later on.

Don't worry too much about the extra 'EB' at the end of the third line. Although you can change it at will, it is ignored by RWTS and has been included in the display merely to allow you to distinguish at a glance the data field marks used for writing from those used for reading. The 'O' in the middle of the third line has no function except to remind you that the data field will be written to disk with a checksum of zero.

Just to give you some practice in dealing with altered formatting, I have "protected" track $22 of your CIA disk using a variety of the formatting changes you're likely to come across in commercial software. This, and a hidden half track I'll soon be discussing, are the only part of your CIA disk which is protected. However, after completing this chapter, you will have no trouble copying track $22 across to your CIA backups if you wish. Just be sure to zero out the track $22 bit map in the copy's VTOC, as I described when we worked on Tricky Dick together.

Following my instructions above, you might want to immediately try putting 'F's' in the checksums and '0's' in the trailers in order to read track $22. Let me save you the bother

As part of our fun in this chapter, I'm going to ask you to write down one or two things about track $22, and also to try to read some of its sectors. Nothing regimented or school-marmish, though - just a few quick projects to get you talking to Tricky Dick and the Linguist a bit more fluently. Later on, you can check your results against mine which I've recorded on the last page of this chapter.

Your first assignment is to check the sync bytes and make a note of them in hex. Paradoxically, altered sync bytes in no way interfere with a normal RWTS read, but do put something of a strain on nibble copiers. This is because RWTS finds valid data by simply recognizing the data field header and reading in the next 256 bytes. Nibble copiers, on the other hand, cannot be dependent on this header, since its 3 bytes can take on a large number of values. This means that they have to home in on the sync bytes in order to find the beginning of each sector (and often also the track start). So if you're trying to copy a disk whose sync bytes have been altered, you'll usually have to tell your favorite copier what they are by means of one or more parameter changes.

Having written down the sync bytes, you're next job is to read the data on sector $00 of track $22 using Tricky Dick. You'll remember from the last chapter that Tricky Dick reads logical sectors, while The Linguist's dump shows only physical sector numbers. Fortunately, logical sector $00 is also physical sector $00 which means you won't have to use the chart below to find it in The Linguist's display. You'll also recall from the last chapter that sector $00 is easy to distinguish from the other sectors in a raw nibble dump because its address field sync bytes usually outnumber those of other sectors by about 3 to 1.

After finding sector $00, all you have to do is check out its headers and trailers, jump back to Tricky Dick, put any that are non-standard in the DOS marks display - and voila - the sector magically appears on your screeen. If all this doesn't work quite so magically on the first attempt, hang in there and keep trying. Resist the temptation to look at the explanations in the back of the chapter, and instead, hit CTRL E to check out the raw data again.

Got it? Good! Now have a shot at reading sectors 1, 2, and 3. They get progressively harder as you work through them in that order. Consult the chart below to translate the disk's physical sectors into logical sectors for Tricky Dick's benefit. Remember that a complete explanation is at the end of this chapter in case you truly get stuck. When you examine sectors 2 and 3, I'd like you to write down the data field headers and trailers you find there. They will provide you with some interesting surprises which I'll be discussing further in the answer section.

[illegible] won't work (you didn't think I

Logical Sector:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Physical Sector:	0	D	B	9	7	5	3	1	E	C	A	8	6	4	2	F

If you've arrived at this point successfully, writing to an altered format will certainly prove to be no sweat. All you have to do is get the right DOS marks in the Tricky Dick display and use CTRL W in the usual way. I'll explain more fully the various horizons this opens up when I show you how to use The Tracer. In the meantime, if you want to practice writing to track $22 just to get the feel of it, be my guest.

How to Edit Half Tracks

As you may recall from Chapter 4, it is possible to format a track half way between one of the 34 pathways normally set aside by DOS for track placement. By writing out a "half track" in this middle position, the disk protecter insures that it will be always overlooked by the stand-DOS during any disk operation. However, his ingenious machinations will not prevent you from getting at the goodies with Tricky Dick and the Linguist.

Under normal circumstances, you will need to use The Linguist as shown in the last chapter to discover the location of any half tracks on a protected diskette. However, in order to walk you through a half track editing job here, I've formatted one between tracks $20 and $21 on your CIA diskette. This means you'll have to use your original CIA disk for this exercise, since the half track won't have gotten transferred to your backup.

If you've just been practicing on track $22 of your CIA disk, start by BRUNning Tricky Dick to reset its sector marks. In any case, you must always "clear" its RWTS by reading track $00, sector $00 on your CIA disk. This precaution will cancel any previous misreads that may have been flagged within RWTS, and hence will avoid a possibly fatal recalibration at a later stage in the proceedings.

Now hit CTRL C which takes you back to BASIC. Then type CALL -151, followed by 3DF4:D0 1A N 39A4: D0 56 EA. Go back to Tricky Dick with a CTRL Y plus RETURN.

Next jump to The Linguist and read track 20 1/2. To do this, hit the ';' key, set the track number to $20, and then press the '>' key once. This changes the display to '20+HALF'. Now hit CTRL R to do a nibble read. We aren't in any way interested in the raw nibble dump from this track, however, but are simply reading it so as to position the disk arm over the half track we want to get at with Tricky Dick.

Finally, go back to Tricky Dick and hit CTRL R. The sector should be read in and you should see the message "THIS IS A HALF TRACK", followed by the number of the sector you've just read. If your drive is not in perfect adjustment, this may not work the first time. If it doesn't, have another try. If you get more problems, have your drive checked out.

But what was the 'D0 1A' for? Well, first off you need to know that whenever RWTS is instructed to move the diskarm to a given track, before doing anything, it asks the question, "where is the arm at this very moment". To get the answer, it reads the number of the track over which the arm is presently sitting. This is easy enough since the address field information of every sector contains the number of the track it belongs to. If RWTS finds that it's already placed over the desired track, then it starts looking right there for the sector that was specified. But if not, it uses the present track as a reference point from which it counts up or down to get to the track it really wants.

What we have just done is to patch Tricky Dick's RWTS so it thinks that the arm is already correctly positioned, no matter what number the number of the present track is. Thus, it searches no further, but immediately goes looking for sector you requested.

You can now read from or write to any or all of the sectors on the half track, AS LONG AS NO RECALIBRATION TAKES PLACE. This could happen if for some reason RWTS had trouble accessing a sector. What you may then get is the sector you requested, but from track $00. If this happens, simply jump to The Linguist, reposition the arm by reading track $20 1/2, and try once more. And here from my vast collection is yet another IMPORTANT NOTE: If you don't see the message "THIS IS A HALF TRACK" written in the first few bytes of any sector you now read in, don't write anything on it - you're in the wrong place and must go back to square one.

Of course a half-track may have also had its headers, trailers, or checksums diddled with, so you'll want to check these out with The Linguist before using Tricky Dick on it. Also, it goes without saying that when you are through editing a half track, you should fix up Tricky Dick's RWTS so it will work properly. So always remember to go into the monitor and type in 3DF4:AD 78 N 39A4: CD 78 04, or better yet, BRUN Tricky Dick.

If you're working with a half-track on a 3.2 formatted disk, the patch for Tricky Dick's 3.2 RWTS is 35E7:1A 20 N 3222: D0 56 EA. Everything then works exactly as above. To put it back to normal, use 35E7:AD 78 N 3222:CD 78 04.

Editing Disks with Mixed Track Numbers

A protection method which has recently cropped up is that of changing the track numbers in the address field information so that some or all tracks have numbers greater than $23 (decimal 35). In fact, the usual pattern is to have track $00 designated as $FE and then to decrement this number for each successive track. Though permissible, setting Tricky Dick's track number to, say, $FE in order to read the real track $00 won't work here.

This is because RWTS will use the current track as a reference point to count up to the one it is supposed to find. Since it moves forward one track for each count, and there are only $23 tracks on the disk, the arm would be pushed against the stop trying to get to track $FE. This brings things to a halt and causes Tricky Dick to display an error message.

This is no big deal, however, since exactly the same procedure used to read half tracks will work for those with funny numbers. Put in the RWTS patch, read sector $00 of the CIA disk with Tricky Dick, and position the disk arm over the desired track with The Linguist. You should then be able to access the track's sectors with Tricky Dick.

PROTECTING YOUR OWN DISKETTES

If you've done your homework diligently during this chapter, you will have discovered a couple of quite effective protection methods already. Making use of Tricky Dick's ability to alter the data field marks and to patch directly to the disk will give you further tools to enable you to lock a disk up so tightly that only the most knowledgeable of nibble copier fans will be able to back it up. Keep in mind that the following changes apply to DOS 3.3 only. Put the newly initialized diskette in your drive that you produced in Chapter 2 (that's the one with track $23 initialized) and we'll start by playing around a bit with the data field on a couple of sectors.

Changing Data Field Marks

Boot up the Chapter 2 diskette and type in the following BASIC program:

```
 5 HOME
10 VTAB 14
20 HTAB 8
30 PRINT "I BET YOU CAN'T COPY ME!"
40 END
```

Now type SAVE HELLO so that the program is installed on your practice diskette. BRUN Tricky Dick and find HELLO's track and sector list (which is probably located on track $12, sector $0F). You can choose to protect this sector, the following sector which contains the program code, or both.

Lets try the T&S list first. Read it in with Tricky Dick and let's decide how evil we want to be. We can change this sector's data field trailer, header, or both. Let's say you want to change the data field header to D5 BB CC, and the data field trailer to FF AA. The first thing to do is set up Tricky Dick's DOS marks display to look like the following.


```
D5AA96 Y DEAA
D5AAAD Y DEAA
D5BBCC 0 FFAAEB
```

This allows you to read in a DOS 3.3 sector whose data field marks are normal, but to write that same sector back with its marks changed as shown. So all you have to do now is read in sector $0F (or whatever HELLO's T&S list sector is) and write it back. Do this and try typing RUN HELLO. If you have been successful, your disk drive will start to sound like a miniature cement mixer as DOS strives vainly to read your program. It will quickly give up and send you the dreaded I/O ERROR message. Just to admire your handiwork, load in The Linguist and dump the track upon which HELLO lives. Logical sector $0F (if this is where HELLO'S T&S list ended up) will be easy to find since it's the same as physical sector $0F on the disk. Just find sector $00's big block of bytes and back up one.

There is one crucially important step yet to be carried out - you've got to let the DOS on your protected diskette in on the tricks you've just been up to. If you neglect to do this, of course, you won't ever be able to LOAD or RUN HELLO. The problem here is that DOS needs both to recognize the changed marks on the sectors you've just doctored and the normal sector marks everywhere else on the disk. Does this mean you've got to do something drastic - like maybe rewriting RWTS or something?

In fact, the solution is absurdly simple. All you have to do is make sure your changed headers have at least the 'D5' byte in common with the normal headers. Then you patch DOS so that it accepts the next two header bytes, irrespective of their values, when searching for the start of the data field. In other words, you fix DOS to look for any header of the form 'D5 XX XX', where the 'XX's' can take on different values on different sectors. This means that you can have as many different data field headers on the disk as you like as long you ensure that they all start with 'D5'.

The rationale for this strategy is that your changed and normal headers must both have a reserved byte (i.e. either a 'D5' or an 'AA') in common so DOS doesn't accidently mistake 3 other bytes of disk data for a header. If you use the 'AA' for this, you end up with XX AA XX, which caused booting problems in my benchmark trials.

Now let's patch the DOS on your newly protected diskette so that it can handle varied headers. Change Tricky Dick's sector mark display back so that it shows the normal 'D5 AA AD' headers, then read in track $00, sector $02. Get lines $F0 and $F8 up on the screen and type a $00 in place of the bytes at buffer locations $F3 and $FE as shown below.

```
F0: C9 AA D0 F2 A0 56 BD 8C
                \
   change to:  00 .

F8: C0 10 FB C9 AD D0 E7 A9
                         \
            change to:  00
```

The values you are about to replace with '00's' tell DOS where to go if a misread of the header's second and third bytes occurs. In fact, the 'F2' and 'E7', plus the 3 numbers in front of each of them, say "IF the byte just read is not equal to the correct header value ('AA' or 'AD') THEN jump along $F2 or $E7 memory spaces to another set of instructions". The instructions in question stop everything and output an error message. Having now replaced the crucial bytes with '0's', write the sector back to the disk.

Your '00' patches tell DOS to jump along zero memory spaces if a misread occurs. In other words, they say: "if the header byte is incorrect, don't jump anywhere at all - just continue as if nothing happened". This insures that DOS will execute the same chain of instructions whether or not it encounters the values it expects when reading the second and third bytes.

Your last job is to fix DOS so that it will accept both the normal address field trailer bytes, and those you have altered. Here, there is even more latitude for creativity, since any two legal (and some "illegal") values will work. In fact, you are now going to tell DOS to keep on doing its thing, irrespective of what bytes it sees in the trailer.

Read in track $00, sector $03 and get lines 30 - 40 on the data display. Then change bytes $37 and $3F to '00'. Finish off by changing byte $40 to a 'D0', and write the sector back to the disk.

```
30: 8C C0 10 FB C9 DE D0 0A
                         \
            change to:  00

38: EA BD 8C C0 10 FB C9 AA
                         \
            change to:  00

40: F0 5C 38 60 A0 FC 84 26
     \
change to:  D0
```

This tells RWTS to branch 0 spaces if the first trailer byte is incorrect, just as before. The patches in lines $38 and $40 change the verification procedure so that RWTS looks for $00 as the second trailer byte, then continues normally if this value is not found. Since $00 is an illegal byte, and hence can't appear in the trailer at all, RWTS always keeps on as if nothing at all,

had happened. So DOS will now ignore the data field trailers entirely, allowing you to change these from sector to sector at will. Now try rebooting and, voila, "I BET YOU CANT COPY ME!" appears on the screen.

You can now repeat the foregoing tricks on the sector containing your HELLO program code. The most effective way to make use of these techniques is to change the data field marks on several sectors of a program you're trying to protect. It's also a good idea to change the headers from sector to sector within the same track. This latter measure makes the process of nibble copying the disk something of a horror show with many copy programs. Also try using data headers made up only of "illegal" bytes as explained in the answer section of this chapter.

Moving the VTOC

One place on the disk that gets overlooked by every standard Apple copy program (and most people using nibble copiers) is track $23. As I've already mentioned, DOS is not configured to use this track, even though it is completely accessable on 99.9% of all Apple drives (if your drive is one of the 0.1% that can't use this track, take it back to your dealer and demand that it be made to work properly). This makes track $23 an ideal place to hide something you want to keep from any would-be backer-uppers of your software. And what better item to conceal there than the VTOC, since no DOS operation can go forward without first visiting this vital sector?

With any disk which has been initialized and doctored to use track $23 as described in chapter 2, use Tricky Dick to read in the VTOC at track $11, sector $00, and write it back to any sector of your choice on track $23. Now read in track $01, sector $0B and change byte $01 from an '11' to a '23'. This tells DOS to go to track $23 to find the VTOC.

```
00: A9 11 8D FA B5 60 20 1D
         \
         change to a '23'
```

The only other patch you have to make is to track $01, sector $0F by changing its byte $0D from a '00' to whatever sector on track $23 now contains the VTOC.

```
08: B7 AE FA B5 A0 00 4C 52
                     \
          change to thesector number you selected
```

Finally, go back and zero out the original VTOC on track $11 by reading it in, pressing CTRL Z with the cursor on byte $00, and writing it back to the disk. If a normal DOS is used to boot or catalog the disk, the only thing that happens is that DOS puts the "DISK VOLUME 254" message at the top of the screen. If, on the other hand, you want a crashout to occur in this situation, simply write an $FF in byte $01 of the now zeroed

sector $00 on track $11. This causes a normal DOS to try to seek the diskarm to track $FF on a CATALOG or boot.

```
00: 00 00 00 00 00 00 00 00
      \
   change to 'FF'
```

You have probably deduced that the VTOC could be somewhwere other than track $23 by simply replacing the '11' in the first patch with the number of the desired track. However, you will need to protect the disk in other ways, since COPYA will scoop up the moved VTOC along with everything else, unless it is on track $23.

Wherever you decide to put the VTOC, be sure to alter its bit maps to show that the sector it occupies is not to be written to. While you're working on the maps, you might as well free up track $11, sector $00 also. That way you can use it for program storage. The bit maps are covered in detail in Chapter 2.

