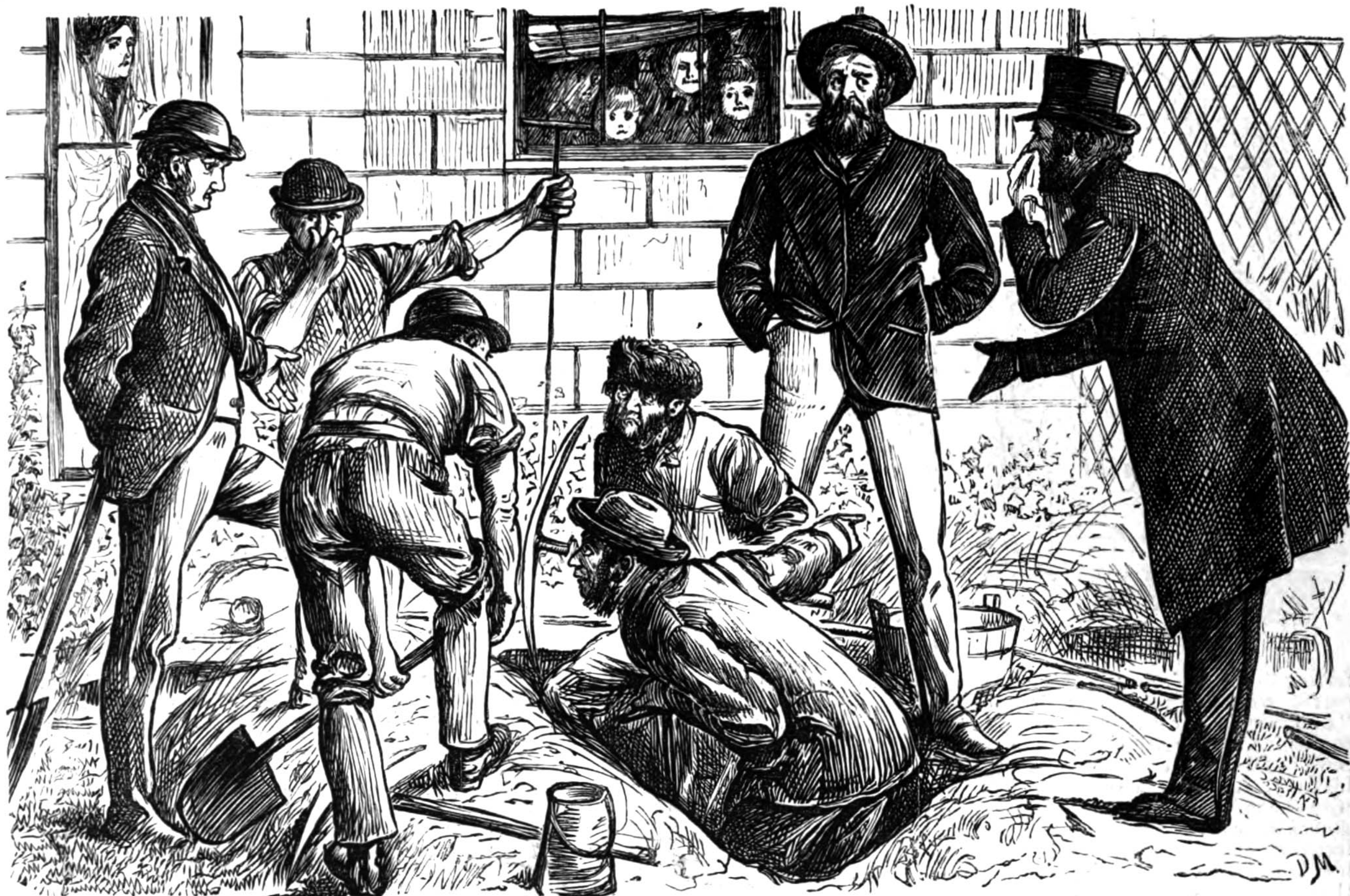
Beneath Apple DOS and Beneath Apple ProDOS 2020

By Don Worth and Pieter Lechner

Beneath Apple DOS
Beneath Apple ProDOS



Are you sure you want to go Beneath Apple DOS?

Original Copyright © 1981 by Quality Software.

Put into the Public Domain by Don Worth.
This edition edited and produced by Snape Publishing.

The word APPLE and the Apple logo are registered trademarks of Apple Computer, Inc. Apple Computer, Inc. was not in any way involved in the writing or other preparation of this manual, nor were the facts presented here reviewed for accuracy by that company. Use of the term APPLE should not be construed to represent any endorsement, official or otherwise, by Apple Computer, Inc.

Disclaimer: Quality Software and Snape Publishing shall have no liability or responsibility to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual or its use, including, but not limited to, any interruption in service, loss of business or anticipatory profits, or consequential damages resulting from the use of this product.

The body typeface is Garamond, while the header and subheads are Helvetica Neue LT Std. The program listings are in Courier 10 Pitch BT. The figure captions are Futura Std. The images are from the *Punch* series of political cartoons from the late 1800s. The figures and tables are drawn in Adobe Photoshop® and the book layout was prepared in Adobe InDesign®.

Contents

Beneath Apple DOS 1

Beneath Apple ProDOS 151

Miscellanea and Glossary. 389

Early Disk Drive

From *Beneath Apple DOS*:

Thanks go to Vic Tolomei for his assistance in dissecting DOS 3.1 and to Lou Rivas for his patient proofreading. Thanks also to my wife Carley for putting up with the clackety clack of my Diablo long into the night.

Don D. Worth

Thanks to the people at Computerland of South Bay (California) who lent me support both of their time and equipment, and special thanks to John Gattuso, whose encouragement helped me to complete the task.

Pieter M. Lechner

From *Beneath Apple ProDOS*:

This book is dedicated to my sister, Betsy, who said she had room on her bookshelf for another one of my books.

Don D. Worth

This book is dedicated to my Father and Mother, with a deep sense of appreciation and gratitude.

Pieter M. Lechner

The authors wish to thank Quality Software for their able assistance in producing this book. Special thanks to Bob Christiansen, Bob Pierce, Kathy Schmidt, George Garcia, Vic Grenrock, and Jeff Weinstein for their unique and special contributions.

From the Editor:

This book combines two works from the 1980s: *Beneath Apple DOS* and *Beneath Apple ProDOS*. I have tried to keep as much of the original text as possible, but some of the figures and tables have been re-drawn to either make them easier to read or easier to understand (or both).

Also, some of the information is duplicated in both books, but I have kept the duplicates so if you're studying one operating system, you don't need to reference the other. The one main item I combined were the glossaries, since many of the same terms are in both. It is the last chapter in the book.

My personal history with *Beneath Apple DOS* goes back to high school. I remember someone brought the book into the computer room and we all marvelled at the detail and technical savvy in it. This was an explicit look at DOS that went far beyond anything before. I did go home and print out a disassembly of DOS 3.3 and go through each section, labeling the subroutines as listed in the book.

I still think reading code from others is one of the best ways to learn a programming language and I learned quite a bit from DOS 3.3. I learned enough that I eventually wrote my own DOS, using just the RWTS routines (I never did understand completely how it worked!) using my own routines for the rest of it. At the time, our school had a timesharing HP 2000F mainframe, and I used some of the syntax from that for my own DOS. Sadly, my version of DOS has been lost to history.

If this is your first time seeing this book, I hope you experience the joy so many have over the years. If you had this book previously, I hope I've kept the feel of the original book, while updating it for a new generation.

I'd like to specially thank Sean McNamara for his proofreading work on this book. Thank you, Sean!

— John Snape
Editor

Be careful using RWTS

Beneath Apple DOS

Contents

CHAPTER 1: INTRODUCTION 5

CHAPTER 2: THE EVOLUTION OF DOS 7
- DOS 3 - 29 JUNE 1978 7
- DOS 3.1 - 20 JULY 1978 7
- DOS 3.2 - 16 FEBRUARY 1979 7
- DOS 3.2.1 - 31 JULY 1979 8
- DOS 3.3 - 25 AUGUST 1980 9
- DOS 3.3 - 1 JANUARY 1983 .... 1Ø

CHAPTER 3: DISKETTE FORMATTING 11
- TRACK FORMATTING 13
- DATA FIELD ENCODING 21
- SECTOR INTERLEAVING 25

CHAPTER 4: DISKETTE ORGANIZATION 31
- DISKETTE SPACE ALLOCATION 32
- HOW DOS IS STORED ON THE DISK 32
- THE VTOC 33
- VOLUME TABLE OF CONTENTS (VTOC) FORMAT 33
- BIT MAPS OF FREE SECTORS ON A GIVEN TRACK 34
- THE CATALOG 36
- CATALOG SECTOR FORMAT 37
- FILE DESCRIPTIVE ENTRY FORMAT 37
- THE TRACK/SECTOR LIST 38
- TRACK/SECTOR LIST FORMAT 39
- TEXT FILES 42
- BINARY FILES 43
- APPLESOFT AND INTEGER FILES 43
- OTHER FILE TYPES (S, R, NEW A, NEW B) 43
- EMERGENCY REPAIRS 45

CHAPTER 5: THE STRUCTURE OF DOS.................................49
DOS MEMORY USE 49
THE DOS VECTORS IN PAGE 3 ....5Ø
DOS VECTOR TABLE ($3D0-$3FF).......................................51
WHAT HAPPENS DURING BOOTING..52

CHAPTER 6: USING DOS FROM ASSEMBLY LANGUAGE..........57
CAVEAT..57
DIRECT USE OF DISK DRIVE...57
STEPPER PHASE OFF/ON...58
MOTOR OFF/ON...59
ENGAGE DRIVE 1/2...59
READ A BYTE..59
SENSE WRITE PROTECT 59
LOAD AND WRITE A BYTE 59
CALLING READ/WRITE TRACK/SECTOR (RWTS) 6Ø
INPUT/OUTPUT CONTROL BLOCK - GENERAL FORMAT 6Ø
RWTS IOB BY CALL TYPE 61
CALLING THE DOS FILE MANAGER 63
FILE MANAGER PARAMETER LIST - GENERAL FORMAT........................65
FILE MANAGER PARAMETER LIST BY CALL TYPE............................66
DOS BUFFERS 7Ø
DOS BUFFER FORMAT 7Ø
THE FILE MANAGER WORKAREA...7Ø
FILE MANAGER WORKAREA FORMAT 71
COMMON ALGORITHMS...72
LOCATE A FREE DOS BUFFER 72
WHICH VERSION OF DOS IS ACTIVE? 72
IS DOS IN THE MACHINE?...73
WHICH BASIC IS SELECTED? 73
SEE IF A BASIC PROGRAM IS IN EXECUTION...........................73
PROCESS DOS ERROR FROM ASSEMBLY LANGUAGE.........................74

CHAPTER 7: CUSTOMIZING DOS...75
SLAVE VS MASTER PATCHING..75
AVOIDING RELOAD OF LANGUAGE CARD.................................76
INSERTING A PROGRAM BETWEEN DOS AND ITS BUFFERS 77
BRUN OR EXEC THE HELLO FILE......................................77
REMOVING THE PAUSE DURING A LONG CATALOG.........................78
WRITING DOS IN MEMORY TO A DISKETTE..............................78

CHAPTER 8: DOS PROGRAM LOGIC 79
DISK II CONTROLLER CARD ROM - BOOT 0 79
FIRST RAM BOOTSTRAP LOADER - BOOT 1 .... 8Ø
DOS 3.3 MAIN ROUTINES 81
DOS ZERO PAGE USAGE 115

APPENDIX A: EXAMPLE PROGRAMS 117
STORING THE PROGRAMS ON DISKETTE 118
DUMP – TRACK DUMP UTILITY 12Ø
ZAP – DISK UPDATE UTILITY 124
INIT – REFORMAT A SINGLE TRACK 128
FTS – FIND T/S LISTS UTILITY 132
COPY – CONVERT FILES 137

APPENDIX B: DISK PROTECTION SCHEMES 145

It says here that file starts on Track $18, Sector $ØF

Chapter 1
Introduction

Beneath Apple DOS is intended to serve as a companion to Apple's *DOS Manual*, providing additional information for the advanced programmer or the novice Apple user who wants to know more about the structure of diskettes. It is not the intent of this manual to replace the documentation provided by Apple Computer Inc. Although, for the sake of continuity, some of the material covered in the Apple manual is also covered here, it will be assumed that the reader is reasonably familiar with the contents of the *DOS Manual*. Since all chapters presented here may not be of use to each Apple owner, each has been written to stand on its own.

The information presented here is a result of intensive disassembly and annotation of various versions of DOS by the authors and by other experienced systems programmers. It also draws from application notes, articles, and discussions with knowledgeable people. This manual was not prepared with the assistance of Apple Computer Inc. Although no guarantee can be made concerning the accuracy of the information presented here, all of the material included in *Beneath Apple DOS* has been thoroughly researched and tested.

There were several reasons for writing *Beneath Apple DOS*:

- To show direct assembly language access to DOS.
- To help you to fix clobbered diskettes.
- To correct errors and omissions in the Apple documentation.
- To allow you to customize DOS to fit your needs.
- To provide complete information on diskette formatting

When Apple Computer Inc. introduced its Disk Operating System (DOS) version 3 in 1978 to support the new Disk II drive, very little documentation was provided. Later, when DOS 3.2 was released, a 178-page instructional and reference manual became available covering the use of DOS from BASIC in depth and even touched upon some of the internal workings of DOS. With the advent of DOS 3.3, the old 3.2 manual was updated but the body of information in it remained essentially intact. Beyond these Apple manuals,

there have been no significant additions to the documentation on DOS, apart from a few articles in Apple user group magazines and newsletters. This manual takes up where the *Disk Operating System Manual* leaves off.

Throughout this manual, discussion centers primarily on DOS version 3.3. The reasons for this are that 3.3 was the most recent release of DOS at the time of this writing and that it differs less from DOS 3.2 than one would imagine. Wherever there is a major difference between the various DOS releases in a given topic, each release will be covered.

In addition to the DOS dependent information provided, many of the discussions also apply to other operating systems on the Apple II and Apple III. For example, disk formatting at the track and sector level is, for the most part, the same.

one week

two years

Programmer Age Progression

Chapter 2
The Evolution of DOS

Since its introduction, Apple DOS has gone through three major versions. All of these versions look very much the same on the surface. All commands supported by DOS 3.3 are also supported in 3.2 and 3.1. The need for additional versions has been more to fix errors in DOS and to make minor enhancements than to provide additional functionality. Only DOS 3.3 has offered any major improvement in function; an increase in the number of sectors that will fit on a track from 13 to 16.

DOS 3 - 29 JUNE 1978
DOS 3.1 - 20 JULY 1978

The first release of DOS was apparently a victim of a rush at Apple to introduce the Disk II. As such, it had a number of bugs. With the movement towards the Apple II Plus and the introduction of the Autostart ROM, a new release was needed.

DOS 3.2 - 16 FEBRUARY 1979

Although DOS 3.2 embodied more changes from its predecessor than any other release of DOS, 90% of the basic structure of DOS 3.1 was retained. The major differences between DOS 3.1 and 3.2 and later versions of DOS are listed below:

- `NOMON C,I,O` is the initial default under DOS 3.2. `MON C,I,O` was the default under DOS 3.1.
- Input prompts (`>` , `]` , `*`) are echoed when `MON O` is in effect, not under `MON I` as was the case under 3.1.
- When a DOS command was entered from the keyboard, DOS executed it and then passed a blank followed by a carriage return to BASIC under 3.1. Under 3.2 only a carriage return is passed.

- Under 3.2, certain commands may not be entered from the keyboard but may only be used within a BASIC program (`READ`, `WRITE`, `POSITION`, `OPEN`, `APPEND`).
- Under 3.2, when `LOAD`ing an Applesoft program, DOS automatically converts from Applesoft ROM format to Applesoft RAM format if the RAM version of BASIC is in use and vice versa.
- DOS 3.1 could not read lower case characters from a text file; DOS 3.2 can.
- Some DOS commands are allowed to create a new file, others will not. Under DOS 3.1, any reference to a file that didn't exist, caused it to be created. This forced DOS 3.1 to then delete it if a new file was not desired. (`LOAD XYZ` under 3.1 if `XYZ` did not exist, created `XYZ`, deleted `XYZ`, and then printed the file not found error message.) Under 3.2, `OPEN` is allowed to create a file if one does not exist, but `LOAD` may not.
- Under 3.1, exiting to the monitor required that the monitor status register location ($48) be set to zero before reentering DOS. Under DOS 3.2 this is no longer necessary.
- The Read/Write-Track/Sector (RWTS) section of DOS disables interrupts while it is executing. Under 3.1, RWTS could be interrupted by a peripheral while writing to a disk, destroying the disk.
- The default for the B (byte offset) keyword is 0 under 3.2.
- DOS was reassembled for 3.2 causing most of its interesting locations and routines to move slightly. This played havoc with user programs and utilities which had DOS addresses built into them.
- Additional file types (beyond T, I, A, and B) are defined within DOS 3.2, although no commands yet support them. The new types are S, R, a new A, and a new B. R has subsequently been used by the DOS Toolkit for relocatable object module assembler files. At present, no other use is made of these extra file types.
- Support was added under 3.2 for the Autostart ROM.
- All files open when a disk full condition occurs are closed by DOS 3.2.
- As with each new release of DOS, several new programs were added to the master diskette for 3.2. Among these was `UPDATE 3.2`, a replacement for `MASTER CREATE`, the utility for creating master diskettes. `UPDATE 3.2` converts a slave into a master and allows the `HELLO` file to be renamed.

DOS 3.2.1 - 31 JULY 1979

DOS 3.2.1 was essentially a "maintenance release" of DOS 3.2. Minor patches were made to RWTS and the `COPY` program to correct a timing problem when a dual drive copy was done. Additional delays were added following a switch between drives.

DOS 3.3 - 25 AUGUST 1980

Introduced in mid 1980 as a hardware/software upgrade from DOS 3.2.1, the DOS 3.3 package includes new bootstrap and state ROM chips for the disk controller card which provide the capability to format, read, and write a diskette with 16 sectors. (These ROMs are the same ones used with the Language System.) This improvement represents almost a 25% increase in available disk space over the old 13 sector format. Also included in the 3.3 package is an updated version of the DOS manual, a BASICS diskette (for 13 sector boots), and a master diskette. Although the RWTS portion of DOS was almost totally rewritten, the rest of DOS was not reassembled and only received a few patches:

- The initial DOS bootstrap loader was moved to $800 under 3.3. It was at $300 under 3.2. In addition, as stored on the diskette (track 0 sector 0) it is nibbilized in the same way as all other sectors under 3.3.
- A bug in `APPEND` which caused it to position improperly if the file was a multiple of 256 bytes long was fixed under 3.3.
- A `VERIFY` command is internally executed after every `SAVE` or `BSAVE` under 3.3.
- All 4 bytes are used in the Volume Table Of Contents (VTOC) free sector bit map when keeping track of free sectors. This allows DOS to handle up to 32 sectors per track. Of course, RWTS will only handle 16 sectors due to hardware limitations.
- If a Language Card is present, DOS stores a zero on it at `$EØØØ` during bootstrap to force the `HELLO` program on the master diskette to reload BASIC.
- DOS is read into memory from the top down (backwards) under 3.3 rather than the bottom up. Its image is still stored in the same order on the diskette (tracks 0, 1, and 2), however.
- Additional programs added to the master diskette under 3.3 include `FID`, a generalized file utility which allows individual files or groups of files

to be copied, `MUFFIN`, a conversion copy routine to allow 3.2 files to be moved to 16 sector 3.3 diskettes, `BOOT 13`, a program which will boot a 13 sector diskette, and a new `COPY` program which will also support single drive copies.

- Under 3.2, speed differences in some drives prevented their use together with the DOS `COPY` program. Because the `COPY` program was rewritten under 3.3, that restriction no longer applies.

DOS 3.3 - 1 JANUARY 1983

This "maintenance release" of DOS was introduced with the Apple IIe computer. It contains a few minor patches and no additional function.

- A patch was introduced in DOS 3.3 to fix a bug in `APPEND` processing. This patch also had bugs. Additional patches were added to (hopefully) correct this problem.
- An error in the `POSITION` calculation for large files is corrected in this release.
- A few instructions were added to properly support the 80 column display on the Apple IIe.
- The System Master diskette contains some different files. Notably, a fast loader for the language card and all of the example programs have been moved to a separate diskette.

Chapter 3
Diskette Formatting

Apple Computer's excellent manual on the Disk Operating System (DOS) provides only very basic information about how diskettes are formatted. This chapter will explain in detail how information is structured on a diskette. The first section will contain a brief introduction to the hardware, and may be skipped by those already familiar with the DOS manual.

For system housekeeping, DOS divides diskettes into tracks and sectors. This is done during the initialization process. A track is a physically-defined circular path which is concentric with the hole in the center of the diskette. Each track is identified by its distance from the center of the disk. Similar to a phonograph stylus, the read/write head of the disk drive may be positioned over any given track. The tracks are similar to the grooves in a record, but they are not connected in a spiral. Much like playing a record, the diskette is spun at a constant speed while the data is read from or written to its surface with the read/write head. Apple formats its diskettes into 35 tracks. They are numbered from 0 to 34, track 0 being the outermost track and track 34 the innermost. Figure 3.1 illustrates the concept of tracks, although they are invisible to the eye on a real diskette.

It should be pointed out, for the sake of accuracy, that the disk arm can position itself over 70 "phases". To move the arm past one track to the next,

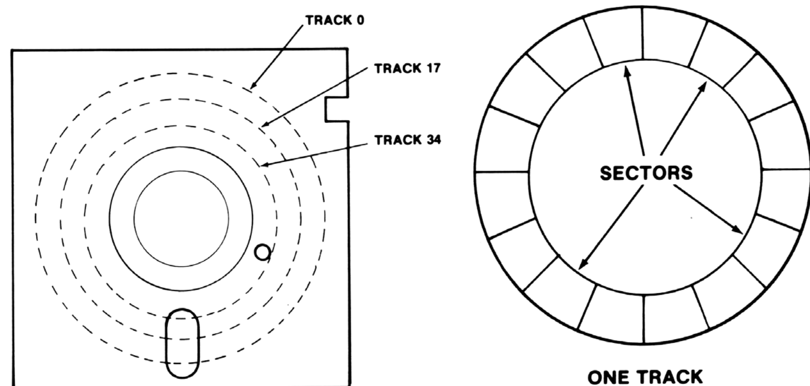


Figure 3.1 Tracks and Sectors

two phases of the stepper motor, which moves the arm, must be cycled. This implies that data might be stored on 70 tracks, rather than 35. Unfortunately, the resolution of the read/write head and the accuracy of the stepper motor are such, that attempts to use these phantom "half" tracks create so much cross-talk that data is lost or overwritten. Although the standard DOS uses only even phases, some protected disks use odd phases or combinations of the two, provided that no two tracks are closer than two phases from one another. See Appendix B for more information on protection schemes.

A sector is a subdivision of a track. It is the smallest unit of "updatable" data on the diskette. DOS generally reads or writes data a sector at a time. This is to avoid using a large chunk of memory as a buffer to read or write an entire track. Apple DOS has used two different track formats—one divides the track into 13 sectors, the other, into 16 sectors. The sectoring does not use the index hole, provided on most diskettes, to locate the first sector of the track. The implication is that the software must be able to locate any given track and sector with no help from the hardware. This scheme, known as "soft sectoring", takes a little more space for storage but allows flexibility, as evidenced by the recent change from 13 sectors to the present 16 sectors per track. The following table catagorizes the amount of data stored on a diskette under both 13 and 16 sector formats.

```
                DISK ORGANIZATION

TRACKS
 All DOS versions................35
SECTORS PER TRACK
 DOS 3.2.1 and earlier...........13
 DOS 3.3.........................16
SECTORS PER DISKETTE
 DOS 3.2.1 and earlier..........455
 DOS 3.3........................560
BYTES PER SECTOR
 All DOS versions...............256
BYTES PER DISKETTE
 DOS 3.2.1 and earlier.......116480
 DOS 3.3.....................143360
USABLE* SECTORS FOR DATA STORAGE
 DOS 3.2.1 and earlier..........403
 DOS 3.3........................496
USABLE* BYTES PER DISKETTE
 DOS 3.2.1 and earlier.......103168
 DOS 3.3.....................126976

* Excludes DOS, VTOC, and CATALOG
```

TRACK FORMATTING

Up to this point we have broken down the structure of data to the track and sector level. To better understand how data is stored and retrieved, we will start at the bottom and work up.

As this manual is primarily concerned with software, no attempt will be made to deal with the specifics of the hardware. For example, while in fact data is stored as a continuous stream of analog signals, we will deal with discrete digital data, i.e., a 0 or a 1. We recognize that the hardware converts analog data to digital data but how this is accomplished is beyond the scope of this manual.

Data bits are recorded on the diskette in precise intervals. The hardware recognizes each of these intervals as either a 0 or a 1. We will define these intervals to be "bit cells". A bit cell can be thought of as the distance the diskette moves in four machine cycles, which is about four microseonds. Using this representation, data written to and read back from the diskette takes the form shown in Figure 3.2. The data pattern shown represents a binary value of 101.

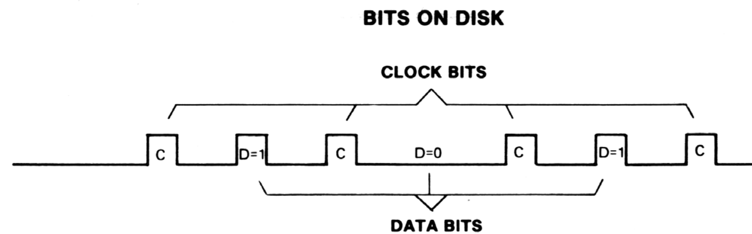


Figure 3.2 Bits on Diskette

A byte as recorded on the disk consists of eight (8) consecutive bit cells. The most significant bit cell is usually referred to as bit cell 7 and the least significant bit cell would be bit cell 0. When reference is made to a specific data bit (i.e. data bit 5), it is with respect to the corresponding bit cell (bit cell 5). Data is written and read serially, one bit at a time. Thus, during a write operation, bit cell 7 of each byte would be written first, with bit cell 0 being written last. Correspondingly, when data is being read back from the diskette, bit cell 7

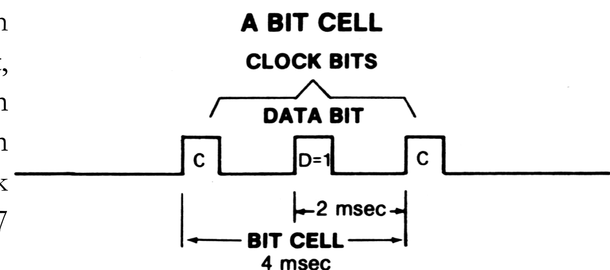


Figure 3.3 A Bit Cell

is read first and bit cell 0 is read last. Figure 3.4 illustrates the relationship of the bits within a byte.

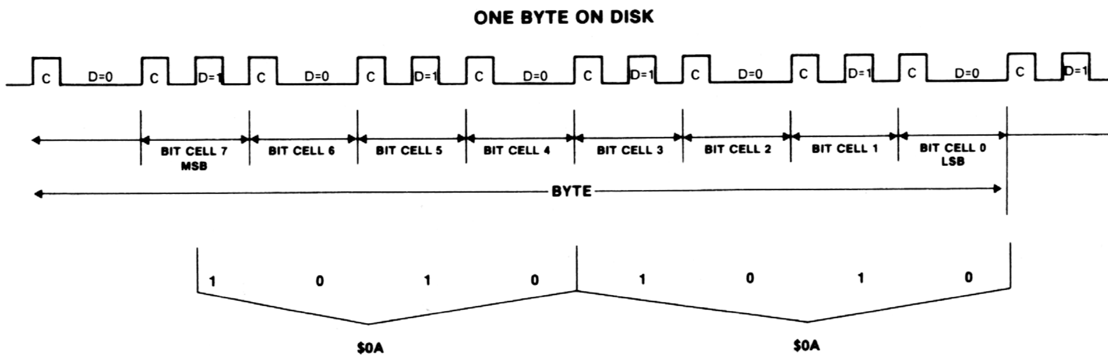


Figure 3.4 One Byte on Diskette

To graphically show how bits are stored and retrieved, we must take certain liberties. The diagrams are a representation of what functionally occurs within the disk drive. For the purposes of our presentation, the hardware interface to the diskette will be represented as an eight bit "data latch". While the hardware involves considerably more complication, from a software standpoint it is reasonable to use the data latch, as it accurately embodies the function of data flow to and from the diskette.