Protecting RAM

No matter how much you mess around with the disk formatting, there still remains one important potential leak to be plugged. A would-be program purloiner can simply hit RESET while your software is running, and then save it to a normally formatted disk. This is easily done by booting in DOS from a normal slave diskette after first moving any file code on pages $08 and $03 (the only program storage areas that are disturbed by this process) to a unused part of memory with the monitor move routine (described on pages 44-46 of the Apple manual). After the boot, it remains only to restore the code to pages $08 and $03 and to SAVE or BSAVE your program.

Fortunately, it's just as easy to take precautions against this process as it is to carry it out. First of all, a single byte DOS patch will effectively prevent anyone from interrupting your program with an autostart ROM RESET. Start by reading in track $00, sector $00 of the disk to be protected, and place an 'FF' in byte $37 as shown below.

```
30: AD 53 9E 8D F3 03 49 A5
                        \
                   change to 'FF'
```

Now write the sector back to the disk. When DOS is booted in, this patch will cause a flag (the "powerup byte" - see page 37 of the Apple manual) to be altered so that when RESET is pressed, your machine will think it has just been turned on and will boot up again. This effectively stops anyone from breaking into your program while it is running.

Another way of protecting RAM is to alter the DOS command table so that commands like SAVE, BSAVE, or CATALOG won't work. This table is located at the end of track $01, sector $07, and

runs from byte $84 to byte $FF (plus a bit on sector $08). Using Tricky Dick, you could change any DOS commands that might be used to rip off your programs. Or, you could simply write '00's' over them, rendering them completely inoperative. The chapter on The Tracer which follows this one describes the process in detail.

A better idea, however, is to modify DOS so that no direct keyboard commands of any kind can be entered. This does away with the necessity of revectoring RESET to reboot and stops people from LISTing a BASIC program. The following method applies to DOS 3.3 only.

The first step is to read in track $00, sector $0E and put the cursor on byte $ED. Change this and the next two bytes (i.e., the '59 84 A8') to '4C E5 BC' (JMP $BCE5) as shown below. Then write the sector back.

```
E8: 20 A4 A1 29 7F 4C E5 BC
```

What you've done is modify the DOS routine which takes apart all direct entries from the keyboard to see if they're DOS commands. Instead of comparing an entry with the DOS command table the routine is now set to JUMP to $BCE5 in DOS. This section of memory is completely unused under normal circumstances, but we're soon going to fix that.

Next read in track $00, sector $06 and put the Tricky Dick cursor on byte $E5 (which should be an '88'). When you boot DOS, this part of the sector gets loaded into the unused memory locations that start at $BCE5, so this is where we want to put our patch. Here you have to type in a fairly lengthy set of instructions, so be sure check your work before writing it back to the disk. Change lines $E0, $E8, and $F0 to look like the illustration below.

```
                        patch starts here
                        /
E0: A5 E8 91 A0 94 A9 08 85
E8: CE A9 00 85 CD A8 A9 91
F0: CD C8 D0 FB E6 CE D0 F7
                  \
                   patch ends here
```

The above patch is actually a loop which wipes most of the machine's memory. When control is transferred to it, the subroutine starts at $800 writing zero after zero, and doesn't stop until it overwrites part of its own code and crashes.

These two patches change things so that whenever a command is keyed in from BASIC, any program in memory (except code residing on page $03), plus a goodly chunk of DOS, will be totally eradicated. The subroutine terminates when it finally jumps to one of the '00's' it has written over itself. This is interpreted as a BReaK instruction, causing the monitor to display the usual '*', register contents, and BReaK address.

However, if the memory around that address is now inspected, what's left of the subroutine, though somewhat disguised, can still be seen and perhaps interpreted by an assembly language programmer. And if that programmer really knows his DOS, he might just figure out where the calling statement was patched in.

So to make things even more difficult, there is one final patch to go in. Read in track $00, sector $0D and change line $70 to look like this:

```
$70: 4C 03 E0 4C 65 FF 4C 58
      \  \
       bytes which have been altered
```

What you've done is changed the address that tells the 6502 where to go when it encounters a BReaK instruction. Normally, this destination is $FA59, a monitor subroutine which brings about the familiar register display and "beep". When DOS is booted, it stores this address at $3F0 in memory where it is then consulted for any BReaK handling that may be necessary during the operation of the machine. This is one of the well documented "page 3 vectors" (see page 65 of the Apple manual) that you've no doubt read about and/or used.

The new address that you've just patched into DOS tells the 6502 to jump to $E003 and start execution there whenever a BReaK occurs. However, this address is the entry point for BASIC, and merely causes the BASIC prompt to be displayed. Since the memory wipe routine terminates only when it writes '00's' into itself and one of them gets interpreted as a BReaK, control is simply returned to BASIC after the dirty work is done.

So let's say somebody tries to LIST or SAVE one of your protected programs. To his amazement, he gets absolutely no response except another ']' prompt. Because DOS has been wiped out, keyboard commands are passed straight on to the BASIC interpreter, so the user can now enter any BASIC command he likes (although DOS commands merely elicit a SYNTAX ERROR). But it's too late - your program is gone so there's nothing to LIST, and DOS is inoperative. He'll play hell figuring out what happened because the address of the memory wipe subroutine has not been displayed, nor do the sequence of the events that have occurred make any apparent sense.

Having effectively nonplussed the competition, let's see how you did when the shoe was on the other foot. The discussion below will allow you to compare your work on track $22's altered formatting with my analysis of it.

WHAT WAS ON TRACK $22

Sync Bytes - These consisted of the alternating values, FE AB, throughout all the address field sync gaps (nobody said sync bytes all have to be the same value). Toggling the sync byte values like this stops almost every bit copier if it's being

operated in automatic mode - even if the appropriate sync byte parameters are correctly specified. As I mentioned earlier, neither these nor any other altered sync bytes have any effect whatsoever on the standard Apple copy programs.

Sector $00 - Reading this sector requires that you change the Tricky Dick address field headers to DC BF AA as shown below.

```
DCBFAA Y DEAA
D5AAAD Y DEAA
D5AAAD 0 DEAAEB
```

Sector $01 - Use the same address headers as above, plus D9 AA AD in the data field header. Note that to write to this sector you would also have to change the data headers in the third line to the correct value.

```
DCBFAA Y DEAA
D9AAAD Y DEAA
D5AAAD 0 DEAAEB
```

Sector $02 - Here the data field header and trailer have switched places; the address header is the same as above. You could also have used '0000' in place of the 'D5AA' trailer.

```
DCBFAA Y DEAA
DEAAEB Y D5AA
D5AAAD Y DEAAEB
```

This switch really fouls up bit copiers, because their analysis routines often look for correct header and trailer values, and if they are found, make assumptions about the beginning of the data field accordingly.

Sector $03 - Here I pulled a bit of a fast one. In most of the current literature purporting to take you underneath Apple DOS, it is claimed that "illegal" bytes can't be used on the disk for hardware reasons. This is not strictly true, however, and if the illegal byte doesn't contain more than 3 consecutive zeros, it will often work. Witness the data field header on this sector which is F8 C7 B8 - all of whose bytes are allegedly unusable. In addition I changed the data trailer to FF FF FF, just to keep you on your toes. Your Tricky Dick parms should have looked like this (FFFF could be replaced with 0000).

```
DCBFAA Y DEAA
F8C7B8 Y FFFF
D5AAAD Y DEAAEB
```

Using illegal bytes in headers is quite handy because these bytes are "reserved" in that they will not be found elsewhere on the disk - a status normally enjoyed only by 'D5' and 'AA'. As a result, illegal values are really good for screwing up many of the current bit copiers, since these programs often expect to

find at least one occurrence of 'D5 or 'AA' in all headers. So if you're really into copy protecting your own software, you might try some of these values in your disks' data headers, and experiment around a bit to see exactly which of them work.

Another possibility is to use illegal values as data field sync bytes. The way to do this is type CALL -151 to get into the monitor. Then key in 'BC60:' followed by whatever sync byte value you want to use. Finally, INIT a diskette. You can transfer programs across to this disk in the normal way. After doing this, use some of the measures described previously to protect it further.

Well, this brings my discussion of copy protection to a close for the moment. But we are by no means finished with the subject. In the next couple of chapters, you'll be learning how to examine locked software even more closely when I'll be talking about some secrets of software protection never before documented. Now it's time for you to extend your circle of friends to The Code Breaker.

CHAPTER SIX — The Code Breaker

If you've been doing your homework up to now, your disk savvy has reached the point where you're ready to delve into one of the most esoteric aspects of DOS. In this chapter you and I are going to be looking at a method of protecting program code from prying eyes known only to a few of the most priveledged software "insiders" (until this book hits the dealers' shelves, that is). You're going to discover how, by altering only two bytes in DOS, it's possible to scramble a whole diskfull of software, yet enable it to work perfectly well once loaded into RAM.

What I'm talking about is a simple method of encrypting programs as they are written to the disk and decrypting them as they are loaded back into memory which, up to now, has effectively stopped the likes of you and me from investigating their secrets and patching them to suit our needs. This remarkable method of software disguise is accomplished by altering two crucial blocks of data in RWTS - the nibble and byte translate tables. Just in case you aren't familiar with these, I'll explain them first, and then describe The Code Breaker's instructions afterwards. If you already know all about the tables, you should jump right to the instructions a few pages hence.

UNRAVELING THE RWTS TRANSLATE TABLES

Probably the best way to get into this subject is to refresh your memory with a brief summary of what I said in Chapters 2 and 4 about the way file code gets changed just before being recorded on the disk. You probably remember that data to be stored undergo a series of transformations designed to circumvent certain hardware limitations in the Apple. You'll no doubt also recall that the first step of this process involves removing some bits (two in DOS 3.3, 3 in DOS 3.2) from the front of each byte and putting '0's' in their place. Following this, the chopped-off bits are reassembled into spareate bytes which themselves get two leading '0's' stuck on the front.

Now you might have noticed an apparent contradiction in all of this: I said in Chapter 4 that all bytes written to the disk have to have the highest bit set, yet I also said in the same breath that their high two or three bits invariably end up as zeros. The resolution to this paradox is where the translate tables come in. Their job is to transform these bytes into "legal" values which have the high bit set, and which have also met the other hardware imposed requirements for disk storage.

The Nibble Translate Table

The easiest way to end up with legal disk bytes is to set up

a list or "table" in memory which contains all the legal values, then to use the bytes we want to translate as offsets (there's that word yet again) into the table. For example, let's say that DOS 3.3 ends up with the value '1A' (00011010 in binary) to be stored on a disk. What it does is count down $1A (decimal 26) places into the table below and use the value found there. Try this out yourself.

3.3 Nibble Translate Table

```
offset:00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
table value: 96 97 9A 9B 9D 9E 9F A6 A7 AB AC AD AE AF B2 B3

offset: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
table value: B4 B5 B6 B7 B9 BA BB BC BD BE BF CB CD CE CF D3

offset: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
table value: D6 D7 D9 DA DB DC DD DE DF E5 E6 E7 E9 EA EB EC

offset: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
table value: ED EE EF F2 F3 F4 F5 F6 F7 F9 FA FB FC FD FE FF
```

If you counted alogn $1A bytes, starting with the '96' in the above table, you should have come up with the value $BF. This is the legal value that gets substituted for '1A' and subsequently written on the disk.

The table above is the "nibble translate table" and is used by DOS 3.3 in the last stage of translating data in RAM to disk nibbles. Its use is a part of the "6&2" encoding scheme discussed in chapter 4. The table starts at $BA29 in DOS, and you can see it in its natural environment by typing CALL -151, followed by BA29.BA68. You can also find it with Tricky Dick on track $00, sector $04, bytes $29 - $68 in the data display.

The Byte Translate Table

What goes on must eventually come off, so there also has to be some provision for translating disk nibbles back into data which has some meaning to your Apple. And since table look-up was the last step in writing, it seems inevitable that the same process would be the first step involved in reading back. In fact, it is, and it requires a different table which is illustrated below.

3.3 Byte Translate Table

```
offset: 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5
table value: 00 01 98 99 02 03 9C 04 05 06 A0 A1 A2 A3 A4 A5

offset: A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5
table value: 07 08 A8 A9 AA 09 0A 0B 0C 0D D0 D1 0E 0F 10 11

offset: B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5
table value: 12 13 B8 14 15 16 17 18 19 1A C0 C1 C2 C3 C4 C5
```



```
offset: C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5
table value: C6 C7 CB C9 CA 1B CC 1C 1D 1E D0 D1 D2 1F D4 D5

offset: D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 29
table value: 20 21 D8 22 23 24 25 26 27 28 E0 E1 E2 E3 E4 29

offset: E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5
table value: 2A 2B E8 2C 2D 2E 2F 30 31 32 F0 F1 33 34 35 36

offset: F6 F7 F8 F9 FA FB FC FD FE FF
table value: 37 38 F8 39 3A 3B 3C 3D 3E 3F
```

This is the DOS 3.3 "byte translate table" which starts a $BA96 and ends at $BAFF in DOS (type BA96.BAFF from the monito or look at track $00, sector $04, bytes $96 - $FF with Trick Dick). Notice that the second byte of each address in the byt table is the same as a legal disk nibble. For example, the abov table tells us that if '96' had just been read off the disk, i would be translated into '00' (which you'll find at $BA96).

DOS 3.2 Translation

The 3.2 table look-up works just in a manner quite simil to the 3.3 process mentioned above. The main point of departu is that a different pair of translate tables are used. These a shown below, and you'll notice that they are much shorter th the 3.3 tables. This is because there are fewer permissib values used to encode data for disk storage by 3.2's "5& encoding system. The nibble table shown below runs from $BAAB $BAAF in RAM (and is in sector $06 of track $00, bytes $9A $B9 on a 3.2 disk). The table shown just after it is the 3 byte table and ranges from $BC9A - $BCB9 (track $00, sector $0 bytes $AB - FF).

3.2 Nibble Translate Table

```
offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
table value: AB AD AE AF B5 B6 B7 BA BB BD BE BF D6 D7 DA DB

offset: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
table value: DD DE DF EA EB ED EE EF F5 F6 F7 FA FB FD FE FF
```

3.2 Byte Translate Table

```
offset: AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA
table value: 00 01 08 10 18 02 03 04 05 06 20 28 30 07 09 38

offset: BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA
table value: 40 0A 48 50 58 0B 0C 0D 0E 0F 11 12 13 14 15 16

offset: CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA
table value: 17 19 1A 1B 1C 1D 1E 21 22 23 24 60 68 25 26 70

offset: DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA
table value: 78 27 80 88 90 29 2A 2B 2C 2D 2E 2F 31 32 33 98
```

```
      offset: EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA
table value: A0 34 A8 E0 B8 35 36 37 39 3A C0 C8 D0 38 3C D8

      offset: FB FC FD FE FF
table value: E0 3E E8 F0 F8
```

I think you can already see that the use of translate tables in the encoding process opens up wide vistas for protecting software. Consider what would happen if you decided to initialize a disk with a few bytes switched around in these tables, and then SAVE'd or BSAVE'd a program. Only the altered RWTS on the diskette, thus protected, would be able to correctly translate the disk data back into a sensible sequence of code when the program was retrieved. A normal RWTS, on the other hand, would substitute all the standard byte values in place of the disk nibbles. However, some of these (the ones you deliberately changed) would be incorrect, resulting in meaningless values being deposited in RAM. Any attempt to execute this gartaged code would quickly result in a crash.

And that's only if somebody succeeded in LOADing or BLOADing your program in the first place. In most cases, even getting the scrambled program off the disk would be impossible for a standard RWTS since the large majority of the program sectors' data field checksums would appear invalid. These checksums would make sense only to the RWTS whose translate tables contained the appropriate alterations.

So tweaking the tables blocks the two main access routes to your software. First off, the checksum effect stops any copy program which uses a standard RWTS from backing the software up. Furthermore, even if someone figures out how to stop RWTS from calculating the data field checksum (which for DOS 3.3, by the way, merely requires typing CALL -151, followed by B92E:0), the code he ends up with will be useless. This stops software peeping Toms from using any of the presently available disk utilities (except the CIA, of course) to even look at your code.

Since installing this "secret code" on a disk is pretty simple (as you'll see in this chapter's tutorial), it is not too surprising that it is rapidly becoming widely used by commercial software houses. As you will learn from the instructions that follow, The Code Breaker can be used to alter Tricky Dick's translate tables so that a disk protected in this manner can be read and written to. And what's more, you can use The Tracer to find tables that have been moved from their standard location. Once you have found the tables you can then compare them with the standard versions we've already looked at in this chapter. Any changes can then be typed into The Code Breaker, allowing you to read, list, and write in the usual manner. In the tutorial later in this chapter, I'll also be discussing in detail both how to put this type of protection into your own software.