Figure 3.5 shows the three bits, 101, being read from the diskette data stream into the data latch. Of course another five bits would be read to fill the latch. As can be seen, the data is separated from the clock bits. This task is done by the hardware and is shown more for accuracy than for its importance to our discussion.

Writing data can be depicted in much the same way (see Figure 3.6). The clock bits which were separated from the data must now be interleaved with the data as it is written. It should be noted that, while in write mode, zeros are being brought into the data latch to replace the data being written. It is the task of the software to make sure that the latch is loaded and instructed to write in 32-cycle intervals. If not, zero bits will continue to be written every four cycles, which is, in fact, exactly how self-sync bytes are created. Self-sync bytes will be covered in detail shortly.

A "field" is made up of a group of consecutive bytes. The number of bytes varies, depending upon the nature of the field. The two types of fields present on a diskette are the Address Field and the Data Field. They are similar in that they both contain a prologue, a data area, a checksum, and an epilogue. Each field on a track is separated from adjacent fields by a number of bytes. These

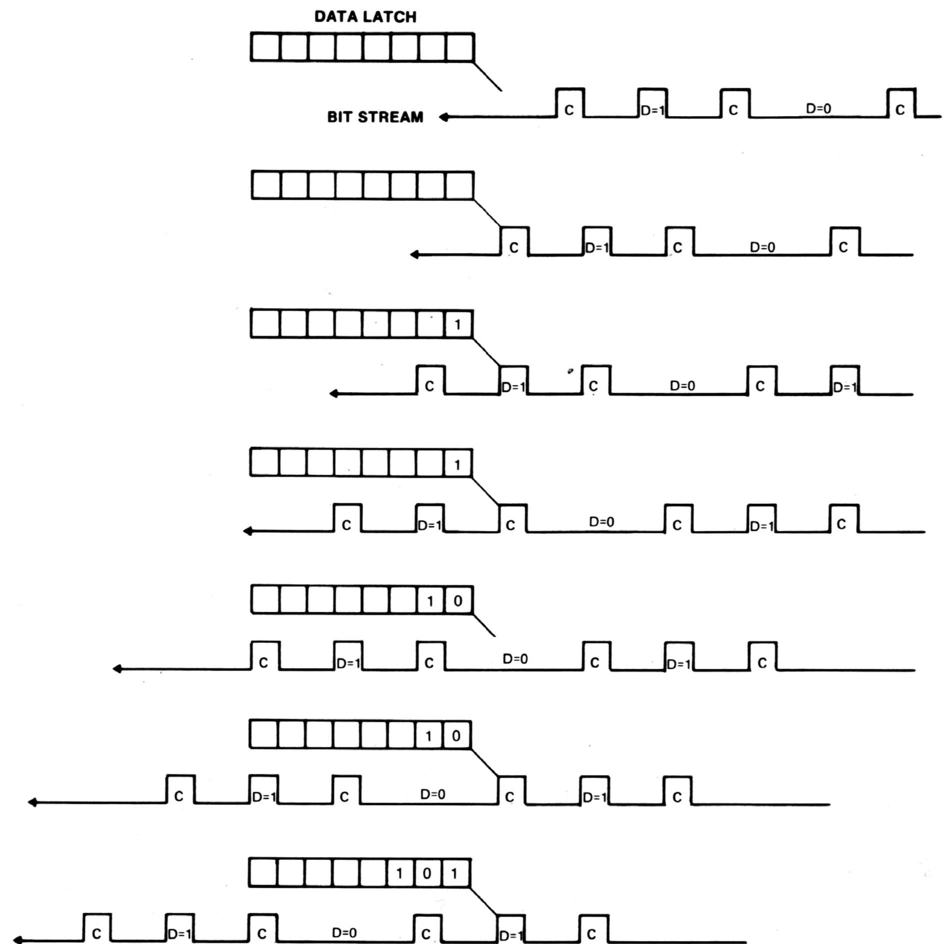


Figure 3.5 Reading Data from Diskette

areas of separation are called "gaps" and are provided for two reasons. One, they allow the updating of one field without affecting adjacent fields (on the Apple, only data fields are updated). Secondly, they allow the computer time to decode the address field before the corresponding data field can pass beneath the read/write head.

All gaps are primarily alike in content, consisting of self-sync hexadecimal `FF`s, and vary only in the number of bytes they contain. Figure 3.7 is a diagram of a portion of a typical track, broken into its major components.

Self-sync or auto-sync bytes are special bytes that make up the three different types of gaps on a track. They are so named because of their ability to automatically bring the hardware into synchronization with data bytes on

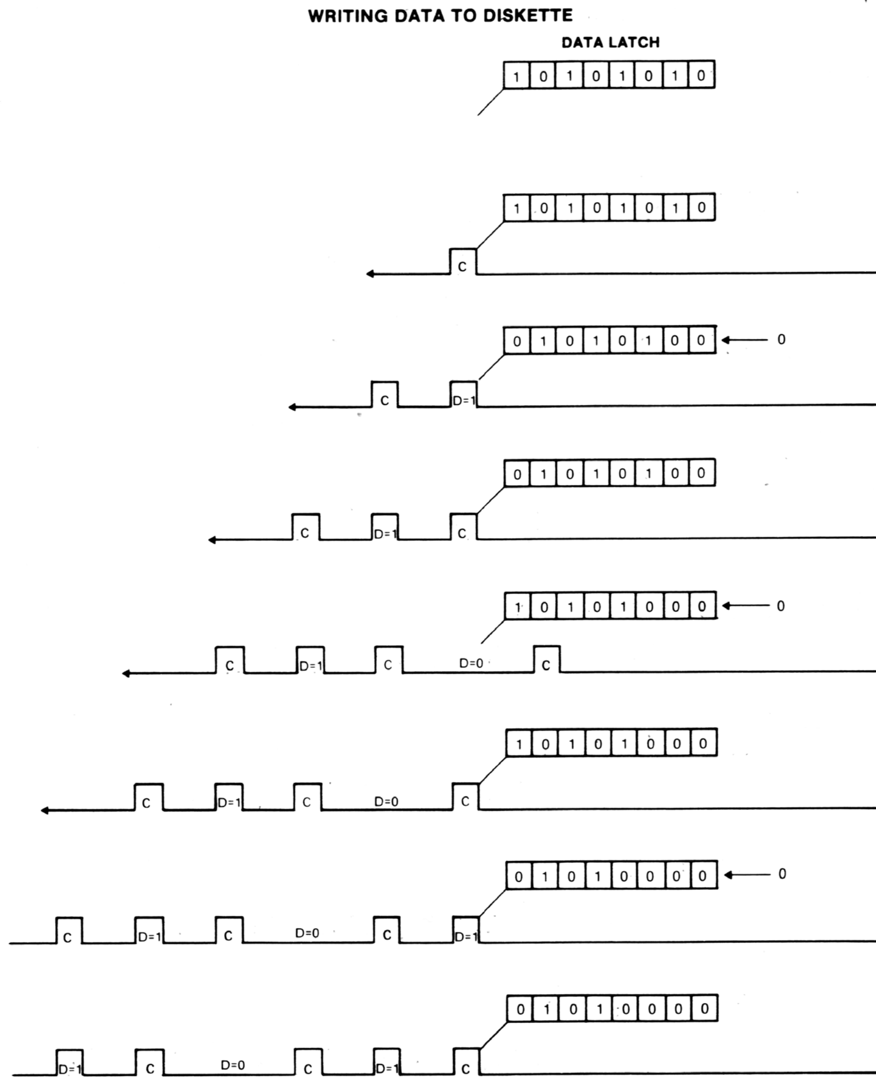


Figure 3.6 Writing Data to Diskette

the disk. The difficulty in doing this lies in the fact that the hardware reads bits and the data must be stored as eight bit bytes. It has been mentioned that a track is literally a continuous stream of data bits. In fact, at the bit level, there is no way to determine where a byte starts or ends, because each bit cell is exactly the same, written in precise intervals with its neighbors. When the drive is instructed to read data, it will start wherever it happens to be on a particular track. That could be anywhere among the 50,000 or so bits on a track. Distinguishing clock bits from data bits, the hardware finds the first bit

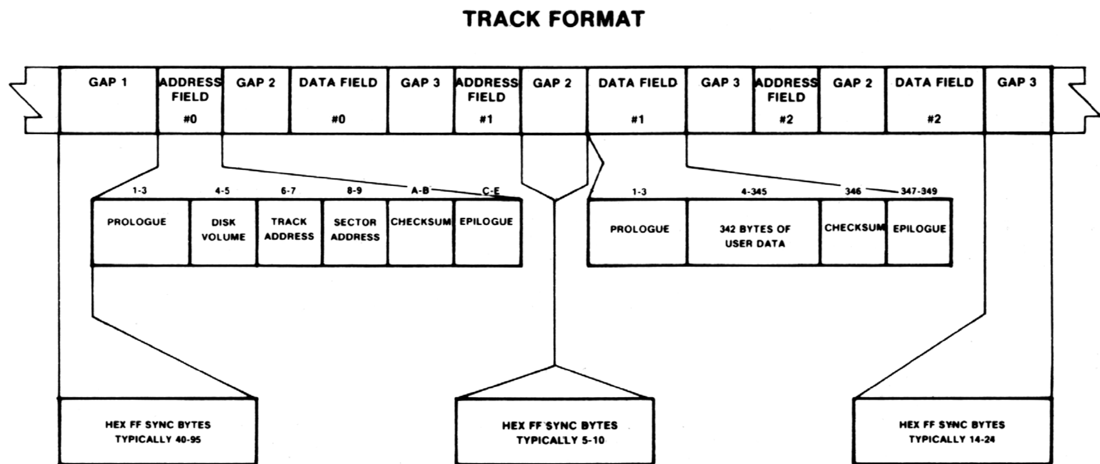


Figure 3.7 Track Format

cell with data in it and proceeds to read the following seven data bits into the eight bit latch. In effect, it assumes that it had started at the beginning of a data byte. Of course, in reality, the odds of its having started at the beginning of a byte are only one in eight. Pictured in Figure 3.8 is a small portion of a track. The clock bits have been stripped out and 0's and 1's have been used for clarity.

There is no way from looking at the data to tell what bytes are represented,

AN EXAMPLE BIT STREAM ON THE DISK

0 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 1 1 1 1 0 1 1 0 1 1 1 0 1 0 1

Figure 3.8 Example Bit Stream

because we don't know where to start. This is exactly the problem that self-sync bytes overcome.

A self-sync byte is defined to be a hexadecimal `FF` with a special difference. It is, in fact, a 10 bit byte rather than an eight bit byte. Its two extra bits are zeros. Figure 3.9 shows the difference between a normal data hex `FF` that might be found elsewhere on the disk and a self-sync hex `FF` byte.

A self-sync is generated by using a 40 cycle (micro-second) loop while writing an `FF`. A bit is written every four cycles, so two of the zero bits brought into the

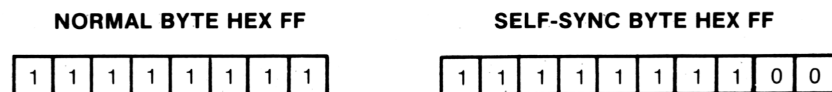


Figure 3.9 Normal Byte and Self-Sync Byte

data latch while the `FF` was being written are also written to the disk, making the 10 bit byte. (DOS 3.2.1 and earlier versions use a nine bit byte due to the

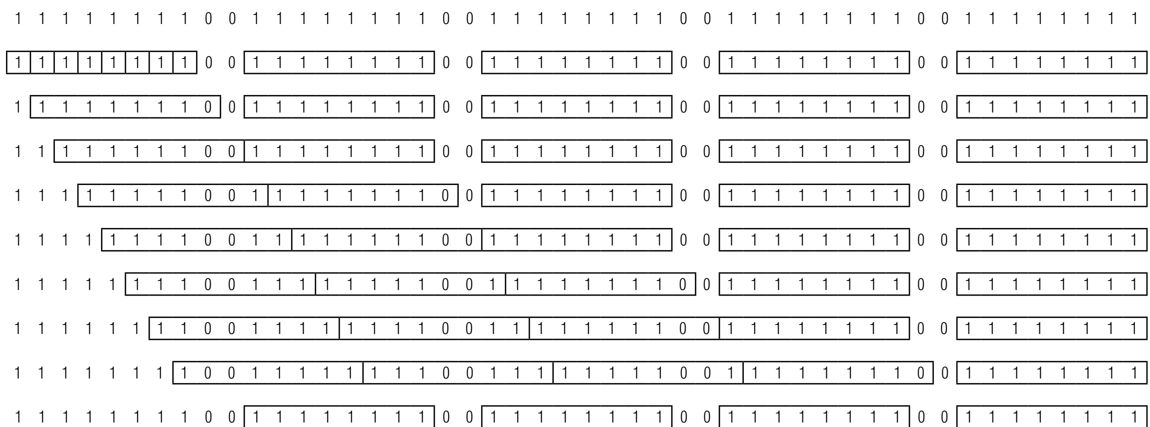


Figure 3.10 Five Auto-Sync Bytes

hardware's inability to always detect two consecutive zero bits.) It can be shown, using Figure 3.10, that five self-sync bytes are sufficient to guarantee that the hardware is reading valid data. The reason for this is that the hardware requires the first bit of a byte to be a 1. Pictured at the top of the figure is a stream of five `FF`s, four self-sync `FF`s followed by a normal `FF`. Each row below that demonstates what the hardware will read should it start reading at any given bit in the first byte. In each case, by the time the five sync bytes have passed beneath the read/write head, the hardware will be "synced" to read the data bytes that follow. As long as the disk is left in read mode, it will continue to correctly interpret the data unless there is an error on the track.