INSIDUCTING THE CODE BREAKER

As with the other modules previously discussed, The Code Breaker only works in conjunction with Tricky Dick, and is entered from the latter by means of the usual CTRL E. Hitting CTRL C while in The Code Breaker gets you back to Tricky Dick.

Tricky Dick Preparations: Before jumping into the Code Breaker, you need to set the slot number of your disk controller card, the drive number of the target diskette, and the DOS version.

The Code Breaker Display: When you first enter The Code Breaker, you will see its introductory screen. Hitting any key here takes you immediately to the nibble table display. This will consist of either the 3.2 or 3.3 nibble translate table, depending on the DOS version you selected while in Tricky Dick. The Code Breaker's cursor will be resting on the table's first byte. To the left of each byte in the table is shown the value that it replaces.

The Help Screen (? or ?): Just like the previous modules, a '/' or '?' takes you to the help screen. This displays each of the commands and gives a brief summary of them. It can be summoned up at any time while you are using The Code Breaker.

Editing the Translate Tables: Once you have the nibble table on the screen, you can move the cursor from byte to byte with the 'I', 'J', 'K', and 'M' keys, just as in the other utilities. CTRL I, J, K, and M enable you to make big jumps around the display in the usual manner. CTRL B takes you to the first entry in the table, and CTRL N shoots you down to the last byte.

Changing byte values in the translate tables is carried by positioning the cursor over the byte in question and typing the new hex value over it. Remember to type a leading zero in front of any single-digit hex number (e.g., type '03' if you want to enter a '3').

Saving the Changed Table (CTRL S): When you have finished altering the translate table to match the one you found on the target disk, hit CTRL S to incorporate this table into Tricky Dick's RWTS. The CTRL S command puts your changes both in the nibble table and in the byte table, enabling Tricky Dick to read from and write to the encrypted disk. If you accidently duplicate a value or type in an "illegal" value, it will flash when you hit CTRL S. This is your cue to use the instruction described in the next paragraph.

Erasing a Mistake (CTRL R): If you mistype a value when editing the table, hit CTRL R to restore it to its original state. This causes all the values you typed in be scrubbed and displays the previous values.

Restoring the Standard DOS Table (CTRL D): If after altering a translate table (either before or after consolidating the change with CTRL S) you wish to return the standard DOS version to Tricky Dick's RWTS, simply hit CTRL D. The standard table will be displayed by the Code Breaker and incorporated by Tricky Dick. Always remember to do this if you have finished working with an encrypted disk and now wish to use Tricky Dick on normally encoded software.

THE CODE BREAKER TUTORIAL

In this tutorial I'm going to adopt a change of pace. Instead of asking you to work with something on the CIA disk, I'm going to show you how to make up your own practice materials. This approach will provide you with a double-edged sword. First, you'll learn how to use the translate tables to protect your own software by writing it on the disk in a specially scrambled form. Once you've done this, you'll then have an ideal practice disk for learning to use The Code Breaker to decipher software already encrypted in the same manner.

Encrypting Your Programs

As I've already pointed out, a simple patch to the translate tables scrambles any programs written to the disk, and usually also alters the data field checksums so as to make the disk uncopyable by any utility using a normal DOS. Only the DOS on the disk thus protected will be able to make sense of the programs recorded upon it. Add a few more protection measures from chapter 5 and you've got a pretty formidable disk lock.

Start by booting in a normal 3.3 DOS and placing your System Master diskette in the drive. Now type in LOAD BRIAN'S THEME. List the program just to make sure it's there, and type CALL -151, followed by BA29L. This will display the beginning of the nibble translate table used in writing program code to the disk. You can verify this by comparing the first few bytes ('96 97 9A', etc.) with the 3.3 nibble table I reproduced a few pages back. About a quarter of the way down the screen you will see the following line.

```
BA32-   AB          ???
```

When DOS is ready to store the value $09, it first substitutes $AB in place of it. You can confirm this by looking it up in the 3.3 nibble shown earlier, or counting up from the '96' at the top of the screen. Let's change this byte to 'AA' by typing in BA32:AA. After keying in BA29L once again, the line should you changed should look like this.

```
BA32-   AA          TAX
```

What you have done so far is modify the nibble table so that whenever RWTS encounters the value $09 during a write cycle, it will substitute 'AA' in its place instead of the standard value, $AB. Though $AA is a "reserved" value to be used only for headers and trailers, we can still get away with using it here (because we haven't touched 'D5' which is also used exclusively for headers and trailers, and DOS really only needs one such reserved value).

Whenever DOS saves your program code on the disk, it always reads it back as a final check. And, of course, you'd like to be able to run your program later on. This means we also have to alter the byte translate table which is used during the read cycle. So type BAA0L and have a look at the following two lines about a third of the way down the screen.

```
BAA9-   A9 AA       LDA     $AA
BAAB-   09 0A       ORA     $0A
```

Whenever RWTS reads an 'AB' off the disk, it comes to location $BAAB and retrieves the hex number at that location (the rule is: add the byte value to $BA00, i.e, $BA00 + $AB = $BAAB). Thus, $AB gets translated right back into the original value, $09. However, we've just set things up to translate $09 into $AA, so we've got to make sure the reverse process also works. This means we need to put the '09' in location $BAAA so that $AA will get interpreted as $09. Do this by typing BAAA: 9 AA. Then type BAA0L again and the same 4 bytes should look like this.

```
BAA9-   A9 09       LDA     $09
BAAB-   AA          TAX
BAAC-   0A          [illegible]
```

This is all the table doctoring you need to do, so place a blank diskette in your drive and type INIT HELLO. This of course makes BRIAN'S THEME the "hello" program on the newly initialized disk.

BRIAN'S THEME and all of DOS is now encrypted so that the program code is scrambled and most of the checksums appear faulty to a normal DOS. In fact, we've actually protected the disk too well - so well, in fact, that the disk controller card can't recognize the DOS image, and hence the disk is unbootable.

You see, whenever you boot up, your Apple starts the boot process off by running a small machine language program in ROM on the controller card. Since this program's job is to read in some data from the disk, it has to make use of a byte table. However, none exists in memory since the translate tables are contained in RWTS which hasn't been booted in yet. The controller card resolves this Catch-22 situation by generating a temporary table for its own use.

The program on the card loads in sector $00 on track $00, and looks at its very first byte. This byte specifies the total

number of sectors that are to be read during this stage of the boot process. In a normal DOS 3.3 (but not always in protected DOS') this byte is always $01, conveying the message that only this first sector is to be loaded into RAM. When this information is digested, the program in the card's ROM passes program control over to sector $00's code.

The disk controller's ROM code has more than a walk-on part, however. Sector $00 contains another short batch of machine language which now calls that ROM code as a subroutine in order to read in the next 9 consecutive sectors. This means that the temporary byte table generated by the controller card must be used to read in sectors $00 - $09 of track $00 (which, by the way, contain the RWTS image with the permanent translate tables). Unfortunately, the temporary table contains the standard values, and hence won't read your protected disk's DOS because you initialized the entire disk (including its DOS) with a diddled byte table.

The solution is elementary, though slightly tedious. What you next have to do is copy the first 9 sectors from any normal 3.3 disk onto the protected disk. That way, the checksums will be recognizable to the controller card when it uses its standard byte table to read these sectors in.

Start by putting the CIA in drive one and your newly initialized disk in drive two. Now boot up the CIA disk and BRUN Tricky Dick. Read in track $00, sector $00 from the CIA and change the drive number to 'DR=2'. Then write the sector to your protected disk in drive two and increment the sector number by one with the right arrow. Change the drive number back to '1' and read in track $00, sector $01 from the CIA disk. Write it to your protected disk, and continue the process until sectors $00 - $03 have been transferred. If you only have one drive, you'll have to swap the two disks back and forth for each sector you copy with Tricky Dick. This will call for some extra slight-of-hand, but will obviate the necessity of constantly changing the drive number back and forth.

Now read in sector $04 from the CIA disk and change byte $32 from an 'AB' to an 'AA' as shown below.

```
30: A6 A7 AB AC AD AE AF B2
            \
        change to an 'AA'
```

Then move down to line $A8 and switch the positions of bytes $AA and $AB (their values are $09 and $A9 in a normal table). When you've done this the line should look like:

```
A8: AB A9 09 AA 0A 0B 0C 0D
      \  \
      these bytes have been switched
```


These changes will no doubt appear familiar, since you just used them to protect the target disk. If you transfer a normal $04 across without changing these values, the RWTS on the protected disk will just end up with normal nibble and byte tables, and hence not be able to make sense of the encrypted data on that disk. On the other hand, if you don't make the transfer, the checksum in the target disk's sector $04 will be unrecognizable to the controller card, and hence any attempt to boot the protected disk will fail. The only way to resolve this dilemma is to install the same changes in the tables on sector $04 that you originally used to protect the disk.

Finish off by writing the sector to the protected diskette. Now advance the sector number to $05, switch back to drive one, and copy it across to drive two. Do the same with sectors $06 - $09. When this is completed, the disk will be ready to go. You'll find that it boots normally and runs BRIAN'S THEME as the hello program (if not, repeat the protection process, being extra careful when copying sectors $00 - $09 across). However, if you boot up a normal DOS and try to RUN HELLO on the protected disk, you'll just get an I/O error due to the checksum effect. You'll also get the same result if you try to copy the protected disk with one of the standard Apple copy programs.

Just to see what the encrypted version of BRIANS THEME looks like to a normal DOS, reboot your CIA disk. Then type CALL -151, followed by B92E:0. This cancels all data field checksum errors and enables you to LOAD HELLO from the disk you've just encrypted. Do this, and if the machine hangs, hit RESET. Now type LIST and have a look at the program. The first lines will appear more or less normal, but starting at line 200, the program code really hits the fan. Everything after that is a real mess, and of course is incomprehensible to any would-be snoopers. Running this code will have unpredictable results.

The corrupted BRIAN'S THEME is pretty representative of what files look like which have been encrypted by changing the translate tables. You'll usually see a large batch of meaningless data, followed by a few lines of sensible instructions. This in turn merges into more garbage, and so on.

But how does changing only one value in each of the two tables have such a disasterous effect? Don't forget that there are only 1/4 as many DOS 3.3 values available for disk storage as there are for RAM storage (64 as opposed to 256). So to start with, $AB (or any other disk nibble you choose to mistranslate) is more likely to be encountered on the disk than you might think, and a fair number of bogus translations of it should thus be found when a file is read in.

Moreover, certain details of the translation process (too trivial and tedious to go into here) insure that once a cha... value in the byte table is used to translate a disk nibble, the rol... spreads to a substantial number of the nibbles near it, regardless of their value. Thus, a simple change to the tables

up front causes a striking amount of scrambling later on.

Of course if you want to increase the probability of all your program being undecipherable, you only have to change some more bytes in the tables. The best way to do this is to switch some of the values around. For example, try swapping the nibble table values at $BA48 (normally a 'D3') and $BA53 (an 'E6') by typing CALL -151, followed by BA48:E6 ↵ BA53:D3. Make the corresponding alterations to the byte table with $BAD3:2A ↵ BAE6:1F (these addresses are obtained by adding $D3 and $E6 onto $BA00). Then put in the change from $AB to $AA described above. Finally, initialize a disk and transfer the first 9 sectors across the way you did before. Use a different diskette, however, because you'll need the original $AA disk for the next section. Remember to incorporate all the changes in the sector $04 tables using the same method you've already learned. Now get the program into memory (by cancelling the checksum) and have a look. Ibal should stop 'em from ripping off your software!

How to Edit Encrypted Software

Now that you've discovered how to protect software by patching the translate tables, you're more than ready to learn how to unprotect it using the same method. The disk you created above with the $AA substitution for $AB will make an ideal medium for you to practice on. Since The Code Breaker does automatically some of the operations you've just done by hand, you should find this half of the tutorial a piece of cake.

Start by BLOADing The Code Breaker and BRUNing Tricky Dick. Then put your newly protected disk into your drive and make sure the drive number and DOS parameters in Tricky Dick's display are set to read it. Look at track $11, sector $0F to find the location of HELLO (in reality, BRIAN'S THEME). Its track and sector list should be on sector $0F of track 12, and the program code should begin on sector $0E of the same track. You'll find that if you try to read in the first few sectors of the program, Tricky Dick will fail on about half of them. This is because your translate table patches have caused the RWTS on the protected disk to generate data field checksums which appear incorrect to a normal RWTS.

Now hit CTRL S and change the data field checksum designator from 'Y' to 'N' as shown in chapter two. This should allow you to read all the sectors in BRIAN'S THEME. However, you'll now find that those which caused a Tricky Dick I/O error before you cancelled the checksum look like garbage when you finally do succeed in getting them into Tricky Dick's display. To verify this, do an Applesoft listing on them using the sequence I explained in Chapter two. If you ever encounter this same situation with another disks, you should immediately suspect the presence of some type of protection involving the translate tables.

To test this hypothesis, read in sector $04 on track $00 and find the beginning of the byte table. As mentioned previously, it should start on byte $96 of row $90. This byte should be a '00' as shown below.

```
$90: 82 C5 C4 B3 B0 88 00 01
                        \
                  first value in the byte table
```

A quick inspection of the standard DOS 3.3 byte table shown earlier will reveal that the positions of 'AA' and '09' have been reversed. Of course, you already know about this little subterfuge because you yourself were the agent of its creation. Nevertheless, pretend that it's new to you and continue. In order to illustrate how to proceed from here, I'll reproduce the appropriate line of 3.3 the byte table.

```
offset: A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5
table value: 07 08 A8 A9 AA 09 0A 0B 0C 0D B0 B1 0E 0F 10 11
                         \  \
                   bytes which you reversed
```

In order to read the disk, we've got to tell The Code Breaker to edit Tricky Dick's tables to take account of the altered table values.

To do this, use the 'I', 'J', 'K', and 'L' keys (with or without pressing the CTRL key at the same time to hurry things up) to move The Code Breaker's cursor over the 'AB' in the display, then type in 'AA'. This value is in the first column of the display, part of which is shown below.

```
.
.
07- A6
08- A7
09- AB ===> change to AA
0A- AC
0B- AD
.
.
```

Now to hit CTRL S to "save" the current table, i.e., patch it into Tricky Dick. Just for some extra practice, hit CTRL N, followed by CTRL B. This will make the cursor jump to the last and first entries in the table respectively - handy if you're ever in a hurry to get from one end to the other.

While you're on the first byte, a '96', type in 'AB'. Keep your eye on this value and hit CTRL R. Since this command restores the table to its previous form, the 'AB' changes right back to a '96'. However, notice that the 'AA' you previously entered remains unchanged. This is because you saved it with CTRL S. If you wanted to go right back to the standard DOS 3.3 nibble table (which you don't at this point), you would type in a

CTRL D and the 'AA' would be replaced with the standard value for that position, an 'AB'. If while on the '96', you had duplicated one of the bytes in the table, or keyed in an illegal value, it would have begun to flash. In either case, CTRL R would have put things back the way they were before you made the offending keystrokes.

Having successfully edited the translate tables, you can now read all of the sectors in BRIAN'S THEME, alias HELLO, on the protected disk. Hit CTRL C to get back to Tricky Dick, and check this out. If you get an I/O error, go back to The Code Breaker, hit CTRL D, and try again.

The D5 = D6 Switch

This is the most common alteration to the tables currently in use. What the person protecting the disk does is substitute 'D5' (a value normally reserved for headers) in place of 'D6' (a legal value). This leaves 'D6' out of the tables entirely, and hence turns it into a reserved value as far as that paticular DOS is concerned. 'D6' is then substituted for 'D5' in the address and data headers (after all, it has just become a reserved byte). So if you see headers beginning with this byte, change Tricky Dick's DOS mark display accordingly, and also cancel the data field checksum (as explained in Chapter 5). Then read in a few sectors from here and there and list their contents. If you find garbage alternating with sensible code, you should suspect that the D5 - D6 switch is in effect.

The next step is to use The Code Breaker to edit the translate table, putting a 'D6' in place of the 'D5'. Now read in a few sectors and list them. The odds are pretty good you'll be make sense of their contents.

Remember finally that disks whose tables have been altered may have also been subjected to other protection methods. Be sure to check for this when working with such a disk.

Well folks, that's all there is to The Code Breaker. You are now privy to one of the latest software protection secrets (at the time of writing). By now you should also be pretty handy with the CIA utilities, and know your way around the disk quite well. All of which means you are ready for your next encounter with a new member of the CIA - so turn the page and shake hands with The Tracer.

CHAPTER SEVEN — The Tracer

If you've ever wanted to find something on a disk, The Tracer is for you. This member of the CIA team will search each sector in the range you specify, sniffing out the catalog, the VTOC, all track and sector lists, and any 1 - 6 strings you elect. You can choose any or all of these options and The Tracer will turn them up in record time, jumping into Tricky Dick to point out each found item on the data display.

The Tracer's services allow you to carry out an almost unlimited variety of disk tasks quickly and easily. For example, I'm going to show you in this chapter how to find and rescue lost files in the most rapid manner physically possible on the Apple. You'll also learn how to turn your machine into a text retrieval "search engine" to track down word processor and document files on the basis of key words and phrases. Or would you like to search for one or a series of strings specified in high ASCII, low ASCII, or hex - and do this where either a normal or abnormal disk format is present? And on the subject of funny formatting, perhaps you'd find it useful to locate track and sector lists, the VTOC, and the catalog on protected disks. These and many other disk miracles are possible once you've soaked up this chapter, so let's get started right away by learning how to communicate with The Tracer.

THE TRACER'S INSTRUCTIONS

In the manner to which you are no doubt accustomed by now, I'm going to start out with a brief discussion of the Tracer's instructions, and then follow up with a more in-depth tutorial. As usual, the best way to use this chapter is to read this first section carefully, trying out the various commands, then proceed afterwards to the tutorial.

Tricky Dick and The Tracer

Just like the other two modules, you always have to use The Tracer in conjunction with Tricky Dick. To get The Tracer into memory, type BLOAD THE TRACER and start up old Tricky. Just as with the other modules, a CTRL E now gets you directly into The Tracer and displays its introductory screen. From this screen you can either press CTRL C to get back to Tricky Dick, or hit any other key to obtain the menu.

Getting Ready for The Tracer

Before jumping from Tricky Dick into The Tracer, you will need to set the slot and drive numbers (CTRL O) and the DOS version (CTRL D) of the disk you wish to search. The 'T=' and 'S=' parameters are adjusted automatically by The Tracer, and the DOS marks display can be set from The Tracer's menu (as described

later). When the menu appears, Tricky Dick's parameters will still remain in view, but it's author's name (T. TSE) is blanked from the screen. He won't mind though, since he didn't write The Tracer, and I needed the room to display some messages.

The Menu

The best way to make use of this section is to get The Tracer's menu up on the screen and refer to it as I describe its use. The top part of the menu lists the 5 search options available. These are:

VERIFY FORMATTING

FIND: T&S LISTS CATALOG SECTORS VTOC

STRINGS:

You make your selections by pressing 'Y' for "yes" or 'N' for "no" when the cursor appears on the first letter of each option. As selections are made, the cursor jumps from one option to the next. The Tracer immediately inverses each option you select, thus highlighting them for later reference as you proceed through the menu.

The numbers 1 - 6 directly beneath "STRINGS" are where you type in the hex and/or ASCII strings you want to look for, and the RANGE section allows you to specify the high and low points of the range you wish to search. Now let's start at the top of the menu and work down through it in detail.

Verify Formatting

This is where the cursor comes when the menu first appears. A 'Y' here tells The Tracer to read in sectors on the disk, but not to analyse them in any way. The Tracer will do this using the DOS version and sector marks shown in the Tricky Dick display. This allows you to check very rapidly whether a range of sectors (or the entire disk, if you like) is readable using a particular set of values in these parameters. Since the following 4 options do involve analysis of the sector data, if you type a 'Y' in response to "verify", the cursor will skip over them and jump straight down to bottom of the menu, prompting you to enter the range of sectors you wish to check out. If instead you ty[illegible] the cursor will proceed to ....

T&S Lists

A 'Y' here tells The Tracer to look for and display all Track and Sector Lists within the search range you are going to enter later in the menu. Starting here, either a 'Y' or an 'N' reply to each option causes the cursor to move directly on to the next.

Catalog Sectors

As the name implies, selecting this option tells The Tracer to search for catalog sectors within a given range and show them to you. Of course, with a normal disk you already know where the catalog is, but this particular service could prove rather useful when working with protected software where the catalog is likely to be somewhere else on the disk.

VTOC

This enables you to search for the Volume Table of Contents, which under a normal DOS is always found on track $11, sector $00, but which could turn up anywhere on a protected disk. If you need any brushing up on the function of the VTOC, have another quick look at my discussion of it in Chapter 2.

IMPORTANT NOTE: Though this never happened once in my benchmark trials with several dozen disks of personal and commercial software, it's just possible that a sector which is not a T&S list, catalog sector, or the VTOC could look like one of these to The Tracer. Similarly, having written the program myself, I can think of a few ways of "protecting" these 3 types of sectors by changing them so that they would be unrecognizable to The Tracer, yet would not screw up any DOS operations. Similarly, The Tracer may pass over one of them if the sector has been badly clobbered. So expect about 99 and 44/100% accuracy, but don't hope for absolute infallibility.

ANOTHER IMPORTANT NOTE: Protected disks may not have a VTOC, catalog, or T&S lists. Their programs may be loaded by direct disk access, thus obviating the need for these 3 items.

Strings

If you answer 'Y' to this option, you will be able to specify from 1 to 6 strings either in hex, high ASCII, or low ASCII, and The Tracer will look for them in each sector of the range you designate. Your 'Y' reply causes the word "STRINGS" to light up and the cursor to jump just below it, next to the "1-". This prompts you to tell The Tracer which of the 3 types of strings you want to enter, so you'll need to select one of the following:

String Type	String Designator
Hexadecimal	type a '$' sign
High ASCII	type an 'H'
Low ASCII	type an 'L'

The character you type will be echoed in inverse on the left of the cursor. This is your signal to start keying in the string, whereupon the inversed string designator will jump over to the left of the "1-" to remain there while you fill out the rest of the menu.

If you are entering a hex string, spaces between hex bytes won't be accepted. This allows maximum room on the menu for your string. Hex strings can contain from 1 to 15 bytes. Be sure to put the leading zeros on all hex bytes you enter. For example, '3' should be entered as 03.

If you are entering an ASCII string, spaces will be accepted and will become part of the string. You can also enter control characters in an ASCII search string and these will be displayed in inverse on the menu. If you want to designate a CTRL M, type in a CTRL SHIFT M (ordinary CTRL M is the same as a carriage return - which tells The Tracer that you have finished typing the present string). High or low ASCII strings can consist of 1 - 20 alphanumeric characters.

When you enter the 15th byte of a hex string or the 20th character of an ASCII string, The Tracer will sound a tone and will refuse to take any more input.

You can indicate wildcard bytes or characters in any string by using the '=' sign. However, to avoid certain search problems, a wildcard will not be accepted as the first character of a string. So if you want to search for, say, =ILLER, just enter ILLER, and you will get the same results. If you use wildcard in a hex string, you'll find that The Tracer will print a double '=' to indicate that the wildcard status applies to both digits of the byte.

When you have finished typing in string 1, simply hit RETURN (or CTRL M) and the cursor will move to the "2-", prompting you to type in a second string. If you're entering a hex string, you'll notice that The Tracer won't allow you to press RETURN unless you have specified both digits of a byte. If after completing the first string, you don't want to search for any more, hit RETURN again and the cursor will jump down to the "RANGE" section of the menu. On the other hand, if you do want to specify more strings, simply repeat the same entering procedure for each one.

Specifying the Range

When the cursor first comes here (after you have specified all the search arguments you are intested in) it will land next to "LOW TRACK-", prompting you to enter the lowest numbered track in the range you wish to search. "LOW SECTOR" is the cursor's next stop and works exactly the same way for the lowest sector. The high track and sector come last and refer, as their names imply, to the highest numbered track and sector in your search range.

Track and sector numbers MUST ALWAYS BE ENTERED IN HEX and will appear on the menu as 2-digit bytes. However, if you want to enter a single digit number without typing the leading zero, just key in the digit followed by RETURN. For example, if you wanted to enter '3' as the low sector, hitting the '3' key plus

RETURN would leave the entry looking like this: LOW SECTOR-03.

Just as with Tricky Dick, there is no restriction on the track or sector numbers you are allowed to enter. So, for example, you could make The Tracer attempt to start its search at track $FF, sector $FF if you wish. However, since no such address is likely to exist on any disk, you will simply get a tortured clicking and grinding sound from your drive as DOS vainly attempts to push the disk arm 255 tracks inwards towards the center of the disk. This will will in no way harm your drive, but it will bring the proceedings to an abrupt halt and elicit an error message from The Tracer. The reason for making it possible to enter any track or sector numbers to the standard values is that the latter may be altered to nonstandard values on a protected disk you wish to search, and the former may increase on future disk drives to a quantity well beyond the present $23 (decimal 36) now available on standard Apple drives.

The only time this is likely to cause you problems is if you are searching a 3.2 disk and, having become accustomed to 3.3 sectoring, accidently specify a sector larger than $0C (decimal 13) in the search range.

Menu Verification

When you complete the SEARCH RANGE section, the words "ALL OK?" will appear at the left edge of the menu, giving you a chance to check that all the information you have just entered is correct. If it isn't, type an 'N' here and the menu will be restarted, erasing all your entries in the process. Hitting a 'Y' at this point begins the search.

Editing the Menu

If before pressing RETURN, you discover that you have made a typing error when entering a string type designator ('H', 'L', or '$') or a string itself, key in CTRL @ (CTRL SHIFT P). This erases the entire line and returns the cursor to the starting point, thus permitting you to retype your string. CTRL @ also wipes the string designator, so you will have to re-enter this also. CTRL @ also erases the present entry in the "RANGE" section. Hitting CTRL Z a few times steps back through the RANGE entries, erasing as it goes. This allows you to replace these entries without restarting the MENU. It can even be used when you get the "ALL OK" prompt.

Jumping out of the Menu

At almost any time when you are filling in the menu, hitting CTRL C will return you to Tricky Dick, and ESCAPE will restart the menu, both instructions cancelling everything you have entered. The only time you can't use these two commands is after you have typed the first byte of a string, or the first digit of a track or sector number. At that point in the proceedings, you will have to hit CTRL @ first before issuing the command.

How The Tracer Traces

When The Tracer scans a disk, the menu is replaced with an inverse "SEARCHING" message, and the tracks and sectors being examined are clocked up next to "T=" and "S=" respectively. If you watch this latter display, you will notice that The Tracer ALWAYS STARTS ITS SEARCH AT THE HIGHEST TRACK AND SECTOR IN THE RANGE AND WORKS DOWNWARDS TO THE LOWEST TRACK AND SECTOR. This strategy has been adopted because DOS always writes files on the disk in this manner. By searching through the sectors in the same backwards order, The Tracer gains substantial speed advantages and greatly increases the likelihood of detecting sector overlaps (soon to be discussed).

How The Tracer Flags What It Has Found

During the course of a search, The Tracer reads one sector at a time into Tricky Dick's buffer (at $2E00 - $2EFF) and checks it to see either if it's one of the sector types you've selected or contains a string you're looking for. When The Tracer finds one of the search arguments you've specified on the menu, it immediately jumps back into Tricky Dick and uses Tricky's data display with ASCII to exhibit the contents of the lucky sector. The hex numbers which then appear after "T=" and "S=" indicate the track and sector address where the find was made.

If The Tracer has found a specified String, it places Tricky Dick's cursor on the first byte of that string and, where possible, the data display is adjusted so that the row in which the string appears at the top of the data display. In addition, the word "STRING", in inverse, appears in the upper left corner of the screen (where "BY T TSE" used to be). This is followed by the number on the menu after which you typed the string.

Whenever The Tracer finds a T&S list, a catalog sector, or the VTOC, it displays the first half of the sector in Tricky Dick's data plus ASCII display, and places the cursor on byte $00. The Tracer will also show, in inverse and at the [illegible] corner of the screen, exactly which of these 3 sector types it has found.

Note that when The Tracer jumps to Tricky Dick's display to flag something it has located, it passes complete control to Tricky Dick. This means that you are actually "in" T.D. and can use all of its commands and functions. So, for example, if The Tracer has just found a search string and jumped to Tricky Dick to place the cursor on the first character, you can easily change the string and then write the altered sector back to the disk in the usual manner. Or, you might want to hit CTRL 'L' to disassemble all the code from the flagged string to the end of the sector. I guess you can see that using the Tracer in tandem with Tricky Dick opens up a wealth of possibilities for disk diddling, limited only by your requirements and experience with the two programs.

Continuing the Search

Once The Tracer has spotted a string or sector type you are interested in, and has returned to Tricky Dick to display it, the hunt is by no means over and the The Tracer is still "live". This means that if you now press CTRL E, The Tracer will not flash up its introductory screen as it normally would, but will continue to analyse the data presently in the buffer if your selections so require. After that, it will look through all the remaining sectors in the search range for everything you selected in the menu.

This means that, unless you specify otherwise, The Tracer will search through the rest of the sector data presently in Tricky Dick's buffer for further occurences of any string it has just found, and will repeat this process for every other string you typed in the menu. It does this in numerical order, so every occurrence of string 1 on the menu is flagged first, then all examples of string 2 are shown, and so on. Once a string has been located in a given sector, the search for other examples of it or other selected strings is instantaneous.

However, once The Tracer has found a T&S list, catalog sector, or VTOC, it will not analyse that sector further when you hit CTRL E, but will pass on to the next one in the search range (this doesn't apply unless you specifically selected one or more of these 3 search options). In such a case, further analysis of the same sector data is almost always unnecessary, and skipping it speeds up the search. For example, if a particular sector turns out to be a VTOC, then it can't possibly also be a catalog sector or a T&S list, nor is it likely to contain any string that the user is searching for. If you want to be absolutely certain, however, have a quick look for any of your search strings in these 3 types of sectors if The Tracer turns one up.

Handling Sector Overlap

By "sector overlap", I mean that part of a search string is found in the last couple of bytes of one sector, and the rest of it is located in the first few bytes of the very next sector. The bad news about this particular situation is that there is no way to handle it completely satisfactorily. The good news is that it is pretty unlikely to occur. For example, the odds against a 5-byte string occurring across a disk data split between two sectors is 4 in 256, or 63 to 1 against.

Nonetheless, The Tracer has an algorithm for spotting and flagging overlapping strings. What it does is display the sector which contains the last half of the split string. It also places Tricky Dick's cursor on the last byte of the string, and, just to be sure you know what's going on, displays a flashing '0' (for "overlap") in front of the "STRING" message in the upper right corner of the screen. This means you will have to look at the previous (i.e. consecutively higher numbered) sector to see and/or change the first part of the string. To do this, press

the right arrow once, followed by CTRL R from Tricky Dick. Then move to line $F8 to see the string's first bytes.

Unfortunately, however, there doesn't seem to be any practical way of designing a search utility both for standard and nonstandard DOS which is certain to pick up every single occurrence of an overlapping string. This is because there are many possible sector orders other than the standard DOS order utilized by The Tracer. For example, the sector order may have been rearranged to speed up disk access, and hence be different from the sequence in which The Tracer searches the disk (which is based on a normal DOS skew). Often too, files are not written in consecutive sectors, but are scattered around the disk. Finally, and most importantly, don't forget that the image of DOS itself is written in a sector order which is the reverse of that of files.

Altering the Search Range from Tricky Dick

If The Tracer has just jumped Tricky Dick and is still live (i.e., hasn't reached the end of its current search range), you can use "<", ">", and the arrows to select a new starting address from which to search. If you make use of this facility, The Tracer will always continue searching downwards from any track and sector shown at "T=" and "S=".

When the Search is Completed

When The Tracer has searched throuh the range you nominated, it will display an "END OF SEARCH RANGE" message and a menu which allows you to either return to the main menu or go back to Tricky Dick. Hit a '1' or a '2' to specify your selection.

Aborting a Search

At any time while the disk is spinning and The Tracer is looking at sectors, you can press CTRL A to stop the search. The track and sector where The Tracer stopped will be shown after "T=" and "S=", and a "SEARCH ABORTED" message will appear. You'll also be given the choice of proceeding either to the menu or to Tricky Dick.