We can now discuss the particular portions of a track in detail. The three gaps will be covered first. Unlike some other disk formats, the size of the three gap types will vary from drive to drive and even from track to track. During the initialization process, DOS will start with large gaps and keep making them smaller until an entire track can be written without overlapping itself. DOS makes sure that each gap type (see Figure 3.7) contains a minimum of four self-sync bytes. The result is fairly uniform gap sizes within each particular track.

Gap 1 is the first data written to a track during initialization. Its purpose is twofold. The gap originally consists of 128 bytes of self-sync, a large enough area to insure that all portions of a track will contain data. Since the speed of a particular drive may vary, the total length of the track in bytes is uncertain, and the percentage occupied by data is unknown. The initialization process is set up, however, so that even on drives of differing speeds, the last data field written will overlap Gap 1, providing continuity over the entire physical track. Care is taken to make sure the remaining portion of Gap 1 is at least as long as a typical Gap 3 (in practice its length is usually more than 400 sync bytes),

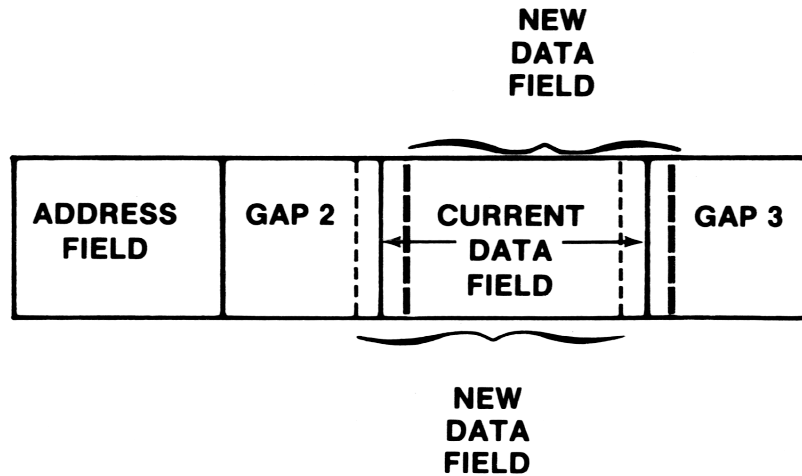


Figure 3.11 Gaps on Diskette

enabling it to serve as a Gap 3 type for Address Field number 0 (See Figure 3.7 for clarity).

Gap 2 appears after each Address Field and before each Data Field. Its length varies from five to ten bytes on a normal drive. The primary purpose of Gap 2 is to provide time for the information in an Address Field to be decoded by the computer before a read or write takes place. If the gap were too short, the beginning of the Data Field might spin past while DOS was still determining if this was the sector to be read. The 240 odd cycles that six self-sync bytes provide seems ample time to decode an address field. When a Data Field is written there is no guarantee that the write will occur in exactly the same spot each time. This is due to the fact that the drive which is rewriting the Data Field may not be the one which originally `INIT`ed or wrote it. Since the speed of the drives can vary, it is possible that the write could start in mid-byte. (See Figure 3.11) This is not a problem as long as the difference in positioning is not great. To insure the integrity of Gap 2, when writing a data field, five self-sync bytes are written prior to writing the Data Field itself. This serves two purposes. Since relatively little time is spent decoding an address field, the five bytes help place the Data Field near its original position. Secondly, and more importantly, the five self-sync bytes guarantee read-synchronization. It is probable that, in writing a Data Field, at least one sync byte will be destroyed. This is because, just as in reading bits on the track, the write may not begin on a byte boundary, thus altering an existing byte. Figure 3.12 illustrates this.

Gap 3 appears after each Data Field and before each Address Field. It is longer than Gap 2 and generally ranges from 14 to 24 bytes in length. It is quite similar in purpose to Gap 2. Gap 3 allows the additional time needed to

manipulate the data that has been read before the next sector is to be read. The length of Gap 3 is not as critical as that of Gap 2. If the following Address

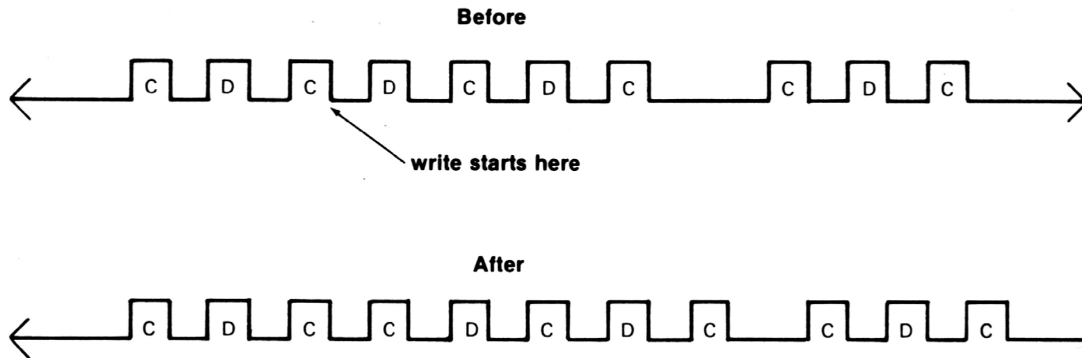
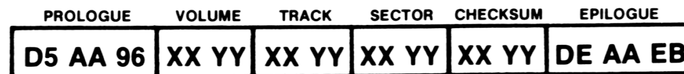


Figure 3.12 Writing Out of Sync

Field is missed, DOS can always wait for the next time it spins around under the read/write head, at most one revolution of the disk. Since Address Fields are never rewritten, there is no problem with this gap providing synchronization, since only the first part of the gap can be overwritten or damaged. (See Figure 3.11 for clarity)

An examination of the contents of the two types of fields is in order. The Address Field contains the "address" or identifying information about the Data Field which follows it. The volume, track, and sector number of any given sector can be thought of as its "address", much like a country, city, and street number might identify a house. As shown previously in Figure 3.7, there are a number of components which make up the Address Field. A more detailed illustration is given in Figure 3.13.

The prologue consists of three bytes which form a unique sequence, found in no other component of the track. This fact enables DOS to locate an Address Field with almost no possibility of error. The three bytes are `$D5`, `$AA`, and `$96`. The `$D5` and `$AA` are reserved (never written as data) thus insuring the uniqueness of the prologue. The `$96`, following this unique string, indicates that the data following constitutes an Address Field (as opposed to a Data Field). The address information follows next, consisting of the volume, track, and sector number and a checksum. This information is absolutely essential for DOS to know where it is positioned on a particular diskette. The checksum is computed by exclusive-ORing the first three pieces of information, and is used to verify its integrity. Lastly follows the epilogue, which contains the three bytes `$DE`, `$AA` and `$EB`. Oddly, the $EB is always written during initialization but is never verified when an Address Field is read. The epilogue bytes are



ODD-EVEN ENCODED

DATA BYTE — $D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0$
XX — $1\ D_7\ 1\ D_5\ 1\ D_3\ 1\ D_1$
YY — $1\ D_6\ 1\ D_4\ 1\ D_2\ 1\ D_0$

Figure 3.13 Address Field

sometimes referred to as "bit-slip marks", which provide added assurance that the drive is still in sync with the bytes on the disk. These bytes are probably unnecessary, but do provide a means of double checking.

The other field type is the Data Field. Much like the Address Field, it consists of a prologue, data, checksum, and an epilogue. (Refer to Figure 3.14) The prologue is different only in the third byte. The bytes are `$D5`, `$AA`, and `$AD`, which again form a unique sequence, enabling DOS to locate the beginning of the sector data. The data consists of 342 bytes of encoded data. The encoding scheme used will be discussed in the next section. The data is followed by a checksum byte, used to verify the integrity of the data just read. The epilogue portion of the Data Field is absolutely identical to the epilogue in the Address Field and it serves the same function.

DATA FIELD ENCODING

Due to Apple's hardware, it is not possible to read all 256 possible byte values from a diskette. This is not a great problem, but it does require that the data written to the disk be encoded. Three different techniques have been used. The

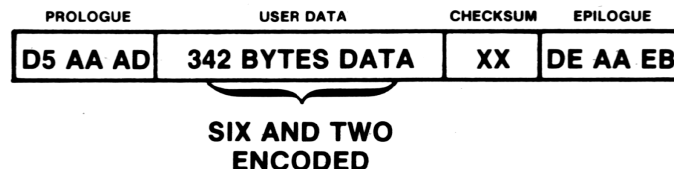


Figure 3.14 Data Field

first one, which is currently used in Address Fields, involves writing a data byte as two disk bytes, one containing the odd bits, and the other containing the even

bits. It would thus require 512 "disk" bytes for each 256 byte sector of data. Had this technique been used for sector data, no more than 10 sectors would have fit on a track. This amounts to about 88K of data per diskette, or roughly 72K of space available to the user; typical for 5 ¼-inch single density drives.

Fortunately, a second technique for writing data to diskette was devised that allows 13 sectors per track. This new method involved a "5 and 3" split of the data bits, versus the "4 and 4" mentioned earlier. Each byte written to the disk contains five valid bits rather than four. This requires 410 "disk" bytes to store a 256 byte sector. This latter density allows the now well-known 13 sectors per track format used by DOS 3 through DOS 3.2.1. The "5 and 3" scheme represented a hefty 33% increase over comparable drives of the day.

Currently, of course, DOS 3.3 features 16 sectors per track and provides

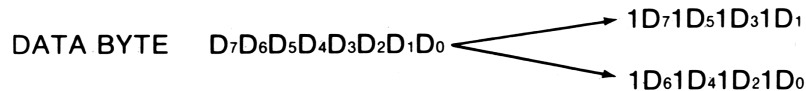


Figure 3.15 "4 and 4" Encoding

a 23% increase in disk storage over the 13 sector format. This was made possible by a hardware modification (the P6 PROM on the disk controller card) which allowed a "6 and 2" split of the data. The change was to the logic of the "state machine" in the P6 PROM, now allowing two consecutive zero bits in data bytes.

These three different encoding techniques will now be covered in some detail. The hardware for DOS 3.2.1 (and earlier versions of DOS) imposed a number of restrictions upon how data could be stored and retrieved. It required

$$\begin{array}{rl} & D_7\ 1\ D_5\ 1\ D_3\ 1\ D_1\ 1 \quad \text{(shifted left)} \\ \text{AND} & 1\ D_6\ 1\ D_4\ 1\ D_2\ 1\ D_0 \\ \hline & D_7D_6D_5D_4D_3D_2D_1D_0 \end{array}$$

Figure 3.16 "4 and 4" Decoding

that a disk byte have the high bit set and, in addition, no two consecutive bits could be zero. The odd-even "4 and 4" technique meets these requirements. Each data byte is represented as two bytes, one containing the even data bits and the other the odd data bits. Figure 3.15 illustrates this transformation. It should be noted that the unused bits are all set to one to guarantee meeting the two requirements.

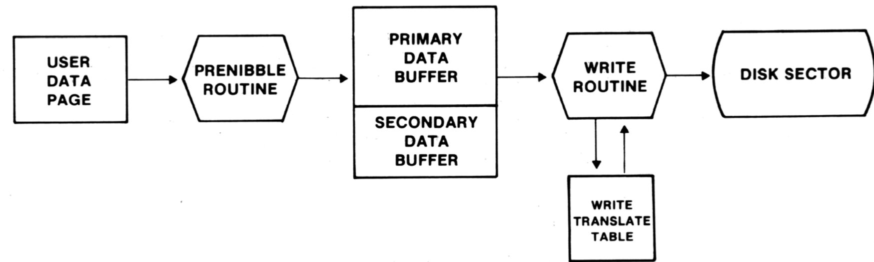


Figure 3.17 Memory Bits to Disk Bits Process

No matter what value the original data data byte has, this technique insures that the high bit is set and that there can not be two consecutive zero bits. The "4 and 4" technique is used to store the information (volume, track, sector, checksum) contained in the Address Field. It is quite easy to decode the data, since the byte with the odd bits is simply shifted left and logically ANDed with the byte containing the even bits. This is illustrated in Figure 3.16.

It is important that the least significant bit contain a 1 when the odd-bits byte is left shifted. The entire operation is carried out in the `RDADR` subroutine at `$B944` in DOS (48K).

The major difficulty with the above technique is that it takes up a lot of room on the track. To overcome this deficiency the "5 and 3" encoding technique was developed. It is so named because, instead of splitting the bytes in half, as in the odd-even technique, they are split five and three. A byte would have the form `ØØØXXXXX`, where `X` is a valid data bit. The above byte could range in value from `$ØØ` to `$1F`, a total of 32 different values. It so happens that there are 34 valid "disk" bytes, ranging from `$AA` up to `$FF`, which meet the two requirements (high bit set, no consecutive zero bits). Two bytes, `$D5` and `$AA`, were chosen as reserved bytes, thus leaving an exact mapping between five bit data bytes and eight bit "disk" bytes. The process of converting eight bit data bytes to eight bit "disk" bytes, then, is twofold. An overview is diagrammed in Figure 3.17.

First, the 256 bytes that will make up a sector must be translated to five bit bytes. This is done by the "prenibble" routine at `$B8ØØ`. It is a fairly involved process, involving a good deal of bit rearrangement. Figure 3.18 shows the before and after of prenibbilizing. On the left is a buffer of eight bit data bytes, as passed to the RWTS subroutine package by DOS. Each byte in this buffer is represented by a letter (A, B, C, etc.) and each bit by a number (7 through 0). On the right side are the results of the transformation. The primary buffer contains five distinct areas of five bit bytes (the top three bits of the eight bit bytes zero-filled) and the secondary buffer contains three areas, graphically

illustrating the name "5 and 3".

A total of 410 bytes are needed to store the original 256. This can be calculated by finding the total bits of data (256 x 8 = 2048) and dividing that by the number of bits per byte (2048/5 = 409.6). (two bits are not used) Once this process is completed, the data is further transformed to make it valid "disk" bytes, meeting the disk's requirements. This is much easier, involving a one to one look-up in the table given in Figure 3.19.