You can even abort a search while in Tricky Dick with a CTRL A. This is used when The Tracer has passed control to Tricky Dick to display a find, and is thus still live. CTRL A here "unhooks" The Tracer so that if you now hit CTRL E, rather than continuing the search, The Tracer flashes up its introductory screen, allowing you to start anew. In addition the message stating what The Tracer found is replaced with "BY T TSE" in the upper right corner of the screen. This latter operation is useful if you intend to work with Tricky Dick for a while, since an out-of-date Tracer message on the screen could prove somewhat distracting.

Tracer Error Handling

If for any reason The Tracer is unable to read a sector in its search range, everything will come grinding (literally!) to a halt and you will get a flashing "SECTOR UNREADABLE" message. The address of the offending sector will appear after "T=" and "S=", and a 4-option menu will be presented to you. As usual, you can decide either to go to Tricky Dick or return to The Tracer's main Menu. In addition, you have the choice of continuing the search starting with either the next sector or the next track. The NEXT TRACK option decrements the track number shown and takes up the search from sector $0F or $0C, depending on the DOS verions of the target disk.

This may happen frequently with DOS 3.2 disks, since empty sectors are not formatted with data field marks, and hence cannot be read. When it does, remember to try the NEXT SECTOR option a few times before opting out with NEXT TRACK.

If you want to try using some different DOS marks to search the unreadable sector, you can modify the ones shown in the Tricky Dick display and The Tracer will automatically attempt to read the sector again. This process is described under "Changing the Sector Marks" below.

Defaulting to the Previous Search Arguments

If you decide to return directly to The Tracer's main menu from either a "SEARCH ABORTED", an "END OF RANGE", or a "SECTOR UNREADABLE" menu, you can elect to preserve all the search arguments (i.e., the strings and sector types) you last specified. This enables you to avoid typing your menu selections over again if you want to search another range for the same arguments. To accomplish this, press CTRL D immediately upon returning to the main Tracer menu, and the cursor will jump straight down to the RANGE section. Now you just specify the high and low addresses of the new search range and The Tracer will look through this for exactly the same strings and/or sector options that it was previously hunting for.

Remember, this command is available only if you've just returned to the main menu from one of the above 3 submenus and the cursor is flashing over the 'V' in "VERIFY". Moreover, you must decide to default before entering anything else. If you start working through the menu, then use ESCAPE to get back to the start, the default option will be cancelled.

Changing the Sector Marks

Any time you are at the beginning of the menu (i.e., when the cursor is sitting on "VERIFY"), you can type CTRL S to change the DOS marks. This process works just as in Tricky Dick and is described in the two chapters which deal with this utility. You can also change these marks any time you get a "SECTOR UNREADABLE" message. Instead of selecting one of the

accompanying menu's 8 choices (described above under "Tracer Error Handling"), simply press CTRL S. The cursor will jump up to the Tricky Dick sector marks, enabling you to alter them in the usual manner. When you have made your changes, press RETURN, and The Tracer will automatically attempt once again to search the unreadable sector. If it succeeds in reading this sector, it will continue the search from there using the new sector marks you just specified.

The above two options come in particularly handy when, having searched one group of sectors on a protected disk, you have to nominate different parameters to get through another batch which have a different set of marks (DOS marks may change from sector to sector on protected software).

Tracer Scan Times

Whenever I've described The Tracer's search capabilities to my computer buddies, their first reaction has almost always gone something like "wow, it must take a long time to do all that searching". This led me to carry out some benchmark tests which, in fact, demonstrated that The Tracer is just like Speedy Gonzales - fast, fast, fast. The scan time for a 3.2 disk is always 68 seconds, irrespective of which options or how many you choose. Due to sector skew differences between the two DOS', however, the picture with 3.3 disks is a bit more complicated. Here the scan times vary from 19 - 135 seconds, depending on which options are in effect and what's on the disk.

THE TRACER TUTORIAL

Let's start by checking out your CIA disk to make sure it's readable by a normal RWTS. BLOAD The Tracer and then BRUN Tricky Dick. Use CTRL D and CTRL O to set the slot, drive, and DOS type parameters. Then hit a CTRL E to get The Tracer's introductory screen up and press any key to display the main menu. The cursor should be on the first option, VERIFY FORMATTING, so answer 'Y' to this. When the option lights up and the cursor jumps down to the RANGE section, type the following sequence: 0 RETURN 0 RETURN 21 RETURN F RETURN. This establishes that The Tracer is to read in each sector in descending order, starting with track $1F sector $0F, and working down to track $00, sector $00. Since the disk has some funny formatting from track $20 onward (for you to practice on) we'll skip these tracks for now and pick them up later.

Type a 'Y' in reply to the "ALL OK?" message, and sit back while The Tracer checks out your disk. You'll be able to follow The Tracer's progress by watching "T=" and "S=", and noting the track and sector numbers flying by. As I mentioned earlier, the disk gets scanned backwards, starting with the highest track and sector in the search range and ending with the lowest. So if everything is working well, you should observe that the track and sector numbers decrease consecutively as the search proceeds. When The Tracer has finished, it will display the "END OF SEARCH RANGE" menu (If you got an error message before The Tracer reached track $00, sector $00, you'd better make another copy of the CIA disk). This indicates that every track from $1F downwards is RWTS readable using the DOS marks shown in the upper left of the screen. Hit a 'I' to get back to the main menu.

Now type an 'N' and the cursor will jump down to T&S LISTS. For the moment, let's skip this option by typing another 'N'. Reply 'N' as well to CATALOG SECTORS and VTOC. When you get to STRINGS, hit a 'Y'. This option will light up and bring the cursor down in position to type in string one. We're going to be entering several strings in this part of the menu, so if you make a typing error in any string and notice it before hitting RETURN, remember that CTRL P (CTRL SHIFT P) erases the erroneous entry and allows you to retype the string. If you spot an error after you've already hit RETURN, use the ESCAPE key to restart the menu and begin anew.

Type a '$' sign to tell The Tracer that string one will be in hexadecimal. The inverse '$', which replaces the '-', is your cue to start entering some hex bytes, so type in 2058FC and hit RETURN. Keying in the first character makes the '$' jump over beside the '1-' for your reference throughout the menu. You'll also notice that The Tracer doesn't allow you to insert spaces between the individual bytes, which means that you'll need to type the string just as I've written it above. This feature conserves space on the screen so that you can get up to 15 bytes on one line. This is the maximum amount you use to specify a hex string.

By the way, '20 58 FC', listed in assembler, is 'JSR FC58', which says "do a GOSUB to the monitor routine (at $FC58 - remember, you have to reverse the two bytes of hex addresses when using 6502 machine code) which clears the screen and homes the cursor. It is equivalent to "HOME" in BASIC.

You should now find that the cursor is positioned after the '2-', so hit the '$' again and start string two off by typing in a '20'. Now enter a '=' sign and notice that it is immediately echoed twice after the '20'. You already know from the "Tracer's Instructions" section above that this is a wildcard designator which signifies that any two-digit byte in the position the two '=' characters occupy is acceptable. Finish off by typing 'FE' and hitting RETURN. The '20==FE' entry tells The Tracer to find and display any 3-byte machine code instruction which begins with JSR $FE. In other words, we are interested in having a look at any Jump to a page $FE SubRoutine in the range of sectors we want to search. Page $FE ($FE00 - $FEFF) is in the monitor and contains several subroutines which perform setting up and other functions for the video.

Just to break the pattern a bit, let's also look up some ASCII. The cursor should now be happily flashing away next to

the '3-', so type an 'H', followed by the word "HELLO". Finish with a RETURN as usual. The 'H' tells The Tracer that the string following it is in high ASCII. Type another 'H' for string 4 and enter the word APPLESOFT, followed by a RETURN.

Now get ready for string 5 by typing an 'L'. This indicates that the string is to be in low ASCII (high and low ASCII are explained in Appendix A at the back of the book). Just to get acquainted with the error correcting facility, type the misspelled word "CATALEG". Now correct this by hitting CTRL @. The entire string, plus the 'L' string type designator, will disappear from the screen, and the cursor will be returned to the '5-'. From here you could decide either to restart the menu with ESCAPE, or to jump back to Tricky Dick with CTRL C.

Don't use either of these commands, however. Instead, key in 'L' again, and type out the following string: CATALO=. When I discuss the results of the search I'll explain why you've replaced the terminal 'G' of this word with a wild card.

Hit RETURN to go to '6-', but because we're not going to look for any more strings on this search, do another RETURN. This should bring the cursor down to the RANGE section of the menu, next to "LOW TRACK-". The range we will search on this trip will extend from track $00, sector $00 up to track $02, sector $04 (this takes in all of DOS). So type in the following sequence: 0 RETURN 0 RETURN 2 RETURN 4 RETURN. If all has gone well, this should leave the lower part of your menu looking like the illustration below.

```
          STRINGS:

          $1-2058FC
          $2-20==FE
ALL       H3-HELLO
OK?       H4-APPLESOFT
          L5-CATALO=
           6-

RANGE: LOW  TRACK-00      HIGH  TRACK-02
       LOW SECTOR-00      HIGH SECTOR-04
```

Now hit CTRL Z a couple of times and retype the data you've [illegible]. Then answer 'Y' to "ALL OK?", starting the search.

The first string The Tracer will locate is string 3 ("HELLO). The "T= S=" address shows you that this is on track $01, sector $09. In order to point out the string, The Tracer has transferred control back to Tricky Dick and put the latter's cursor on the first byte of HELLO. The Tracer has also adjusted the TD display so that the line containing HELLO (line $70) is at the top. Counting along this line until you get to the cursor tells you that the string's first character is in byte $75 of the sector. Finally, The Tracer has put the inverse message "STRING 3" in the upper right corner of the screen where "BY T TSE" used to be. You are now effectively in Tricky Dick and could change the file's name on the disk as discussed in chapter 2, or for that matter, use any of the TD commands.

For the moment, however, let's skip the commands and continue tracing our strings. In order to do this, you have to transfer control back to The Tracer by hitting CTRL E, Tricky Dick's instruction to jump to an external module. This time, however, The Tracer knows that it is in the middle of a search and doesn't display its introductory screen as the modules normally do when summoned in this manner. Instead, The Tracer keeps looking through the same sector for all of your 5 strings.

You'll find on pressing CTRL E that in less than an eyeblink, The Tracer has found an occurrence of string 4 ("APPLESOFT") near the end of the same sector (at byte $B8), and has displayed it in the usual way. This word tells DOS the principal language the machine is likely to be using.

Hit CTRL E again and The Tracer looks through two more sectors, stopping at track $01, sector $07, and putting the TD cursor on byte $D2. The "STRING 5" message in the upper right corner tells us that "CATALO=" has been located. Notice here that The Tracer can't put the line containing "CATALO=" at the top of the display because this line is too near the end of the sector.

If you inspect the ASCII display at the right of the screen, you'll probably recognize most of the familiar DOS commands. This tells us that we have arrived at the DOS command table which is used to compare direct instructions that come from the keyboard. But why did we put a wild card at the end of "CATALOG"? Notice that the first few letters in CATALOG are in

Changing the last byte of a string from low to high ASCII (or vice versa) is a common programming practice in machine code. It gives the program an easy way of recognizing the end of a string - just look to see if the high bit of the last character is different from that of the other characters. Therefore, it is always wise to substitute a wildcard for this character when searching for string whose representation you are uncertain of.

You could now elect to change the CATALOG command to something else while still in Tricky Dick (DOGALOG perhaps?). The only snag is that the new name must contain the same amount of letters as did the old. If you wish to shorten any of the DOS commands, you have move all the commands ahead of it in the table down a number of spaces equal to the number of letters missing. So, for example, shortening CATALOG to CAT, though useful, would be rather Tricky using Dick, since moving every subsequent command down 4 spaces would have to be done by hand. There have been many programs featured in the users mags which enable you to do this automatically, so it's probably wiser to employ one of them.

Hitting CTRL E takes us back on the trail of our 5 strings and results in The Tracer turning up one of them on track $00, sector $09, byte $C8. The display tells us that we have unearthed an example of String two, which was 20==FE (JSR to any address in page $FE). The string is 20 93 FE, a subroutine call in the monitor which does a PR hash 0.

Another CTRL E leads The Tracer to find a second occurence of String two, this time in track $00, sector $01, byte $44. This time it's a 20 89 FE which is the same as IN hash 0.

Taking the search further leads us to track $00, sector $001, byte $D0 at which we find the first example of String 1 (2058FC). As I mentioned earlier, this is the machine code equivalent of "HOME" in BASIC. Here, CTRL E reveals yet another occurence of string 2 in line $38. It's 20 89 FE again and CTRL E locates 20 93 FE right behind it. A final CTRL E takes us to the "END OF SEARCH RANGE" menu. Hit a '1' at this point to get back to the main menu. YOU'LL NEED TO USE YOUR CIA ORIGINAL DISK FOR THIS NEXT SEARCH, since it has one of the targets you'll be looking for.

From the main menu, reply 'N' to VERIFY FORMATTING, but 'Y' to T&S LISTS, CATALOG SECTORS, VTOC, and STRINGS. When the cursor gets to '1-', enter GOLDEN DELICIOUS as a high ASCII string. Then nominate track $10, sector $0F as the low address of the search range, and track $15, sector $0F as the high end. Start the search and continue it each time The Tracer happens upon one of your search options.

The first thing you'll notice is that there are several T&S lists between track $15 and track $11. Each time The Tracer discovers one of these it will put Tricky Dick's cursor on the first byte of the sector, and a T&S LIST message will appear in the upper right corner of the screen. When you get to track $11, sector $0F, The Tracer will send you a "CATALOG" message in the usual place and display the first sector of the catalog with the cursor again on the first byte. Continue the search with more CTRL E's and The Tracer will display a couple of more catalog sectors. This is not exactly astounding since we made no effort to protect the CIA disk. However, a catalog search could come in handy when you are working with the software of less generous publishers whose catalog (if present at all) may turn up anywhere on the disk. I'll talk about this a bit more in a later section of the chapter.

When you get to track $11, sector $01, you'll get a somewhat different reaction from The Tracer than usual. You will see a "STRING 1" message with a flashing 'O' in front of it. Line $08 will be in the top of the data display, and it will look like this:

```
08: C9 CF D5 D3 A0 A0 A0 A0 :IOUS    :
                \
                cursor is here
```

The flashing 'O' tells you that string one ("GOLDEN DELICIOUS") overlaps two sectors. In such a case, The Tracer places the TD cursor on the last byte of the string, and puts the line which contains it at the top of the display. More of this hallowed name can be seen in line $00. However, to view and/or change the very first part of the string, back up one sector to sector $02 by hitting the right arrow. Now do a CTRL R and get to the bottom of the data display. You will see the first part of the string ("GOLD") in line $F8.

However, you've just altered the track and sector address while in Tricky Dick. Recall from the "Tracer's Instructions" earlier that if you do this before The Tracer has reached the end of the search range, the search will continue from the altered address. So having just changed 'S=' to $02 in order to read that sector, you will find that CTRL E leads The Tracer to search it and rediscover string one on sector $01. In order to make sure that sector $01 gets fully inspected, however, just let this happen, and continue the search with the familiar CTRL E.

The next search option The Tracer will uncover is the VTOC sector on track $11, sector $00. Again, not too astounding, but a useful option when working with a protected disk whose VTOC has been moved. While still in the VTOC, change Tricky Dick's track number to $10 and the sector number to $05. Remember that changing the track and sector address before The Tracer reaches the end of its search range gives you a chance to change your mind in the middle of a search and either skip over some sectors, or go back and have a second look through some that have already been inspected.

Before hitting CTRL E to start the search again, take note that I want you to hit CTRL A as soon as the search is in

progress. Ready? Good, hit CTRL E and follow this up quickly with CTRL A. This will result in a "SEARCH ABORTED" message and a choice of going back to Tricky Dick or the main menu. Type in a '1' to do the latter.

Before doing anything else, hit CTRL D for "default" and the cursor will jump straight to the RANGE section. This means that The Tracer will recall the last search options you selected and is prepared to track them down through [illegible] range you specify. Remember that you can choose this option only when you come to the menu from a Tracer abort, end of range message, or error message. Now make track $12, sector $00 the low end of the range, and track $1F, sector $0F the high address. Hit RETURN, and before long The Tracer will come up with a T&S List. Ignore this and open the drive door. Now hit CTRL E, and surprise, surprise, you get a "SECTOR UNREADABLE" message.

The accompanying menu presents you with the usual choices of proceeding to Tricky Dick or The Tracer. But you also can elect to keep on trying to search the disk, either taking up your mission on the very next (i.e. the next lower) sector, or electing to skip to the next track. Close the drive door, type a '2' to take this latter route, and you'll notice that the 'T= S=' address changes immediately to the next lower track, sector $0F. It then continues its usual sector-by-sector descending search from there. This menu will become highly familiar as you search protected software, since its DOS marks may vary from sector to sector. You'll also see it as you work with normal DOS 3.2 disks whose blank sectors are unreadable due to their lack of data marks

When you arrive at the next T&S list, hit CTRL A while still in Tricky Dick. This aborts the search and removes the message from the upper left corner of the screen, replacing it with "BY T TSE". You have effectively turned off The Tracer's disk reading routines, and if you now hit CTRL E, the search will not continue. Instead, you'll get The Tracer's Introductory screen. Try a CTRL E just to check, and then do a CTRL C to get back to Tricky Dick.

Well, the last few pages should have given you a pretty good chance to rehearse most of The Tracer's instructions. Now that you two have a good working relationship, let's get down to some more interesting stuff.

Tracing Lost Files With The Tracer

Suppose you are working with your machine and suddenly realize that you have DELETE'd a file you really wanted to keep. In chapter 2 we discussed how to fix this in a matter of seconds, *providing you hadn't put any more files on the disk since the deletion*. But suppose further that you have made the supreme blunder of actually SAVEing a couple of programs before realizing that the DELETE'd file consists of, say, the only copy of your nearly complete PhD thesis. You certainly will have read the

dire warnings, repeated time and again in the popular computer literature, that this particular mistake usually causes the file concerned to be overwritten with the SAVE'd programs.

Since you somehow just never got around to making a backup of your thesis, must you now forsake the halls of academe for a less stressful environment, perhaps taking up work as a sludge scraper, or at some other profession that doesn't require you to create word-processed documents? Well, thanks to your sagacious decision to become the owner of The CIA disk, you may yet land that assistant professorship you had your eye on.

In order to determine the possiblity of salvation from the foregoing horror, I'd like your to carry out a little experiment. Put a *newly initialized disk* in your drive and type in the following sequence of BASIC instructions with a RETURN after each one.

```
FP
10 REM TEST 3
SAVE TEST 3
DELETE TEST 3
10 REM TEST 2
SAVE TEST 2
DELETE TEST 2
10 REM TEST 1
SAVE TEST 1
DELETE TEST 1
10 REM TEST 0
SAVE TEST 0
DELETE TEST 0
```

What you have obviously done is to create, SAVE, and subsequently DELETE, 4 APPLESOFT programs. Doing so in the sequence specified above has given each of TESTs 0 - 2 a darn good chance to overwrite one of the programs that preceeded it.

In other words, you have committed the Final Fatal Foul-up no less than 3 times! And so, we ask ourselves, how many of your one-line programs are still intact on the disk, and how many have been obliterated? To answer this pithy question, carry out the operations below with Tricky Dick and The Tracer in memory.

Set up the Tricky Dick parameters to read the disk with the 4 test programs on it and go to the The Tracer's menu. Then select the T&S LISTS and STRINGS options. Type an 'L' for string one and enter the word TEST. The search range should extend from track $12, sector $00 to track $19, sector $0F. Type these numbers into the menu and start off the search.

The Tracer's first stop will probably be somewhere pretty close to track $16, sector $0F, where you will find a T&S list. The next CTRL E turns up string one in the very next sector as part of the "TEST 0" REM statement in program TEST 0 (you could check out the entire program by listing it in APPLESOFT with CTRL L, 'A', and 'L'). This is no big deal, however, since TEST 0 was the last program we saved, and hence couldn't have been clobbered.

Continuing the search will bring you to sector $0F of the next lower track where you'll discover another T&S list. CTRL E again uncovers the program TEST 1 on the sector $0E of the same track. We can see that its code is all intact, establishing that it was not overwritten, even though we DELETE'd it and SAVE'd another program thereafter.

To make a long story short, if you keep pumping in CTRL E's, you'll get exactly the same results for the other two files, neither of which have been replaced by first DELETEing them and then SAVEing additional programs afterwards. All programs will probably reside on sectors $0E and $0F of 4 consecutive tracks. This happy state of affairs is due to DOS having a built-in propensity to avoid overwriting DELETE'd files. This is because it records the last track it accessed in the VTOC (at byte $30) and attempts to find free space on a track other that this one when it performs its next disk operation. As you may remember from our work in Chapter two, the next VTOC byte ($31) tells DOS which direction from the last accessed track to take when looking for empty sectors. Please note, however, that this does NOT guarantee that a happy ending to the foregoing saga of the lost thesis would always be obtainable - so don't get careless, O.K.?

Another important use for the search procedure you used in your little experiment is to locate lost files when one or more catalog sectors have been clobbered. Once the catalog is destroyed, of course, there is no record on the disk of the starting address of any of the disk's files. Never mind, though, because the Tracer makes a rescue operation child's play.

The first step is to search the entire disk for T&S lists, together with any strings or other code that you think might exist in a missing file. Remember to specify these strings in low ASCII if you're trying to recover an APPLESOFT program, and high ASCII for an INTEGER program. High ASCII is usually used in textfiles as well. If you can remember any specific instructions, look up the their BASIC tokens in the Apple manual, and enter these in the menu as a hex string. Whenever The Tracer finds a T&S LIST, always be sure to list the first few sectors it

contains with Tricky Dick. Inspect them carefully to make sure they belong to a file you want to recover. If they do, make a note of the track and sector upon which the T&S list is located, together with the number of track and sector pairs in that list.

Once you have found all the missing files, the next problem is to restore the catalog track and put the file entries back. Chapters two and three describe the composition of catalog sectors in detail, and referring to them should help you in carrying out this reconstruction. Rather than do this completely from scratch, try the following sequence.

(1) Put Tricky Dick's cursor on the first byte of one of the clobbered catalog sector and type CTRL Z. Write the zeroed sector back to the disk. Repeat this for every corrupted sector.

(2) Restore the link pointers of each zeroed sector. You'll recall from chapter two that these specify the track and sector of the next catalog sector. They occupy bytes $01 (the track number) and $02 (the sector number) of each catalog sector. On a normally formatted disk, each sector's two bytes point at the next lower sector.

(3) With the BASIC prompt on the screen, type FP. Then key in SAVE followed by the name of the first APPLESOFT program whose name you wish to restore to the catalog track. Repeat this procedure with each of the other programs to be rescued. If the program is a binary file, use BSAVE. Use BSAVE for all textfiles and INTEGER programs also. This step puts the file names back in the right places and partially restores the catalog sectors.

(4) Replace the length values of each file name with the correct values you have copied down (i.e., the number of track and sector pairs in the file's T&S list). These come at the end of the 'A0's' after the file name and, until you change them to the right values, will read '02 00' for each file entry.

(5) Replace the present address of each filename's T&S list with the correct address (which you wrote down during The Tracer's search). This address consists of two hex bytes just in front of the file type byte.

(6) For INTEGER program and textfile names only, put the correct file type designator ('01' for INTEGER, '00' for textfiles) in place of the one that is now there (an '04' if you followed my instructions above). Remember, this designator byte is the first hex value in front of each file name.

Though the foregoing procedure involves doing some operations by hand, my benchmark trials with Apple users as subjects shows that it is a far faster way to resurrect lost files than using any of the "automatic" file recovery software presently available. This is because the automatic software restores every file on the disk to the catalog whether you really wanted it or not. Most of my subjects found this an unacceptably

slow process.

Remember to refer to Chapter 2 for more details of the placement of the hex values in the catalog sectors. Just to keep in the spirit of this tutorial, I suggest you take your newly initialized disk with the TEST programs on it and deliberately clobber its catalog sector. Then try restoring all 4 TEST programs plus HELLO to the catalog. If it doesn't work the first time, blow away the catalog again and have another try. This will get you in good shape for the Real Thing.

Using The Tracer for Text Retrieval

Suppose you have a substantial number of disks full of word-processor files which are mainly correspondence. Moreover, you've got a meeting with your accountant tomorrow and have to bring copies of every letter you've written to Plummet Airlines. To make things worse, perhaps you (like me) have a memory which is even less reliable than Plummet's arrival times. How are you going to dig out all those letters?

In fact this job is pretty simple because all you do is enter PLUMMET AIRLINES in high ASCII in The Tracer's menu (although you'd better check one or two of your correspondence files just to make sure your word processor doesn't use low ASCII or some method of non-ASCII text compression). Then you search every disk from top to bottom, using this string as a search argument. When you find only the string in a piece of correspondence, make a note of the name you gave it when it was written to the disk. When you're done, you produce hard copy of all the letters whose names you wrote down.

Of course, there could be one minor hitch in all of this - you might not be able to remember the letter's filename when you see it in Tricky Dick's display. If so, you will have to find its T&S list, and from there, go to the catalog track to match the T&S list's address with the letter's name. This is going to require a bit more work - but it serves you right for having such a lousy memory!

Fortunately there are a couple of shortcuts you can use to track down an unknown file's catalog entry. Firstly, to find the file's T&S list, specify a range of its tracks and sectors as a Tracer search string. Let's say Tracer has located the words "Plummet Airlines" on sector $05 of track $14. The commands CTRL A and CTRL E, followed by any keystroke, will take you back to The Tracer's menu. Once here, you select only the STRINGS option and enter string one in hex as shown below.

STRINGS:

$1-1405

This track and sector pair must appear somewhere on the file's T&S in the order illustrated above. The Tracer will most

likely find this T&S list in under 15 seconds if the disk is nearly full.

By the way, remember not to also select the T&S LIST, CATALOG SECTORS, or VTOC options on the menu for this search. As I mentioned earlier, when the Tracer finds one of these, it doesn't search the same sector for any strings you might have specified.

When you've got the track and sector address of the file's T&S list, the final step is to go back to The Tracer's menu and specify these two bytes as a hex search string. So if the list was on, say, track $15, sector $0F, your string entry should look like this: $1-150F. You should now search all the catalog sectors on the disk (i.e from track $11, sector $0F, down to sector $01 of the same track). This search will probably take less than 3 seconds. If all has gone well, The Tracer will place Tricky Dick's cursor exactly 3 bytes in front of the filename you are looking for. This, of course, is because DOS writes the address of each file's track and sector list in this position.

If you find more than several unknown files on the same diskette, you can track down up to 6 of them, simultaneously. First, go through the entire disk making a note of every track and sector address in which your search string occurs. Then specify up to 6 of these pairs as Tracer search strings, writing down each T&S list that The Tracer turns up. Finally, enter all the T&S list addresses as search strings, and put The Tracer to work on the catalog track.

Of course, you don't have to limit yourself to correspondence files when using this technique. You can apply the same procedure to any stored textfiles created by any database program, providing the files contain recognizable alphanumeric characters. You can also use the same technique for locating programs.

Searching Disks with Modified DOS Marks

Everything you have carried out so far in this tutorial can also be done with the vast majority of protected disks presently available. If you have worked carefully through the last chapter, you'll already have a pretty good idea about how to search locked software with The Tracer. With many nonstandard disks, all you have to do is utilize the methods I described in that chapter for determining and setting up the Tricky Dick sector marks, before entering The Tracer.

To get some practice in using The Tracer in this way, let's search track $22 (the one I've protected) on your CIA disk for a couple of strings. From Tricky Dick, specify which drive the disk is in and set the DOS version to 3.3. Then jump into The Tracer. Hit CTRL S and set the address field header to 'DC BF AA'. Select the STRINGS option only, and specify two high ASCII search strings as follows:

H1-SECTOR
H2-INITIALIZE

Set up the SEARCH RANGE to examine all of track $22 and start the Tracer on its merry way. After a bit hesitation on $0F, the sectors will clock by in their usual manner until The Tracer gets down to $03. Here, The Tracer will vainly try to do a read, but will fail, displaying the SECTOR UNREADABLE menu. I might add that if you were not working on the CIA disk, with whose track $22 ideosyncracies you are already familiar, it would have been useful to have first used The Tracer to VERIFY FORMATTING on all $23 tracks. If problems were encountered, the disk's formatting should have next been checked out with The Linguist. Armed with the information on DOS marks gleaned from this member of the CIA, you would then have been able to carry out the proceedure that I'm going to describe next.

From your work in the last chapter you may remember that, in order to gain access to this sector, you have to change the data field header from 'D5AAAD' to 'F8C7D8', and the data field trailer from 'DEAA' to 'FFFF'. To make these changes, ignore the 4 choices on the error menu, and hit CTRL S instead. After fixing up the header and trailer, a RETURN causes the cursor to disappear and the previously inaccessible sector to be automatically reread using the new parameters.

A considerable amount of noisy recalibrating will take place as The Tracer gets to grips with the new marks. Then the familiar "ping" will be heard as the Tracer transfers communication to Tricky Dick and displays string two. By the way, if you haven't already done so, you might find it interesting to read the contents of this sector and the next one we are about to search.

An alternative to the foregoing procedure is to select the "TRACER MENU" option when the Tracer does a misread. From the main menu, hit CTRL S and change the sector marks to those necessary to read the present sector. Then hit RETURN, followed by CTRL D. This latter command takes the cursor directly to the RANGE section where you enter track $22, sector $03 for both the high and low addresses of the search range. Then start the search in the usual way.

A CTRL E here unearths another occurrence of the word "initialize" in the same sector. On the next Tracer call the search continues, again unsuccessfully, on sector $02. The data field header and trailer of this sector are DEAAEB and D5AA, respectively, so change them accordingly and hit RETURN (or go back to the main menu and carry out the alternative procedure I described above). The next attempt should pick up another example of string one. Follow the same plan with sector $01 whose data header is D9AAAD, and whose data trailer is normal. Do the same for $00 where both the data header and trailer are normal. The word 'sector' should be flagged on both these last two sectors.

Searching Disks with Modified Sector Numbers

An occasionally used method of copy protection is to alter the sector numbers in the address field information to nonstandard values. For example, it is not too difficult to format a 3.3 disk so that the sectors on one or more tracks do not range from $00 - $0F, but instead vary, say, from $10 - $1F. By first checking the disk with The Linguist, you can easily detect this and still get at all the sectors with Tricky Dick, since the latter allows you to use the ';' command to enter any 2-digit hex number after 'S='.

The Tracer, however, automatically returns the sector number to $0F (or $0C if DOS 3.2 is active) whenever it finishes one track and moves on to the next. So what you have to do is fix The Tracer so that it sets this value correctly according to the numbering scheme on the track. You first need to find out with The Linguist the highest numbered sector on the track or range of tracks you want to search. Then go into the monitor with CALL -151 and type B11B:A9 XX EA, where XX is the highest sector number. This patch to The Tracer will enable you to read and analyse the entire track with ease. Don't worry too much about the lowest sector; if this is less than $08, The Tracer will still clock the sector numbers by until it reaches $00, at which point it proceeds to the next track. With a value greater than $08, wait for the error menu and take the "NEXT TRACK" option.

When you are through searching the disk with the funny sector numbers, don't forget to either BLOAD The Tracer again, or go back to the monitor to put things back in order with B11B:AD 0A 80.

Another alternative numbering scheme consists of incrementing the sector numbers by even amounts, e.g. $02, $04, $06, etc. If you encounter this subterfuge, specify a one-track search range and set the high address to the highest numbered sector you found on the track. During the search, The Tracer will display the error menu every time it attempts to read an odd numbered sector as it steps through the range. When this happens, just take the "NEXT SECTOR" option, and the track will get fully searched.

Searching Half- and Strangely Numbered Tracks

Having dealt with half tracks previously, you should have no trouble here. However, the proceedure is a bit more involved, and you need two disk drives to bring it off. Here is the proceedure.

(1) Put the CIA disk in drive one and the target disk in drive two.

(2) Get The Linguist into RAM and position the drive two disk arm over the track to be read.

(3) Load in The Tracer from drive one.

(4) Put the half track patch into Tricky Dick's RWTS as described in Chapter 5.

(5) Go into The Tracer and specify a search range from sector $00 to sector $0F of the track to be examined. Make sure you enter the same track number in the menu that appears on the track you're going to search.

Now go to it. If you hear a recalibrate, you may have to start again. Just to make sure you've got the hang of searching half tracks, carry out the above sequence of operations on track 20 1/2 of the CIA disk. Search a few of the sectors for the word "this" in high ASCII.