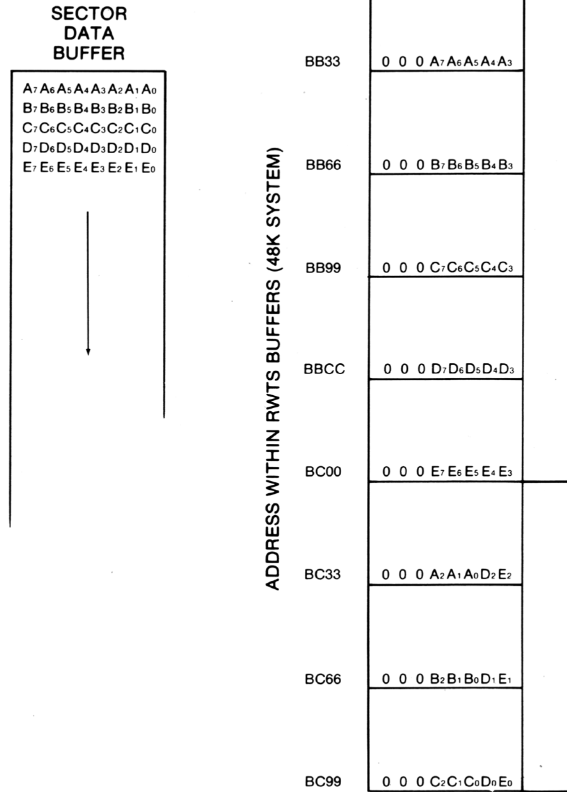


Figure 3.18 "5 and 3" Prenibbilizing

The Data Field has a checksum much like the one in the Address Field, used to verify the integrity of the data. It also involves exclusive-ORing the information, but, due to time constraints during reading bytes, it is implemented differently. The data is exclusive-ORed in pairs before being transformed by the look-up table in Figure 3.19. This can best be illustrated by Figure 3.20.

The reason for this transformation can be better understood by examining how the information is retrieved from the disk. The read routine must read a byte, transform it, and store it—all in under 32 cycles (the time taken to write a byte) or the information will be lost. By using the checksum computation to decode data, the transformation shown in Figure 3.20 greatly facilitates the time constraint. As the data is being read from a sector the accumulator contains the cumulative result of all previous bytes, exclusive-ORed together. The value of the accumulator after any exclusive-OR operation is the actual data byte for that point in the series. This process is diagrammed in Figure 3.21.*

* Figures 3.20 and 3.21 present the nibblizing process used by the "6 and 2" encoding technique. However, the concept is the same for the "5 and 3" technique.

ØØ = AB	Ø8 = BB	1Ø = DD	18 = F5
Ø1 = AD	Ø9 = BD	11 = DE	19 = F6
Ø2 = AE	ØA = BE	12 = DF	1A = F7
Ø3 = AF	ØB = BF	13 = EA	1B = FA
Ø4 = B5	ØC = D6	14 = EB	1C = FB
Ø5 = B6	ØD = D7	15 = ED	1D = FD
Ø6 = B7	ØE = DA	16 = EE	1E = FE
Ø7 = BA	ØF = DB	17 = EF	1F = FF

AA } Reserved Bytes
D5 }

Figure 3.19 "5 and 3" Write Translate Table

The third encoding technique, currently used by DOS 3.3, is similar to the "5 and 3". It was made possible by a change in the hardware which eased the requirements for valid data somewhat. The high bit must still be set, but now the byte may contain one (and only one) pair of consecutive zero bits. This allows a greater number of valid bytes and permits the use of a "6 and 2" encoding technique. A six bit byte would have the form ØØXXXXXX and has values from $ØØ to $3F for a total of 64 different values. With the new, relaxed requirements for valid "disk" bytes there are 70 different bytes ranging in value from $95 up to $FF. After removing the two reserved bytes, $AA and $D5, there are still 68 "disk" bytes with only 64 needed. An additional requirement was introduced to force the mapping to be one to one, namely, that there must be at least two adjacent bits set, excluding bit 7. This eliminates four more bytes and produces exactly 64 valid "disk" values. The initial transformation is done by the prenibble routine (still located at $B8ØØ) and its results are shown in Figure 3.22.

A total of 342 bytes are needed, shown by finding the total number of bits (256 x 8 = 2048) and dividing by the number of bits per byte (2048 / 6 = 341.33). The transformation from the six bit bytes to valid data bytes is again performed by a one to one mapping shown in Figure 3.23. Once again, the stream of data bytes written to the diskette are a product of exclusive-ORs, exactly as with the "5 and 3" technique discussed earlier.

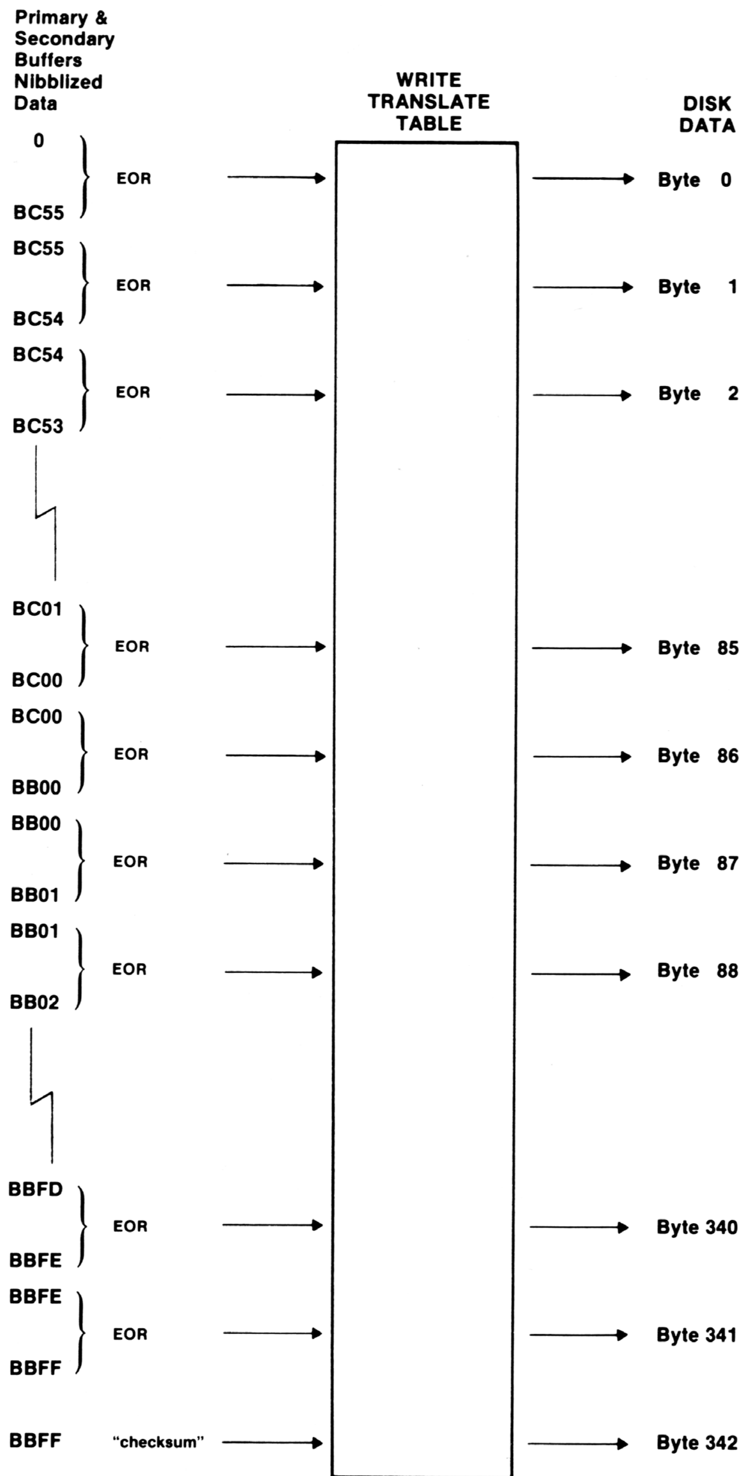


Figure 3.20 Writing to Diskette, DOS 3.3

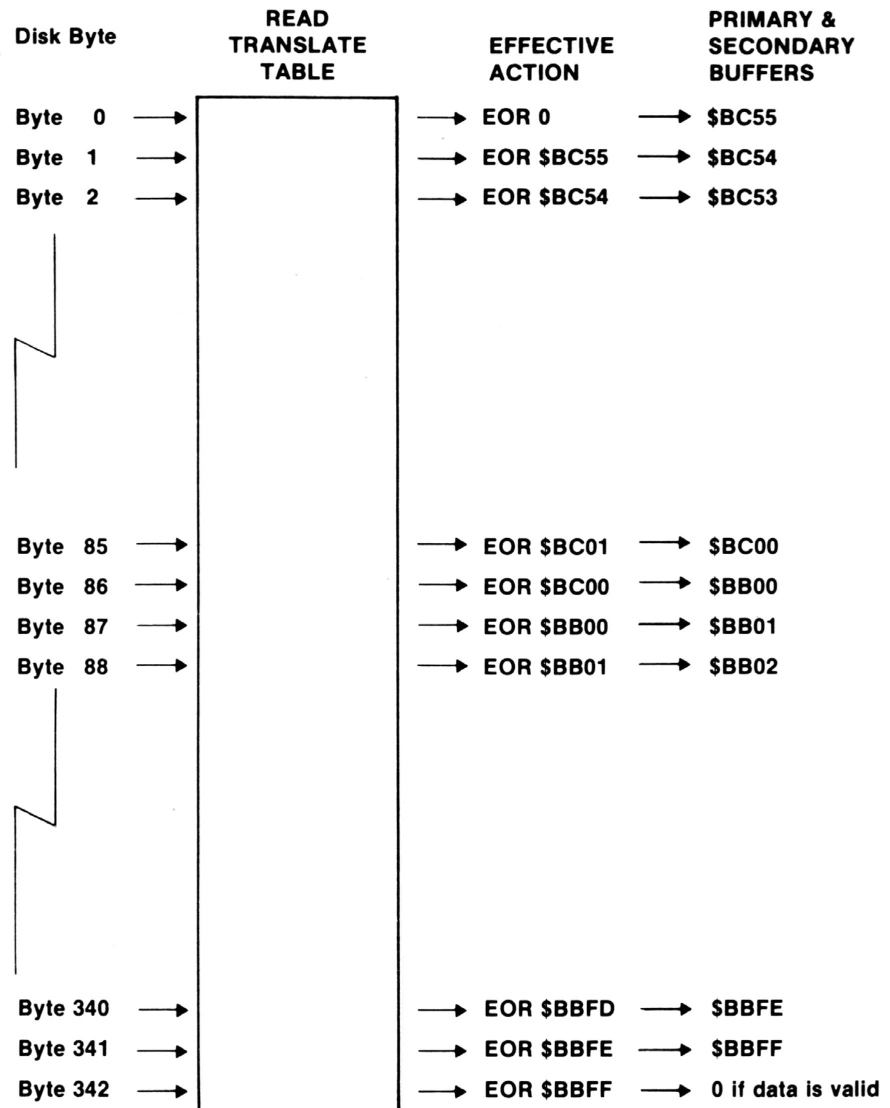


Figure 3.21 Reading From Diskette, DOS 3.3

SECTOR INTERLEAVING

Sector interleaving, or skewing, is the staggering of sectors on a track to maximize access speed. There is usually a delay between the time DOS reads or writes a sector and the time it is ready to read or write another. This delay depends upon the application program using the disk and can vary greatly. If sectors were stored on the track in sequential order, it would usually be necessary to wait a full revolution of the diskette before the next sector could

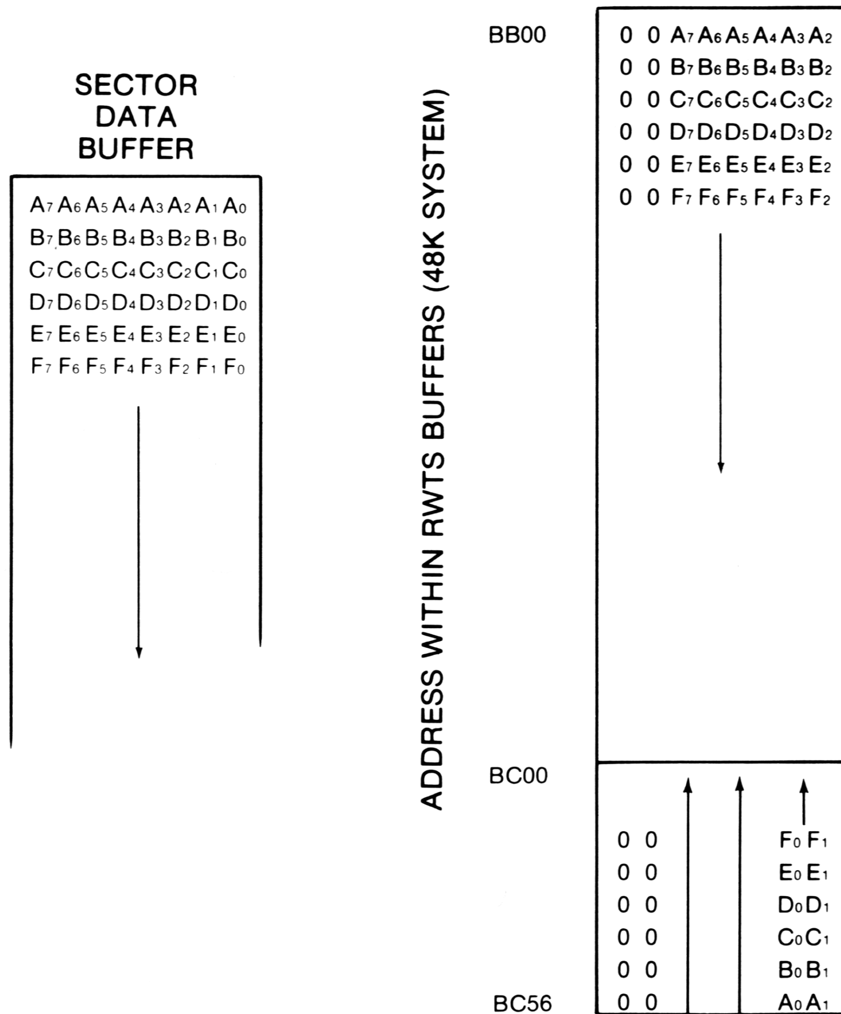


Figure 3.22 "6 and 2" Prenibbilizing

be accessed. Ordering the sectors non-sequentially (skewing them) can provide improved access speeds.

On DOS 3.2.1 and earlier versions, the 13 sectors are physically skewed on the disk. During the boot operation, sectors are loaded from the diskette in ascending sequential order. However, files generally are stored in descending sequential order. As a result, no single interleaving scheme works well for both booting and sequentially accessing a file.

A different approach has been used in DOS 3.3 in an attempt to maximize performance. The skewing is now done in software. The 16 physical sectors are stored in ascending order (0, 1, 2, ... , 15) and are not physically skewed at all. A look-up table is used to translate a logical or soft sector number used by

```
ØØ = 96     1Ø = B4     2Ø = D6     3Ø = ED
Ø1 = 97     11 = B5     21 = D7     31 = EE
Ø2 = 9A     12 = B6     22 = D9     32 = EF
Ø3 = 9B     13 = B7     23 = DA     33 = F2
Ø4 = 9D     14 = B9     24 = DB     34 = F3
Ø5 = 9E     15 = BA     25 = DC     35 = F4
Ø6 = 9F     16 = BB     26 = DD     36 = F5
Ø7 = A6     17 = BC     27 = DE     37 = F6
Ø8 = A7     18 = BD     28 = DF     38 = F7
Ø9 = AB     19 = BE     29 = E5     39 = F9
ØA = AC     1A = BF     2A = E6     3A = FA
ØB = AD     1B = CB     2B = E7     3B = FB
ØC = AE     1C = CD     2C = E9     3C = FC
ØD = AF     1D = CE     2D = EA     3D = FD
ØE = B2     1E = CF     2E = EB     3E = FE
ØF = B3     1F = D3     2F = EC     3F = FF

        AA }
        D5 }  Reserved Bytes
```

Figure 3.23 "6 and 2" Write Translate Table

RWTS into the physical sector number found on the diskette. For example, if the logical sector number were a 2, this would be translated into the physical sector number 11 ($ØB). Thus RWTS treats physical sector 11 ($ØB) as sector 2 for all intents and purposes. This presents no problem if RWTS is used for disk access, but would become a consideration if access were made without RWTS. DOS 3.3 uses what we refer to as a "2 descending" skew.

In an attempt to eliminate the access differences between booting and reading files, another change was made to DOS 3.3. During the boot process, DOS is loaded backwards in descending sequential order into memory, just as files are accessed. However, due to differences in the delays for booting and reading files, no single skewing scheme is optimal.

It is interesting to point out that Pascal, Fortran, and CP/M diskettes all use software interleaving also. However, each uses a different sector order. Pascal and Fortran use a 2 ascending skew and CP/M diskettes use a 3 ascending skew. A comparison of these differences is presented in Figure 3.24.

Physical Sector	DOS 3.3 Sector	Pascal Sector	CP/M Sector
Ø	Ø	Ø	Ø
1	D	2	3
2	B	4	6
3	9	6	9
4	7	8	C
5	5	A	F
6	3	C	2
7	1	E	5
8	E	1	8
9	C	3	B
A	A	5	E
B	8	7	1
C	6	9	4
D	4	B	7
E	2	D	A
F	F	F	D

Figure 3.24 Comparison of Sector Interleaving

Chapter 4

Diskette Organization

As was described in Chapter 3, a 16 sector diskette consists of 560 data areas of 256 bytes each, called sectors. These sectors are arranged on the diskette in 35 concentric rings or tracks of 16 sectors each. The way DOS allocates these tracks of sectors is the subject of this chapter.

A file (be it Applesoft, Integer, Binary, or Text type) consists of one or more sectors containing data. Since the sector is the smallest unit of allocatable space on a diskette, a file will use up at least one sector even if it is less than 256 bytes long; the remainder of the sector is wasted. Thus, a file containing 400 characters (or bytes) of data will occupy one entire sector and 144 bytes of another with 112 bytes wasted. Knowing these facts, one would expect to be able to use up to 16 times 35 times 256 or 143,360 bytes of space on a diskette for files. Actually, the largest file that can be stored is about 126,000 bytes long.

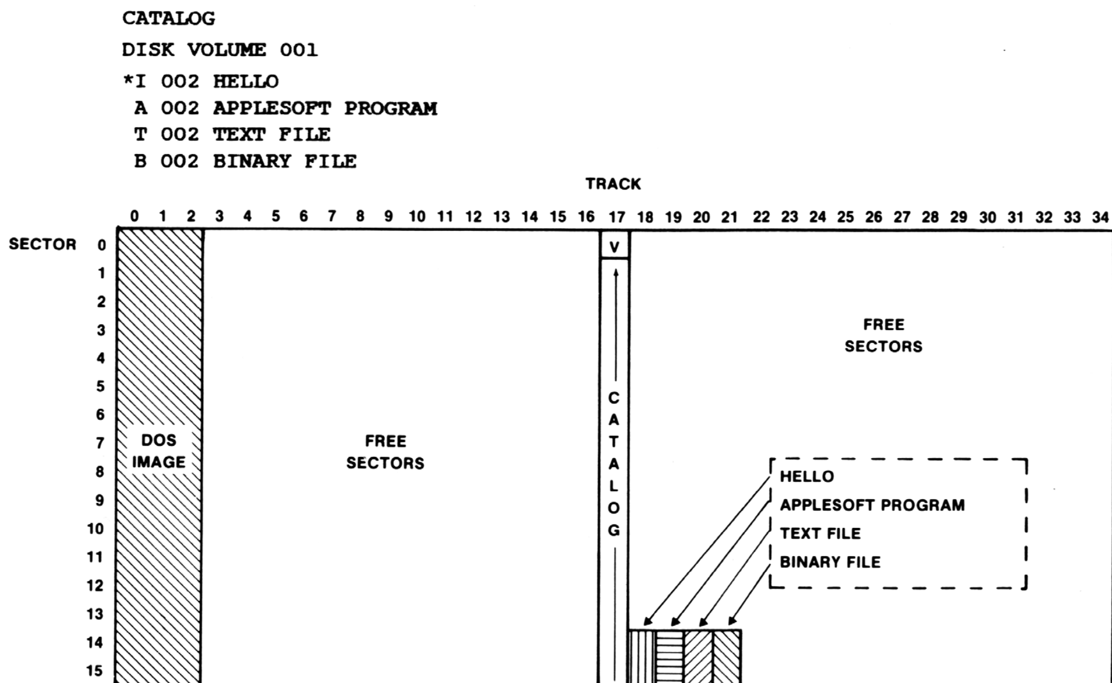


Figure 4.1 A Typical 16 Sector Diskette Map

The reason for this is that some of the sectors on the diskette must be used for what is called "overhead".

Overhead sectors contain the image of DOS which is 1oaded when booting the diskette, a list of the names and locations of the files on the diskette, and an accounting of the sectors which are free for use with new files or expansions of existing files. An example of the way DOS uses sectors is given in Figure 4.1.

DISKETTE SPACE ALLOCATION

The map in Figure 4.1 shows that the first three tracks of each diskette are always reserved for the bootstrap image of DOS. In the exact center track (track 17) is the VTOC and catalog. The reason for placing the catalog here is simple. Since the greatest delay when using the disk is waiting for the arm to move from track to track, it is advantageous to minimize this arm movement whenever possible. By placing the catalog in the exact center track of the disk, the arm need never travel more than 17 tracks to get to the catalog track. As files are allocated on a diskette, they occupy the tracks just above the catalog track first. When the last track, track 34, has been used, track 16, the track adjacent and below the catalog, is used next, then 15, 14, 13, and so on, moving away from the catalog again, toward the DOS image tracks. If there are very few files on the diskette, they will all be clustered, hopefully, near the catalog and arm movement will be minimized. Additional space for a file, if it is needed, is first allocated in the same track occupied by the file. When that track is full, another track is allocated elsewhere on the disk in the manner described above.

HOW DOS IS STORED ON THE DISK

The following table shows how DOS is stored on the disk and which track and sector corresponds to which memory location:

Master Track/Sector	Slave Track/Sector	Master/Slave Track/Sector	Load Address	Relocated Address
DOS 3.2.1		DOS 3.3		
ØØ, ØØ	ØØ, ØØ	ØØ, ØØ	36ØØ	B6ØØ
ØØ, Ø1	ØØ, Ø1	ØØ, Ø1	37ØØ	B7ØØ
ØØ, Ø2	ØØ, Ø2	ØØ, Ø2	38ØØ	B8ØØ
ØØ, Ø3	ØØ, Ø3	ØØ, Ø3	39ØØ	B9ØØ
ØØ, Ø4	ØØ, Ø4	ØØ, Ø4	3AØØ	BAØØ
ØØ, Ø5	ØØ, Ø5	ØØ, Ø5	3BØØ	BBØØ
ØØ, Ø6	ØØ, Ø6	ØØ, Ø6	3CØØ	BCØØ
ØØ, Ø7	ØØ, Ø7	ØØ, Ø7	3DØØ	BDØØ

Master Track/Sector	Slave Track/Sector	Master/Slave Track/Sector	Load Address	Relocated Address
DOS 3.2.1		DOS 3.3		
ØØ, Ø8	ØØ, Ø8	ØØ, Ø8	3EØØ	BEØØ
ØØ, Ø9	ØØ, Ø9	ØØ, Ø9	3FØØ	BFØØ
ØØ, ØA	—	ØØ, ØA	1BØØ	—
ØØ, ØB	—	ØØ, ØB	1CØØ	—
ØØ, ØC	ØØ, ØA	ØØ, ØC	1DØØ	9DØØ
Ø1, ØØ	ØØ, ØB	ØØ, ØD	1EØØ	9EØØ
Ø1, Ø1	ØØ, ØC	ØØ, ØE	1FØØ	9FØØ
Ø1, Ø2	Ø1, ØØ	ØØ, ØF	2ØØØ	AØØØ
Ø1, Ø3	Ø1, Ø1	Ø1, ØØ	21ØØ	A1ØØ
Ø1, Ø4	Ø1, Ø2	Ø1, Ø1	22ØØ	A2ØØ
Ø1, Ø5	Ø1, Ø3	Ø1, Ø2	23ØØ	A3ØØ
Ø1, Ø6	Ø1, Ø4	Ø1, Ø3	24ØØ	A4ØØ
Ø1, Ø7	Ø1, Ø5	Ø1, Ø4	25ØØ	A5ØØ
Ø1, Ø8	Ø1, Ø6	Ø1, Ø5	26ØØ	A6ØØ
Ø1, Ø9	Ø1, Ø7	Ø1, Ø6	27ØØ	A7ØØ
Ø1, ØA	Ø1, Ø8	Ø1, Ø7	28ØØ	A8ØØ
Ø1, ØB	Ø1, Ø9	Ø1, Ø8	29ØØ	A9ØØ
Ø1, ØC	Ø1, ØA	Ø1, Ø9	2AØØ	AAØØ
Ø2, ØØ	Ø1, ØB	Ø1, ØA	2BØØ	ABØØ
Ø2, Ø1	Ø1, ØC	Ø1, ØB	2CØØ	ACØØ
Ø2, Ø2	Ø2, ØØ	Ø1, ØC	2DØØ	ADØØ
Ø2, Ø3	Ø2, Ø1	Ø1, ØD	2EØØ	AEØØ
Ø2, Ø4	Ø2, Ø2	Ø1, ØE	2FØØ	AFØØ
Ø2, Ø5	Ø2, Ø3	Ø1, ØF	3ØØØ	BØØØ
Ø2, Ø6	Ø2, Ø4	Ø2, ØØ	31ØØ	B1ØØ
Ø2, Ø7	Ø2, Ø5	Ø2, Ø1	32ØØ	B2ØØ
Ø2, Ø8	Ø2, Ø6	Ø2, Ø2	33ØØ	B3ØØ
Ø2, Ø9	Ø2, Ø7	Ø2, Ø3	34ØØ	B4ØØ
Ø2, ØA	Ø2, Ø8	Ø2, Ø4	35ØØ	B5ØØ

THE VTOC

The Volume Table Of Contents is the "anchor" of the entire diskette. On any diskette accessible by any version of DOS, the VTOC sector is always in the same place; track 17, sector 0. (Some protected disks have the VTOC at another location and provide a special DOS which can find it.) Since files can end up anywhere on the diskette, it is through the VTOC anchor that DOS is able to find them. The VTOC of a diskette has the following format (all byte offsets are given in base 16, hexadecimal):

VOLUME TABLE OF CONTENTS (VTOC) FORMAT

BYTE	DESCRIPTION
ØØ	Not used
Ø1	Track number of first catalog sector
Ø2	Sector number of first catalog sector
Ø3	Release number of DOS used to INIT this diskette
Ø4-Ø5	Not used
Ø6	Diskette volume number (1-254)
Ø7-26	Not used
27	Maximum number of track/sector pairs which will fit in one file track/sector list sector (122 for 256 byte sectors)
28-2F	Not used
3Ø	Last track where sectors were allocated
31	Direction of track allocation (+1 or -1)
32-33	Not used
34	Number of tracks per diskette (normally 35)
35	Number of sectors per track (13 or 16)
36-37	Number of bytes per sector (LO/HI format)
38-3B	Bit map of free sectors in track Ø
3C-3F	Bit map of free sectors in track 1
4Ø-43	Bit map of free sectors in track 2
	...
BC-BF	Bit map of free sectors in track 33
CØ-C3	Bit map of free sectors in track 34
C4-FF	Bit maps for additional tracks if there are more than 35 tracks per diskette

BIT MAPS OF FREE SECTORS ON A GIVEN TRACK

A four byte binary string of ones and zeros, representing free and allocated sectors respectively. Hexadecimal sector numbers are assigned to bit positions as follows:

```
BYTE     SECTORS
+Ø       FEDC BA98
+1       7654 321Ø
+2       .... .... (not used)
+3       .... .... (not used)
Thus, if only sectors E and 8 are free and all others are
allocated, the bit map will be:
41ØØØØØØ
If all sectors are free:
FFFFØØØØ
```