Searching tracks which are numbered in a nonstandard way requires a bit of patience but can also be handled if you have two disk drives. All you have to do is follow steps 1 - 5 above for each track you want to search.

Finding the Byte and Nibble Translate Tables

If you're dealing with a normal DOS, you know from the previous chapter where these are located. But on protected disks, it's just possible that the two tables may have been moved. This means that you have to find them before you can do any editing.

Finding the tables is usually no sweat. Once you've determined how to read the disk, simply instruct The Tracer to search the disk for the following strings.

for a DOS 3.2 disk	for a DOS 3.3 disk
$1-010810	$1-96979A
$2-141516	$2-BEBFCB
$3-3EE8F0	$3-FCFDFE
$4-ABADAE	$4-000198
$5-D6D7DA	$5-C0C1C2
$6-FDFEFF	$6-3D3C3D

Half the strings in each column consist of 3 consecutive bytes from the byte table of their respective DOS versions; the other half are similarly constructed from the two DOS' nibble tables. Since the reason for finding the tables is to determine if any of their bytes have been altered, it makes sense to search for several several small chuncks of each of them. That way, you maximize the liklihood of specifying search strings which exactly correspond to parts of the tables and hence increase The Tracer's chances of turning them up. Needless to say, if the above strings don't work (a highly unlikely possibility) pick some different chunks and try again.

When The Tracer flags the tables, the final step is to compare them with the normal vesions shown in the chapter on The Code Breaker. If any bytes are different, make a note of them and use The Code Breaker to edit the tables accordingly.

It looks like we're nearing the end of our little journey together. If you have followed along with me all the way, your disk savvy ought to have grown by a substantial amount, and by now you should be commanding some considerable respect among your computer buddies. However, we haven't quite reached the end of the line yet. I still have to introduce you to one more CIA operative. And that's The Tracker.

CHAPTER EIGHT — The Tracker

The Tracker's job is to follow the disk arm from sector to sector as it makes its rounds during any DOS operation. Once on the trail, The Tracker faithfully records on your screen each sector visited and all read or write operations performed while DOS is LOADing, SAVEing, RENAMEing, or doing anything else to a file. The entire record stays on your screen for as long as you wish, and if you need a permanent copy, The Tracker dumps the contents of the screen to your printer at the touch of a key.

The Tracker's activities enable you to pinpoint the exact sector of a corrupted file or catalog where things are going wrong. This enables you to make short work of locating the source of the problem and subsequently fixing it with the other CIA utilities. Or, how would you like to like to put DOS under a magnifying glass, watching every move it makes as it carries out any disk operation. The Tracker acts as a private tutor in this regard, showing you just what DOS is doing during any disk access. The Tracker can even be used with some protected disks, helping you find where hidden-nibble count tracks and other protection devices are located. These and other uses will be detailed, as usual, in the Tutorial which follows a brief summary of the instructions.

MAKING THE TRACKER TRACK

The best way to use these instructions is in the now familiar manner - carefully read through them, trying out each, and then get stuck into the tutorial which follows.

The Tracker Stands Alone

Unlike the previous 3 CIA modules you have been working with in this book, The Tracker does not need Tricky Dick in memory in order to function properly; all of its routines are self-contained. The Tracker works on its own, installing several DOS patches in crucial places so as to momentarily divert the flow of control out of DOS and through itself whenever a sector is read or written. After carefully monitoring DOS' activities during this brief detour, it passes control back to DOS and [illegible] waits for the next burst of information. All of the data it collects are passed along to your video monitor for your scrutiny. They remain there as long as you like.

The Tracker Prefers DOS 3.3

The Tracker was chiefly designed to be used with a standard DOS 3.3. It won't work with 3.2, and may have problems with a modified DOS, including any of the many speeded-up versions currently available.

Getting The Tracker on the DOS Trail

Start by typing in RUN TRACKER, whereupon you will be given a choice of two memory areas in which to place The Tracker. The first is between DOS and its buffers. If you choose this option, the BASIC loader program now running will throw a couple of fast POKES into DOS and finish up with a DOS subroutine CALL to move the buffers down in memory. This creates an insulated "hole" in which The Tracker is automatically BLOADed, and from which it operates with full assurance of not being clobbered by any BASIC program you load into RAM.

The second place that can serve as The Tracker's headquarters consists of the 5 pages of RAM beginning at $8800. This area comes in handy when you wish to follow DOS through the loading of a program which itself makes use of space above the buffers. In fact, $8800 is pretty safe most of the time, since few programs put any code there.

When you type in your choice, you will be asked if you're sure about the location. If you type a 'Y' and The Tracker will be summoned; if you typed an incorrect instruction, hit an 'N' to restart the menu.

By the way, just before BRUNing The Tracker, the loader program commits suicide by overwriting part of itself. This is to [illegible] in memory which might interfere with your work. So if you want to take a look at the loader itself, type in LOAD TRACKER, followed by LIST (the instructions which change the DOS buffer locations are in lines 1000 - 1020, in case you're interested).

Once The Tracker is BLOADed, hit any key to [illegible] screen and start the menu.

The Menu

The Tracker's menu gives you one or two display options, depending on how you respond to the first. Each is followed by a default value which can be accepted by typing a RETURN.

The first thing you will see is the TRACKS ONLY option. If you type 'Y' in response to this, the Tracker will only show you the tracks being visited during a DOS operation; it will omit the sectors being accessed. However, if you hit RETURN or 'N' the sector numbers will also be shown.

The next choice to appear is LOGICAL OR PHYSICAL SECTORS. A RETURN or 'L' takes the logical sector default; a 'P' tells The Tracker to show you the physical numbers. The difference between [illegible] two sector numbering schemes is explained in Chapter 4.

The final question asks "IS THIS OK" [illegible] your choices, type a RETURN or a 'Y'; if not hit an 'N' to restart the menu. An ESCAPE in response to any question also

reruns the menu.

The Tracker's Display

During any DOS disk operation, the tracks and sectors are displayed from left to right in the order that they are visited by the disk arm. This is carried out in real time, with the number of each track and sector getting written on the screen the moment it is accessed. The track numbers are printed in *inverse* and the sector numbers appear in normal type. In order to save screen space, The Tracker never repeats the same track number twice in a row. Thus, every sector number following a track number refers to a sector on that track. This means that if your last DOS command was CATALOG, the display of the next DOS operation will start off without a track number. This is because the catalog track and the first track accessed during any disk access are the same (track $11).

You'll also notice that an inverse 'R' or 'W', followed by a '>', will appear on the screen from time to time. The 'R' tells you that a read cycle is in progress; the 'W' indicates that information is being written to the disk. To conserve screen space, these letters appear only at the beginning of each cycle. In other words, when DOS stops reading and starts writing, or vice versa, the corresponding letter appears on the screen and applies to all the track and sector numbers following it. You'll find in many cases that DOS switches between these two functions several times during a single operation, for example while RENAMEing or SAVEing.

All the information The Tracker outputs to your video monitor is listed horizontally across the screen in rows, starting at the very top, and moving down, row by row. If the screen fills completely and The Tracker is still collecting data, it jumps back to the top of the screen and keeps on writing. This, of course, is going to overwrite some of the data just recorded. However, this almost never becomes a problem, and you'll find that the track and sector numbers accessed during most DOS operations won't take up the entire screen.

When The Tracker prints a row of information on the screen, it also moves the text window down one line. This means that you can't accidentally type something into your Apple and clobber the display. If you issue a HOME command when Tracker information is shown, the screen will be cleared only up to the last line of data. Similarly, a CATALOG will cause the filenames to scroll up underneath the data display.

When a DOS operation is completed, The Tracker puts a flashing '**' after the last sector accessed. This enables you to instantly find the end of the current display, irrespective of anything else that happens to be on the screen. If you watch closely when The Tracker is tracking a DOS operation, you'll see the '**' racing frantically along ahead of the other bytes. If other garbage is present on the screen when this is happening, you may also notice that The Tracker clears a path through it two bytes ahead of the information being written.

If you RUN or BRUN a file, it will load into memory and The Tracker will display in the usual way all the tracks and sectors being accessed. However, the program will not start running. Instead a flashing 'R' will appear at the end of the data display. This is because the very first thing many programs do is reset the text window and clear the screen to prepare for displaying a logo, menu, or what have you. Naturally, all the data The Tracker has just printed instantly goes down the tubes in such a situation. So The Tracker stops DOS from running the program and waits until you have had time to examine and digest the information. When you are ready to continue, hit any key (except 'P') and the program will start up.

This feature comes in handy when you are tracking a program, one of whose first functions is to load or run another program. In such a case, one or both of the programs are highly likely to replace Tracker information with their own display before you get a chance to see what's going on.

If you get a flashing 'R' and you decide that you want a permanent record of the information displayed so far, turn on your printer and hit the 'P' key. This causes the entire screen to be dumped to the printer. YOUR PRINTER CARD MUST BE IN SLOT 1 FOR THIS OPTION TO WORK. When your hard copy is ready, hit any key to run the program just loaded.

The Tracker's Use of RESET

You can instantly get the menu back at any time by pressing RESET. A little later on, I'll show you how this can be disconnected.

The Tracker Lives On

Once The Tracker is installed, it remains "live" until you disconnect it as described below, or reboot DOS. This means that if you load in a file, run it, modify it, or whatever, The Tracker waits patiently for the next disk access. You'll sometimes find that in the course of your work you will have completely forgotten about The Tracker. Then, when you least expect it, The Tracker will pop up again and string data across the screen the very instant the disk drive starts spinning.

The Ampersand and Control-Y Options

Once The Tracker is installed, you can type an '&' from BASIC or a CTRL Y from the monitor and a "YOUR COMMAND?" message will appear. This enables you to make use of one of several options by responding with one of the letters shown below.

B returns you to BASIC if you typed a '&' to get the command

prompt up, or to the monitor if you got the prompt with a CTRL Y. This option is useful in case you hit an '&' or CTRL Y by mistake.

R revectors RESET back to its usual destination in the monitor. Once you have issued this instruction, a RESET will no longer take you to the menu. This means that you can interrupt a disk operation with RESET (after first opening the drive door, of course!). This is handy if you notice that during a particular access, code is being executed which wipes the screen without first engaging the The Tracker's flashing 'R' trap.

U unhooks The Tracker so that it does not output any data during a DOS operation. However, even after this instruction has been given, The Tracker lies dormant, waiting to leap into action once again if you issue an 'S' command as explained below.

Z (zap) unhooks The Tracker and puts the DOS buffers back in their normal location. You only need to use this instruction if you have elected to load The Tracker between DOS and its buffers and now wish to INIT a disk. A disk INITed with the buffers moved is likely to crash. Once having used 'Z', you should RUN TRACKER and start all over again if you wish to work further with The Tracker.

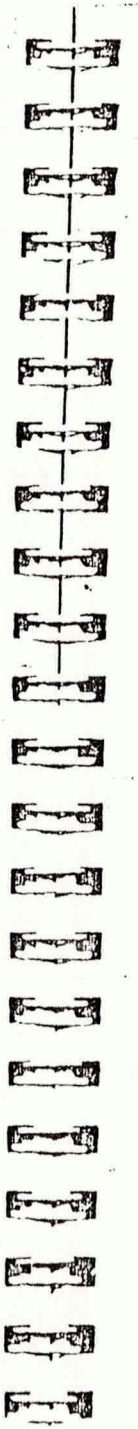
S starts the Tracker up again from the very beginning (handy if you previously issued a 'U' command and want to get The Tracker back). If you use this instruction, the RESET, ampersand, and CTRL Y vectors will be set up by The Tracker, and the menu will be displayed.

P dumps the screen to your printer. This enables you to make a permanent record of The Tracker's display at any time. Before issuing this instruction, make sure your printer is turned on and the printer card is in slot one. Otherwise, the program will hang.

After you have made your selection, The Tracker will ask "ARE YOU SURE?" If you are, hit a 'Y' or RETURN and you command will be carried out. If you type an 'N' here, the "YOUR COMMAND" prompt will appear once again, giving you a chance to change your mind.

THE TRACKER TUTORIAL

Put The CIA disk in your drive, type in RUN TRACKER, and select the "run at $6800" option. Hit any key to get rid of the introductory screen, and in response to The Tracker's menu press RETURN 3 times, taking the menu defaults. You are now ready to start tracking, so place a diskette in your drive newly initialized with the following HELLO program:



```
10 HOME
20 VTAB 14:HTAB 17
30 PRINT "HELLO
```

When this is done, key in LOAD HELLO. The drive will start spinning and The Tracker will start shadowing the disk arm, spewing out information as it goes. When the LOAD is complete, this series of bytes should be at the top of your screen:

```
11 R> 0 F 12 F E
```

In my illustration, the numbers in boldface type should be in inverse on your screen. The inverse '11' tells you that the disk arm moved to track $11 first of all. The inverse 'R' means that RWTS next switched into a read mode. The next two numbers indicate that the arm went first to sector $00, then to sector $0F on track $11. As you probably recall from previous steps, by now, track $11, sector $00 contains the VTOC, and sector $0F on the same track is where the catalog begins. All DOS operations on a normally formatted disk begin by accessing the VTOC, and then going to sector $0F to start searching for the specified file name.

In this case, DOS found the HELLO file name in sector $0F, as you might well expect, and also read in the location of its track and sector list in the same sector. The next inverse number, a '12' tells us that the arm swung over to track $12 to find the T&S list, the first stop in loading in any file. The 'F' indicated that HELLO's T&S list is on sector $0F of track $12. DOS read this list in so it could find the rest of the program's sectors. In this case there is only one more sector - $0E on track $12. The 'E' in the display tells us that DOS read this sector into RAM, completing the LOAD. Type LIST to confirm that the program is right where it ought to be.

Now hit RESET to return to the menu and type 'Y' in response to the first option. This tells The Tracker to display tracks only. If you now LOAD HELLO, you'll get just an inverse '11' and '12, the two tracks visited.

Hit RESET again, and this time answer 'P' to the "LOGICAL OR PHYSICAL SECTORS?" question. Now type RUN HELLO. Your display should look like this:

```
11 R> F 0 12 F 2  R
                  |
                flashing
```

This time, you've listed HELLO's physical sector numbers. The only difference between this display and the last one is that the very last sector number is a '2' instead of an 'E'. You'll probably remember from Chapter 4 that sector numbers $0F and $00 are the same in both systems, but that logical sector $0E's physical number on the disk is $02.

The flashing 'R' indicates that a program wants to run but is being held back by The Tracker. This is in case the program does something to wipe out the display when it first starts running. If your printer card is in slot one, you can get hard copy of the information shown on the screen by turning on your printer and hitting 'P' while the R is still flashing. If not, hit any key and the program will run.

If you want to see a more dramatic example of the differences between the two sector numbering schemes, try BLOADing Tricky Dick first with one option in effect, and then with the other.

The Ampersand and Control-Y Commands

Now type in an '&', and in response to the "YOUR INSTRUCTION?" prompt, press the 'R' key. Hit a 'Y' or RETURN in response to "ARE YOU SURE?", and try to restart the menu with a RESET. Nothing will happen because 'R' revectors 'RESET' back to its normal destination in the monitor. However, The Tracker is still live and will continue to clock up DOS data. Now you can use RESET to interrupt a program running in RAM or a disk access (providing you lift the drive door for the latter).

To make the RESET vector point back to The Tracker's menu, type '&' again and hit the 'S' key. This starts The Tracker up from scratch. While you are experimenting with the ampersand commands, turn to The Tracker's instructions a few pages back and try out the other ampersand/CTRL Y options.

Finally, finish off this practice session by RENAMEing HELLO to HI, and DELETEing HI. Then try INITing a disk with The Tracker installed. Notice the intricate and lengthy sequence of operations carried out by DOS during these various functions. If you do, you'll begin to see why so many speeded-up versions of DOS proliferate.

On the Trail of the Wild Dongle

Some protected disks currently on the market are normally formatted and have a standard DOS, allowing you to CATALOG them, and to load and save files in the usual way. However, try as you will, you can't back them up with an ordinary copy program. In almost every case, this indicates that the programs are accessing the disk during loading and looking for something special.