An example of a VTOC sector is given in Figure 4.2. This VTOC corresponds to the map of the diskette given in Figure 4.1.

```
          A    B        C
00    04 11 0F 03 00 00 01 00 00 00 00 00 00 00 00 00
10    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                           D
20    00 00 00 00 00 00 00 7A 00 00 00 00 00 00 00 00
       E           F     G        H           I
30    15 01 00 00 23 10 00 01 00 00 00 00 00 00 00 00
40    00 00 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
50    FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
                                  J
60    FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
70    FF FF 00 00 FF FF 00 00 FF FF 00 00 00 00 00 00
          K
80    3F FF 00 00 3F FF 00 00 3F FF 00 00 FF FF 00 00
90    FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
A0    FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
B0    FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
          L
C0    FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
F0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

- **A** First CATALOG sector is on Track $11 (17), sector $0F (15)
- **B** This is a DOS 3.3 disk
- **C** Volume #1
- **D** $7A (122) Track/Sector pairs in a Track/Sector list
- **E** Last allocated Track was $15 (21), next will be $16 (22)
- **F** $23 (35) tracks per disk, $10 (16) sectors per track
- **G** $0100 (256) bytes per sector
- **H** Track 0 has no available sectors
- **I** Track 1 has no available sectors
- **J** Track 12 has all sectors available
- **K** Only sectors $0E (14) and $0F (15) are used on track $12 (18)
- **L** Track $22 (34) has all sectors available

Figure 4.2 Example VTOC

THE CATALOG

In order for DOS to find a given file, it must first read the VTOC to find out where the first catalog sector is located. Typically, the catalog sectors for a diskette are the remaining sectors on track 17, following the VTOC sector. Of course, as long as a track/sector pointer exists in the VTOC and the VTOC is located at track 17, sector 0, DOS does not really care where the catalog resides. Figure 4.3 diagrams the catalog track. The figure shows the track/sector pointer in the VTOC at bytes 01 and 02 as an arrow pointing to track 17 (11 in hexadecimal) sector F. The last sector in the track is the first catalog sector and describes the first seven files on the diskette. Each catalog sector has a track/sector pointer in the same position (bytes 01 and 02) which points to the next catalog sector. The last catalog sector (sector 1) has a zero pointer to indicate that there are no more catalog sectors in the chain.

In each catalog sector up to seven files may be listed and described. Thus, on a typical DOS 3.3 diskette, the catalog can hold up to 15 times 7, or 105 files. A catalog sector is formatted as described on the following page.

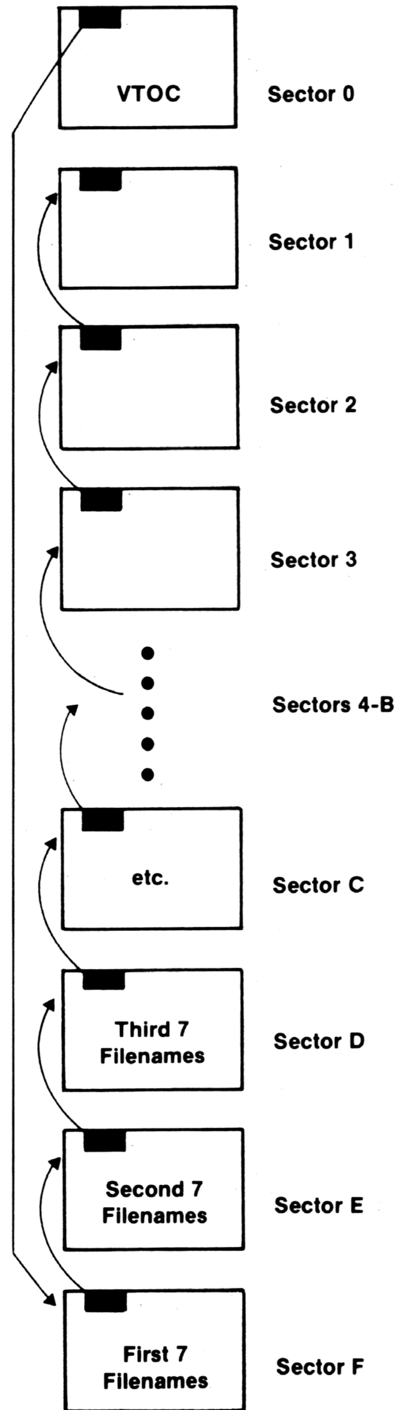


Figure 4.3 Track 17, the CATALOG Track

CATALOG SECTOR FORMAT

BYTE	DESCRIPTION
ØØ	Not used
Ø1	Track number of next catalog sector (usually 11 hex)
Ø2	Sector number of next catalog sector
Ø3-ØA	Not used
ØB-2D	First file descriptive entry
2E-5Ø	Second file descriptive entry
51-73	Third file descriptive entry
74-96	Fourth file descriptive entry
97-B9	Fifth file descriptive entry
BA-DC	Sixth file descriptive entry
DD-FF	Seventh file descriptive entry

FILE DESCRIPTIVE ENTRY FORMAT

```
RELATIVE
BYTE     DESCRIPTION
ØØ       Track of first track/sector list sector.
         If this is a deleted file, this byte contains a hex
         FF and the original track number is copied to the
         last byte of the file name field (Byte 2Ø).
         If this byte contains a hex ØØ, the entry is assumed
         to never have been used and is available for use.
         (This means track Ø can never be used for data even
         if the DOS image is "wiped" from the diskette.)
Ø1       Sector of first track/sector list sector
Ø2       File type and flags:
            Hex 8Ø+file type - file is locked
                ØØ+file type - file is not locked
                ØØ - Text file
                Ø1 - Integer BASIC file
                Ø2 - Applesoft BASIC file
                Ø4 - Binary file
                Ø8 - S type file
                1Ø - Relocatable object module file
                2Ø - A type file
                4Ø - B type file
            (thus, 84 is a locked Binary file, and 9Ø is a locked
R type file)
Ø3-2Ø    File name (3Ø characters)
21-22    Length of file in sectors (LO/HI format).
         The CATALOG command will only format the LO byte of
         this length giving 1-255 but a full 65,535 may be
         stored here.
```

```
             ØØ    ØØ                         .           Unused
                                                          Next CATALOG sector is Track $11 (17), Sector
             Ø1    11 ØE                      .           $ØE (14)
             Ø3    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........    Unused
-------------------------------------------------------------------------------------------
File Entry 1                                              First Track/Sector list for this file is Track
             ØB    12 ØF                      ..          $12 (18), Sector $ØF (15)
             ØD    81                         .           $81 = Locked, Integer BASIC program file
             ØE    C8 C5 CC CC CF AØ AØ AØ    H E L L O   File name is "HELLO"
             16    AØ AØ AØ AØ AØ AØ AØ AØ
             1E    AØ AØ AØ AØ AØ AØ AØ AØ
             26    AØ AØ AØ AØ AØ AØ
             2C    Ø2 ØØ                      ..          File Length is $ØØØ2 (2) sectors
-------------------------------------------------------------------------------------------
File Entry 2                                              First Track/Sector list for this file is Track
             2E    13 ØF                      ..          $13 (19), Sector $ØF (15)
             3Ø    Ø2                         .           $Ø2 = Unlocked Applesoft program file
             31    C1 DØ DØ CC C5 D3 CF C6    A P P L E S O F   File name is "APPLESOFT PROGRAM"
             39    D4 AØ DØ D2 CF C7 D2 C1    T   P R O G R A
             41    CD AØ AØ AØ AØ AØ AØ AØ    M
             49    AØ AØ AØ AØ AØ AØ
             4F    Ø2 ØØ                      ..          File Length is $ØØØ2 (2) sectors
-------------------------------------------------------------------------------------------
File Entry 3                                              First Track/Sector list for this file is Track
             51    14 ØF                      ..          $14 (2Ø), Sector $ØF (15)
             53    ØØ                         .           $ØØ = Unlocked Text file
             54    D4 C5 D8 D4 AØ C6 C9 CC    T E X T   F I L   File name is "TEXT FILE"
             5C    C5 AØ AØ AØ AØ AØ AØ AØ    E
             64    AØ AØ AØ AØ AØ AØ AØ AØ
             6C    AØ AØ AØ AØ AØ AØ
             72    Ø2 ØØ                      ..          File Length is $ØØØ2 (2) sectors
-------------------------------------------------------------------------------------------
File Entry 4                                              First Track/Sector list for this file is Track
             74    15 ØF                      ..          $15 (21), Sector $ØF (15)
             76    Ø4                         .           $Ø4 = Unlocked Binary image file
             77    C2 C9 CE C1 D2 D9 AØ C6    B I N A R Y   F   File name is "BINARY FILE"
             7F    C9 CC C5 AØ AØ AØ AØ AØ    I L E
             87    AØ AØ AØ AØ AØ AØ AØ AØ
             8F    AØ AØ AØ AØ AØ AØ
             95    Ø2 ØØ                      ..          File Length is $ØØØ2 (2) sectors
-------------------------------------------------------------------------------------------
             97    ØØ                         .
             98    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........    Rest of CATALOG sector is empty
             AØ    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........
             A8    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........
             BØ    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........
             B8    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........    Alternating Shaded/Unshaded areas show where
             CØ    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........    file entries would be placed if more files
             C8    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........    are added to the diskette.
             DØ    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........
             D8    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........
             EØ    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........
             E8    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........
             FØ    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........
             F8    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ    ........
```

Figure 4.4 Typical CATALOG Sector

Figure 4.4 is an example of a typical catalog sector. In this example there are only four files on the entire diskette, so only one catalog sector was needed to describe them. There are four entries in use and three entries which have never been used and contain zeros.

THE TRACK/SECTOR LIST

Each file has associated with it a "Track/Sector List" sector. This sector contains a list of track/sector pointer pairs which sequentially list the data sectors which make up the file. The file descriptive entry in the catalog sector points to this T/S List sector which, in turn, points to each sector in the file. This concept is diagramed in Figure 4.5.

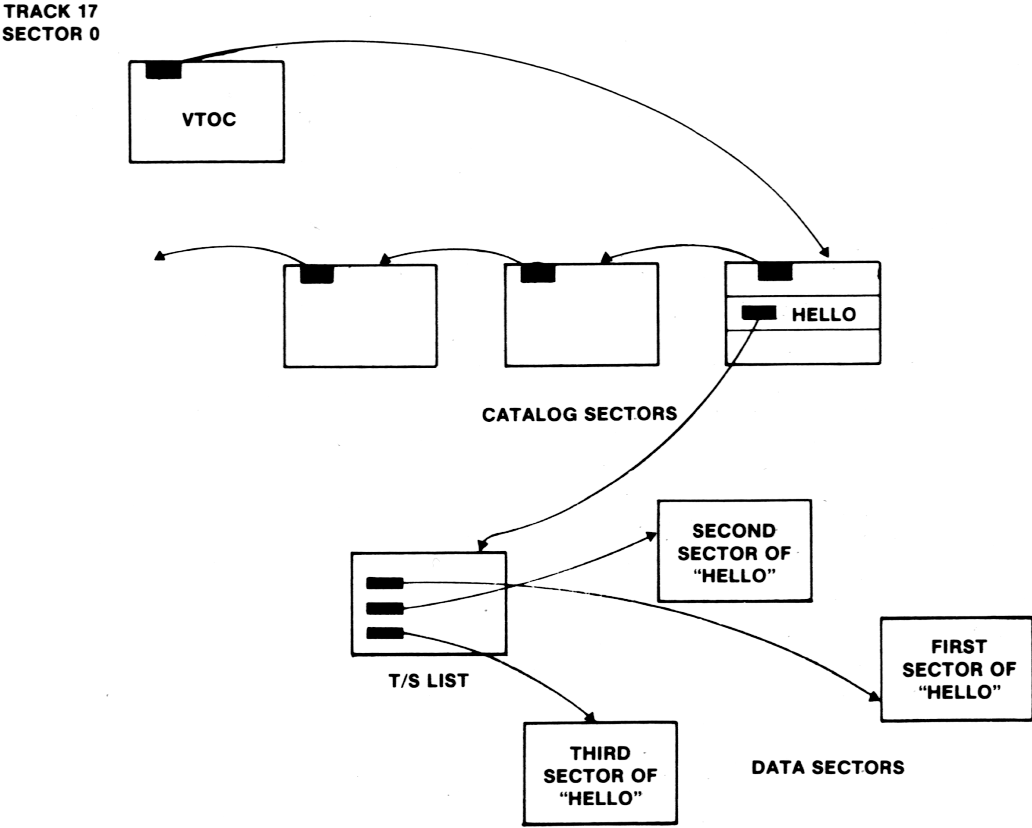


Figure 4.5 Path DOS Must Follow to Find a File

The format of a Track/Sector List sector is given below. Note that since even a minimal file requires one T/S List sector and one data sector, the least number of sectors a non-empty file can have is 2. Also, note that a very large file, having more than 122 data sectors, will need more than one Track/Sector List to hold all the Track/Sector pointer pairs.

TRACK/SECTOR LIST FORMAT

BYTE	DESCRIPTION
ØØ	Not used
Ø1	Track number of next T/S List sector if one was needed or zero if no more T/S List sectors.
Ø2	Sector number of next T/S List sector (if present).
Ø3-Ø4	Not used
Ø5-Ø6	Sector offset in file of the first sector described by this list.
Ø7-ØB	Not used

ØC-ØD Track and sector of first data sector or zeros
ØE-ØF Track and sector of second data sector or zeros
1Ø-FF Up to 12Ø more Track/Sector pairs

A sequential file will end when the first zero T/S List entry is encountered. A random file, however, can have spaces within it which were never allocated and therefore have no data sectors allocated in the T/S List. This distinction is not always handled correctly by DOS. The `VERIFY` command, for instance, stops when it gets to the first zero T/S List entry and can not be used to verify some random organization text files.