There are several things that software protected in this manner might look for on the disk. The most common target, and by far one of the most resistant to copying, is the nibble-count track (also referred to as a "dongle track"). This is usually a completely unformatted track consisting almost entirely of sync bytes. Buried somewhere in the midst of all the 'FF's' (or whatever value the sync bytes take) is one or perhaps a couple of "lock bytes" which differ in value from the syncs. At some stage during the loading of a program protected in this manner, the track is read and its sync bytes are counted up. This is where the lock bytes come in, since they supply a reference point upon which the counting algorithm can start and finish.

If the total number of syncs read in doesn't match or nearly match a magic number the software publisher has put in a special place on the disk, various nasty things immediately take place (such as a memory wipe, a disk crash, and perhaps even a subtle disk corruption). Since a nibble count track is quite difficult to copy even with a so-called "nibble copier", the number of sync bytes on a copied version is unlikely to be the same as on the original disk. This means that when you try to boot the copy, the nibble count won't be anywhere near the value of the hidden number.

One way round this problem is to disable the routine which counts up the nibbles, an operation often made considerably easier if you know exactly when the track is accessed. The former task requires a considerable knowledge of machine code, but the latter task can be carried out automatically with The Tracker, providing the protected disk is running a normal DOS.

After installing The Tracker, simply run the program with the suspected nibble-count routine. Watch the display closely. If a dongle track is present, it is unlikely to be divided up into sectors. So what you'll see is an inverse track number appear on the screen with no sectors between it and the very next track number. In addition, there will usually be some considerable delay between the printing of these two track numbers while all the nibbles are being read and counted up.

If this occurs, you immediately know the number of the dongle track, and precisely when it is accessed during a load. If you find such a track on one program on a disk, check all the other programs for the same trick. However, to put a stop to this nefarious operation, you'll have to bone up on machine code.

Well, I guess we've just about come to the end of our work together. By now, the CIA utilities should be old friends to you, and you should be as familiar with the Apple disk as you are with your own back yard. I sincerely you enjoyed this tutorial and that the CIA will give you many hours of pleasure.

APPENDIX A — Getting on Top of Hex

Using the hexadecimal number system is a bit like having sex - once you've tried it, you'll never want to give it up (unfortunately, however, I can't think of any other features that hex usage and sexual experience have in common). So this appendix is designed to enable the virginal newcomer to hex to effectively penetrate this unnecessarily mysterious subject and come to a full understanding of its seemingly arcane symbol system. If you work carefully through the next few pages, satisfaction is guaranteed.

Funny Number Systems

You already know that the number system we use day in and day out is based on the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. This gives us only 10 digits (a total which corresponds nicely to the number of fingers we have!) with which to construct the nearly infinite range of numbers we have to deal with in our many activities.

Stop to reflect for a moment on how this miraculous process works. If we're counting up a large amount of something, we start out by saying "one, two, three, ...", and so on, until we get to 9. Here we have already run out of our allotted 10 symbols for enumerating things. So what do we do if we need to keep on counting? We go right back to '0' and start over again. Only this time we stick a '1' in front of the numbers we're using. This gives us 11, 12, 13, and so on, up to 19.

To keep the count going from there, we now stick a 2 in front and keep right on using the same 10 symbols. This process repeats itself over and over again in a loop until we reach 99. When that happens, of course, we've used up all our allotted symbols twice. But this doesn't stop us at all. We just stick a third digit in front and keep right on counting: 100, 101, 102, etc. So powerful is this seemingly obvious process that a British mathematician, Alan Turing, showed that any machine that can do it can compute just about any function that we humans might want to carry out. Only a few years after he published this startling research, the first working example of that device we have all come to know and worship - the digital computer - was built.

Why has this kindergarten exercise been the subject of deep mathematical research, not to mention some considerable belaboring by me in the last 3 paragraphs? Because it is so remarkably flexible - so flexible, in fact, that it can be used with any number system, no matter how few or how many symbols it contains.

Let's say for example that homo sapiens had evolved with 8 fingers on each hand, leaving the human race a total of 16 to count with, instead of the familiar ten that actually grace the perimeters of our palms. Would this have meant that primitive man would have been unable to determine, say, how many head of buffalo (or wives, perhaps) that he could call his own? Not a bit of it! We don't have to plow our way through Alan Turing's paper to see that our Neanderthal forebearers (and we ourselves) could still do any amount of counting necessary for survival.

In fact, we would need nothing more than a set of 16 symbols upon which to repeat endlessly the magical loop that I described above. In fact, we already know of 10 symbols from our everyday "decimal" number system which would be likely candidates: the digits from 0 to 9. We couldn't really use the numbers from 10 - 15 for the six symbols we're missing, though. They consist of 2 digits, a characteristic which makes them unsuitable for the counting loop I've been discussing.

One solution would be to use the letters from A - F when we ran out of digits while counting something. This would give us a total of 16 counting symbols:

0 1 2 3 4 5 6 7 8 9 A B C D E F

This time, when we got to '9' in our count, we could keep going a little longer by chanting "nine, ay, bee, see,...", and so on until we got to "ef". At this point we would be in the same predicament we encountered with our "decimal" number system - we would have run out of symbols. Knowing about our special loop would save the day, however. We would simply go back to zero and count up again - just like I did with the decimal numbers. So we'd put a '1' in front of the zero and call it "ten". Then off we'd go with "eleven, twelve", etc. We would be doing exactly the same thing as before, except that 10 would come after 'F' instead of after '9'. When we got to '19', we could continue: '1A', '1B', '1C', ... right up to '1F'. After that we would start with '20', just like before.

If we continue this process long enough, we would get to 'FF', the last of our two-digit numerals. What has happened here is that we have counted up to 'F' repeatedly - just like before when we counted up to '9' repeatedly to get '99'. So just like '99' in our previous system, 'FF' would be followed by '100', and so the process would continue as long as we wished.

What our mythical prehistoric race - with their 16 fingers scraping the earth as they shuffled through their existence - would have left us is a sixteen-based or "hexadecimal" system of numbers. Much to the consternation of beginning computerists, whose counting habits evolved from their 10-fingered ancestors, machine code programmers positively revel in this number system. Does this mean that they were really born with 16 fingers? Since the seemingly incomprehensible verbal output from many such programmers insures that the beginner will never get close enough to shake hands with them, he or she is unlikely ever to find out.

Making It With Binary

Let's put aside the question of our many-fingered friends for the moment and engage in another piece of fantasy. Let's suppose this time that evolution had left us with a cruel endowment - two hands, but only one finger! Surely this would have relegated the human race to a numberless existence - no arithmetic, algebra, calculus, or the remarkable mathematical accomplishments of an Einstein or a Godel.

What could you possibly do with one finger if you wanted to count up a large amount of something (no rude answers please)? In fact, there are two ways you could use your solitary appendage. First, you could close it into a fist (if that's the right word), and second, you could stick it up in the air. I think you can already see that these two movements alone might form the basis of a useful counting procedure. Just as in the previous two systems, let's let our symbols start with '0'. We'll signify this with a closed fist. Then we could let a finger thrust into the air stand for a '1'. This means we could begin by pointing at the ceiling and intoning "one". But we've already run out of fingers, so our number system can only contain the symbols:

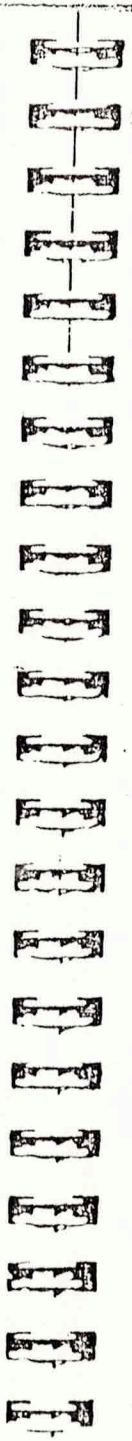
0 1

Let's see what would happen if we applied our counting loop to this number system. As usual, we start by counting "one", but already find that we've run out of single digits. The way we handled this in both our previous examples was to go back and start with '0', this time putting a '1' in front of it. If we do that now, we end up by folding our finger against our palm and saying "ten", writing it as "10". After ten we can thrust our finger in the air again and sing out "eleven", writing "11".

So far we've used exactly the same rule of thumb (or finger) that formed the basis of our counting in the previous two systems: Whenever you use up all the single digits you have, go back and use them once more. Each time you start with zero again, increment the number in front of the digits. In decimal this would enable us to count by '1's' from '10' to '20' to '30', and so on. The result of the same process with only '1's' and '0's' would give us:

0 1 10 11

Notice that here we've already run out of numerals that can be formed with one or two digits. This is exactly the same thing that happened to us when we got to '99' in decimal or 'FF' in hex. In other words, just as '99' is obtained by counting up to '9' repeatedly, here an '11' the result of counting up to '1' repeatedly (well, actually only twice). The only difference between the 3 systems is that the fewer the one-digit counting symbols you have, the quicker you run out of two-digit numbers.



What we did before when we exhausted all our 2-digit combinations was to say "one hundred", and to begin to form numerals by shoving a fresh digit in front of them. We can do exactly the same again and get:

0 1 10 11 100 101 110 111

Now we've run out of 3-digit numbers, so we continue the process by saying "one thousand", just as we do when we reach '999', the last 3-digit number in decimal. We can thus continue from 111:

1000 1001 1010 1011 1100 1101 1110 1111

- and keep on indefinitely. Of course, the quantity of digits in our numerals would increase markedly as we kept on counting. However, this would probably be a trivial consideration to a person with only one finger.

So even using only 2 symbols in this "binary" number system, you can count up to any amount you like. And our old friend, Alan Turing, showed us that if you can count, you can represent and manipulate useful data. However, just one person waggling a single finger up and down would take a long time to process most kinds of information that we might be interested in. So one way to speed things up would be to form a large population of such people into small groups. By arranging a group's collective fingers into different patterns of "up" or "down", it could represent a single unit of information - like a letter or a number, for example.

The size of the groups may be somewhat arbitrary, but let's just say that each contains 8 members of our finger-flinging population. This means that each group could signal a binary number, a raised finger representing a '1', and a lowered finger signifying a '0'. The numbers possible to form in this manner would range from 0, with each member's finger pressed against his palm, to 11111111, with all 8 members' fingers raised. In decimal terms, one group could then represent the numbers from 0 to 255 (binary 11111111, hex FF).

This system also makes possible group-to-group communication which could take place by one group shouting its number to another group. However, since things would get pretty noisy under this system, it might be more orderly to designate a few special groups as registrars (or "registers", perhaps) who would run to one group, look at its finger pattern, reproduce it on their own fingers, and finally, race along to another group to show it to them.

These possibilities are extremely fortunate, since your computer's insides are something like a city populated by this mythical race. Although looking inside your Apple won't reveal a vast field of waggling fingers, your machine does contain thousands of microscopic switch-like devices which can only adopt

one of two states - open and shut.

If one of these tiny switches (properly called a "flip-flop") is open it, signals a '0'; if it's closed, it represents a '1'. Each switch's output is popularly referred to as a "bit". Since 256 bit-patterns can be formed from just 8 bits, it is possible to use them to represent all numbers, letters, punctuation, machine instructions, plus a number of other specialized symbols. Accordingly, the Apple and most other microcomputers to handle information using groups of 8 bits. These are called "bytes".

I guess you can see why the binary number system is pretty important in computing. But where does hexadecimal come into the picture? The most important reason for bringing in hex is that although your Apple loves binary, experience shows that your brain can't cope very well at all with all those confusing arrays of '1's' and '0's'. In fact working in binary proved to be a nightmare for the earlier pioneers of computing.

This means that we need an easier number system to cope with system of numbering, since all data, instructions, and even memory locations are represented in the machine in binary. Now you might think that decimal is the obvious choice. It isn't, however and to see why not, have a look at the following table which compares binary to hexadecimal.

binary:	0	1	10	11	100	101	110	111	1000	1001	1010
hex:	0	1	2	3	4	5	6	7	8	9	A
binary:	1011	1100	1101	1110	1111						
hex:	B	C	D	E	F						

Notice that all the possible patterns of 4 binary bits are exhausted at exactly the same that we run out of single-digit hex numerals! This has to do with the fact that 16 = 2 raised to the power of 4. Don't worry to much about this, but instead think of how easy this fact makes conversion between the two systems.

In fact, all you have to do to convert any binary number to hex, or vice versa, is use the above table. For example, let's convert 11001001 into hex.

Step1: Split the binary number into 4-bit groups.

1100 1001

Step 2: Use the table to translate each group of 4 bits into its hex equivalent.

1100 = C 1001 = 9

Step 3: Shove the two hex numbers together.

C9

The answer is C9. To show that it is a hex number, a '$' is always placed in front, giving us $C9. You'll see me using this convention extensively throughout the book.

To get from hex to binary is equally simple. You just reverse the process. For example, to translate $4A, we use the chart above to find that 4 = 100 and A = 1010. Thus $4A = 1001010. By the way, because every bit in an 8-bit byte has to be accounted for, 1001010 should be written 01001010. This first zero is called a leading zero; it doesn't alter 1001010's value any more than, say, the leading '0' in '09' (i.e., 09 = 9).

No such simple conversion procedure is available for translating between binary and decimal, so hex has got to be the choice. Once you get used to it though, it's really a joy to use. Everything in computing is organized in '16's', making it easy for you to see just where you are. I won't go into examples here, but suffice it to see that you'll soon appreciate hexadecimal as you learn more about computing.

Some Machine Hex and Binary Jargon

When a machine-code programmer is dealing with data he often is concerned about which bits are set (i.e., which equal '1') and which are clear (equal to '0'). Of particular importance are the high bit (alias the sign bit, the high order bit, bit 7) and the low bit (the low order bit, bit 0). The 8 bits in a byte are numbered as follows.

bit number	7	6	5	4	3	2	1	0
	|	|	|	|	|	|	|	|
an 8-bit byte	0	1	0	0	1	0	0	1

You can see that the high bit is clear and the low bit is set. All bytes passing into and out of your Apple must have the high bit set. So when you type in data from the keyboard, load files from the disk, or send information to the screen, this rule will be in effect.

This means that the Apple recognizes two types of ASCII - the normal version, or "low" ASCII, in which each character's high bit is clear (e.g. $41 = binary 0100 0001 = 'A'), and "high" or "screen" ASCII which is exactly the same, except that the high bit is set (e.g. $C1 = 1100 0001 = 'A'). The CIA utilities make it convenient for you to work with both.

Backward Addresses

One quirk displayed by the 6502 (in common with certain other microprocessors) is that it can't properly interpret an memory address unless its two bytes are reversed. For example,

If you wanted to instruct the 6502 to deal with the address, $FC58, you would have to specify it as '58 FC'.

Exclusive ORing

A final item you need to know about is a machine-code instruction which comes up on a number of occasions when you are working with a disk. It is called "Exclusive OR, abbreviated EOR. Now you already know about the 4 arithmetic operations, +, -, /, and x. Each of these 4 can be performed on binary and hex, as well as decimal. The way to think about EOR is as a 5th arithmetic operation that can only be performed on the binary translation of a hex or decimal number. This [illegible] because it involves only the individual bits themselves. Here's how it works:

0 EOR 0 = 0 0 EOR 1 = 1 1 EOR 0 = 1 1 EOR 1 = 0

In a nutshell, if two bits are the same, their EOR product is 0; if they're different, it's 1. In the four cases above I only used one bit. Here's an example of what would happen if you applied these four rules to all 8 bits in a byte.

$41 EOR $73 = $32

$41 = 0100 0001 $73 = 0111 0011

```
     0100 0001
EOR  0111 0011
     ---------
     0011 0010
```

HEX - BINARY - DECIMAL CONVERSION CHART

On the next page, I've set out a chart you can use to convert hex to binary and vice-versa. It shows the first 16 values, and includes some other commonly used higher values. All decimal equivalents are also given.

Hex	Binary	Decimal
0	0000 0000	0
1	0000 0001	1
2	0000 0010	2
3	0000 0011	3
4	0000 0100	4
5	0000 0101	5
6	[illegible]	6
7	0000 0111	7
8	0000 1000	8
9	[illegible]	[illegible]
A	0000 1010	10
B	0000 1011	11
C	0000 1100	12
D	0000 1101	13
E	0000 1110	14
F	0000 1111	15

Hex	Binary	Decimal
10	0001 0000	16
20	0010 0000	32
3F	0011 1111	63
40	0100 0000	64
7F	0111 1111	127
80	1000 0000	[illegible]
AA	1010 1010	170
C0	1100 0000	192
E8	1110 1000	[illegible]
FE	1111 1110	254
FF	1111 1111	255