```
ØØ     ØØ                          Not used
Ø1     ØØ                          Track number of next T/S list (Ø = no more T/S lists)
Ø2     ØØ                          Sector number of next T/S list (Ø = no more T/S lists)
Ø3     ØØ ØØ                       Not used
Ø5     ØØ ØØ                       Sector offset of file in this T/S list (Ø = beginning of file)
Ø7     ØØ ØØ ØØ ØØ ØØ              Not used
ØC     13 ØE                       First Track and Sector of this file
ØE     13 ØD                       Second Track and Sector of this file
1Ø     13 ØC                       Third Track and Sector of this file
12     13 ØB                       Fourth Track and Sector of this file
14     13 ØA                       Final Track and Sector of this file
                                   Rest of list is zeroes (no more T/Ss)
16     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2Ø     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
3Ø     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
4Ø     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
5Ø     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
6Ø     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
7Ø     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
8Ø     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
9Ø     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
AØ     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
BØ     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
CØ     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
DØ     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
EØ     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
FØ     ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
```

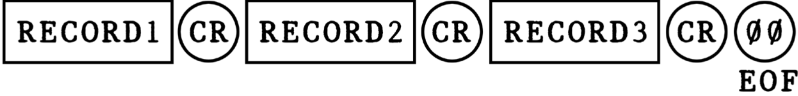
Figure 4.6 Example Track/Sector List

An example T/S List sector is given in Figure 4.6. The example file (`HELLO`, from our previous examples) has only one data sector, since it is less than 256 bytes in length. Counting this data sector and the T/S List sector, `HELLO` is 2 sectors long, and this will be the value shown when a `CATALOG` command is done.

Following the Track/Sector pointer in the T/S List sector, we come to the first data sector of the file. As we examine the data sectors, the differences between the file types become apparent. All files (except, perhaps, a random Text file) are considered to be continuous streams of data, even though they must be broken up into 256 byte chunks to fit in sectors on the diskette. Although these sectors are not necessarily contiguous (or next to each other on the diskette), by using the Track/Sector List, DOS can read each sector of

the file in the correct order so that the programmer need never know that the data was broken up into sectors at all.

Sequential Text File Format




```
ØØ   D2 C5 C3 CF D2 C4 B1 8D   R E C O R D 1 .
Ø8   D2 C5 C3 CF D2 C4 B2 8D   R E C O R D 2 .
1Ø   D2 C5 C3 CF D2 C4 B3 8D   R E C O R D 3 .
18   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
2Ø   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
28   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
3Ø   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
38   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
4Ø   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
48   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
5Ø   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
58   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
6Ø   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
68   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
7Ø   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
78   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
8Ø   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
88   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
9Ø   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
98   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
AØ   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
A8   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
BØ   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
B8   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
CØ   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
C8   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
DØ   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
D8   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
EØ   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
E8   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
FØ   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
F8   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   . . . . . . . .
```

Figure 4.7 Example Sequential Text File Sector

TEXT FILES

The Text data type is the least complicated file data structure. It consists of one or more records, separated from each other by carriage return characters (hex 8Ds). This structure is diagrammed and an example file is given in Figure 4.7. Usually, the end of a Text file is signaled by the presence of a hex ØØ or the lack of any more data sectors in the T/S List for the file. As mentioned earlier, if the file has random organization, there may be hex ØØs imbedded in the data and even missing data sectors in areas where nothing was ever written. In this case, the only way to find the end of the file is to scan the Track/Sector List for the last non-zero Track/Sector pair. Since carriage return characters and hex ØØs have special meaning in a Text type file, they can not be part of the data itself. For this reason, and to make the data accessible to BASIC, the data can only contain printable or ASCII characters (alphabetics, numerics

Offset	Bytes		Description
00	03	08	Program start address ($0803)
02	4E	12	File length ($124E = 4,686 bytes)

Rest of Sector is a Binary Memory Image

04	20	44	08	8D	94	13	A9	00	8D	AD	13	BA				
10	8E	1C	13	20	8B	08	20	DA	08	AD	AC	13	A2	16	20	2C
20	0B	D0	09	20	62	08	20	59	0D	4C	3E	08	20	F1	09	20
30	62	08	20	40	0B	AD	AB	13	D0	05	A2	0F	20	CD	0A	20
40	1D	0B	4C	0E	08	A9	80	85	76	85	D9	85	33	38	AD	D2
50	03	E9	07	E9	1E	30	01	60	20	58	FC	A2	15	20	CD	0A
60	4C	D3	03	A2	01	BD	1F	13	0A	0A	0A	0A	9D	21	13	CA
70	10	F3	20	DC	03	84	00	85	01	20	E3	03	84	02	85	03
80	20	2D	10	20	E8	0F	A9	00	8D	AB	13	60	A9	00	85	22
90	20	58	FC	A2	16	20	CD	0A	A2	17	20	CD	0A	20	8E	FD
A0	A0	00	A2	1F	20	CD	0A	B9	AF	13	20	ED	FD	A2	20	20
B0	CD	0A	98	18	69	28	AA	20	CD	0A	C8	C0	09	D0	E3	20
C0	8E	FD	A2	21	20	CD	0A	20	6F	FD	AD	00	02	A2	00	20
D0	2C	0B	D0	B8	8D	AC	13	8C	AE	13	60	20	58	FC	A9	0F
E0	85	24	AD	AE	13	18	69	28	AA	20	CD	0A	A9	03	85	22
F0	20	58	FC	AD	AC	13	A2	0A	20	2C	0B	D0	07	A9	00	8D

Figure 4.8 Example Binary File Sector

or special characters, see p. 8 in the *Apple II Reference Manual*) This restriction makes processing of a Text file slower and less efficient in the use of disk space than with a Binary type file, since each digit must occupy a full byte in the file.

BINARY FILES

The structure of a Binary type file is shown in Figure 4.8. An exact copy of the memory involved is written to the disk sector(s), preceded by the memory address where it was found and the length (a total of four bytes). The address and length (in low order, high order format) are those given in the `A` and `L` keywords from the `BSAVE` command which created the file. Notice that DOS writes one extra byte to the file. This does not matter too much since `BLOAD` and `BRUN` will only read the number of bytes given in the length field. (Of course, if you `BSAVE` a multiple of 256 bytes, a sector will be wasted because of this error) DOS could be made to `BLOAD` or `BRUN` the binary image at a different address either by providing the `A` (address) keyword when the command is entered, or by changing the address in the first two bytes of the file on the diskette.

APPLESOFT AND INTEGER FILES

A BASIC program, be it Applesoft or Integer, is saved to the diskette in a way that is similar to `BSAVE`. The format of an Applesoft file type is given in Figure 4.9 and that of Integer BASIC in 4.10. When the `SAVE` command is typed, DOS determines the location of the BASIC program image in memory and its length. Since a BASIC program is always loaded at a location known to the BASIC interpreter, it is not necessary to store the address in the file as with a Binary file. The length is stored, however, as the first two bytes, and is followed by the image from memory. Notice that, again, DOS incorrectly writes an additional byte, even though it will be ignored by `LOAD`. The memory image of the program consists of program lines in an internal format which is made up of what are called "tokens". A treatment of the structure of a BASIC program as it appears in memory is outside the scope of this manual, but a breakdown of the example Integer BASIC program is given in Figure 4.10.

OTHER FILE TYPES (S, R, NEW A, NEW B)

Additional file types have been defined within DOS as can be seen in the file descriptive entry format, shown earlier. No DOS commands at present use

```
        Applesoft File Format on Disk

   L/H     Program Memory Image...
```

```
1Ø PRINT "[CTRL-D]OPEN TEXT FILE"
2Ø PRINT "[CTRL-D]WRITE TEXT FILE"
3Ø PRINT "1,2,3,4"
4Ø PRINT "[CTRL-D]CLOSE TEXT FILE"
5Ø END
```

```
ØØ    5F ØØ     Program is $ØØ5F ( 95 ) bytes long

          Applesoft Program
Ø2    18 Ø8 ØA ØØ BA 22             . . . . : "
Ø8    Ø4 4F 5Ø 45 4E 2Ø 54 45       . O P E N   T E
1Ø    58 54 2Ø 46 49 4C 45 22       X T   F I L E "
18    ØØ 3Ø Ø8 14 ØØ BA 22 Ø4       . Ø . . . : " .
2Ø    57 52 49 54 45 2Ø 54 45       W R I T E   T E
28    58 54 2Ø 46 49 4C 45 22       X T   F I L E "
3Ø    ØØ 3F Ø8 1E ØØ BA 22 31       . ? . . . : " 1
38    2C 32 2C 33 2C 34 22 ØØ       , 2 , 3 , 4 " .
4Ø    57 Ø8 28 ØØ BA 22 Ø4 43       W . ( . : " . C
48    4C 4F 53 45 2Ø 54 45 58       L O S E   T E X
5Ø    54 2Ø 46 49 4C 45 22 ØØ       T   F I L E " .
58    5D Ø8 32 ØØ 8Ø ØØ ØØ ØØ       ] . 2 . . . . .
6Ø    ØA 45 ØØ ØØ ØØ ØØ ØØ ØØ       . E . . . . . .
68    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ       . . . . . . . .
7Ø    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ       . . . . . . . .
78    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ       . . . . . . . .
                  |
                  v
F8    ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ       . . . . . . . .
```

Figure 4.9 Example Applesoft File Sector

Integer BASIC File Format on Disk

L/H	Program Memory Image...

1Ø END

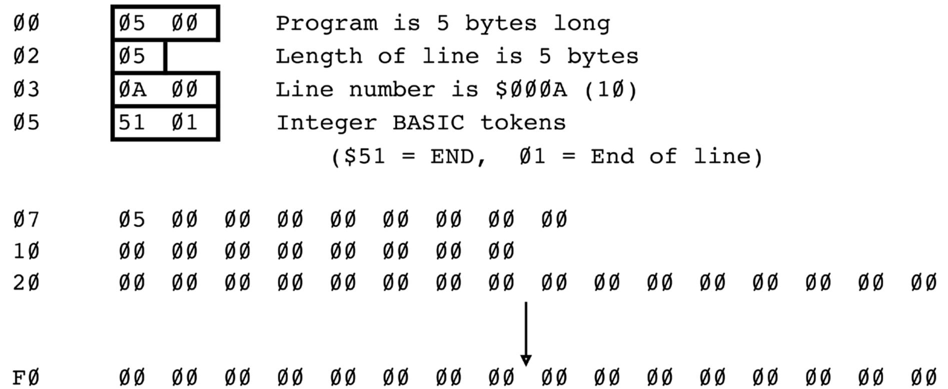


Figure 4.10 Example Integer BASIC File Sector

these additional types so their eventual meaning is anybody's guess. The R file type, however, has been used with the DOS Toolkit assembler for its output file, a relocatable object module. This file type is used with a special form of Binary file which can contain the memory image of a machine language program which may be relocated anywhere in the machine based on additional information stored with the image itself. The format for this type of file is given in the documentation accompanying the DOS Toolkit. It is recommended that if the reader requires more information about R files he should refer to that documentation.

EMERGENCY REPAIRS

When you forget to backup your disks

From time to time the information on a diskette can become damaged or lost. This can create various symptoms, ranging from mild side effects, such as the disk not booting, to major problems, such as an input/output (I/O) error in the catalog. A good understanding of the format of a diskette, as described previously, and a few program tools can allow any reasonably sharp Apple II user to patch up most errors on diskettes.

A first question would be, "how do errors occur?". The most common cause of an error is a worn or physically damaged diskette. Usually, a diskette will warn you that it is wearing out by producing "soft errors". Soft errors are I/O errors which occur only randomly. You may get an I/O error message when you catalog a disk one time and have it catalog correctly if you try again. When this happens, the smart programmer immediately copies the files on the aged diskette to a brand new one and discards the old one or keeps it as a backup.

Another cause of damaged diskettes is the practice of hitting the RESET key to abort the execution of a program which is accessing the diskette. Damage will usually occur when the RESET signal comes just as data is being written onto the disk. Powering the machine off just as data is being written to the disk is also a sure way to clobber a diskette. Of course, real hardware problems in the disk drive or controller card and ribbon cable can cause damage as well.

If the damaged diskette can be cataloged, recovery is much easier. A damaged DOS image in the first three tracks can usually be corrected by running the `MASTER CREATE` program against the diskette or by copying all the files to another diskette. If only one file produces an I/O error when it is `VERIFY`ed, it may be possible to copy most of the sectors of the file to another diskette by skipping over the bad sector with an assembler program which calls RWTS in DOS or with a BASIC program (if the file is a Text file). Indeed, if the problem is a bad checksum (see Chapter 3) it may be possible to read the bad sector and ignore the error and get most of the data.

An I/O error usually means that one of two conditions has occured. Either a bad checksum was detected on the data in a sector, meaning that one or more bytes is bad; or the sectoring is clobbered such that the sector no longer even exists on the diskette. If the latter is the case, the diskette (or at the very least, the track) must be reformatted, resulting in a massive loss of data. Although DOS can be patched to format a single track, it is usually easier to copy all readable sectors from the damaged diskette to another formatted diskette and then reconstruct the lost data there.

When you know you made a backup

Disk utilities, such as Quality Software's *Bag of Tricks*, allow the user to read and display the contents of sectors. *Bag of Tricks* will also allow you to modify the sector data and rewrite it to the same or another diskette. If you do not have *Bag of Tricks* or another commercial disk utility, you can use the `ZAP` program in Appendix A of this book. The `ZAP` program will read any track/sector on an unprotected diskette into memory, allowing the user to examine it or modify the data and then, optionally, rewrite it to a diskette. Using such a program is very important when learning about diskette formats and when fixing clobbered data.

Using `ZAP`, a bad sector within a file can be localized by reading each track/sector listed in the T/S List sector for the file. If the bad sector is a catalog sector, the pointers of up to seven files may be lost. When this occurs, a search of the diskette can be made to find T/S List sectors which do not correspond to any files listed in the remaining "good" catalog sectors. As these sectors are found, new file descriptive entries can be made in the damaged sector which point to these T/S Lists. When the entire catalog is lost, this process can take hours, even with a good understanding of the format of DOS diskettes. Such an endeavor should only be undertaken if there is no other way to recover the data. Of course the best policy is to create backup copies of important files periodically to simplify recovery. More information on the above procedures is given in Appendix A.

A less significant form of diskette clobber, but very annoying, is the loss of free sectors. Since DOS allocates an entire track of sectors at a time while a file is open, hitting RESET can cause these sectors to be marked in use in

the VTOC even though they have not yet been added to any T/S List. These lost sectors can never be recovered by normal means, even when the file is deleted, since they are not in its T/S List. The result is a `DISK FULL` message before the diskette is actually full. To reclaim the lost sectors it is necessary to compare every sector listed in every T/S List against the VTOC bit map to see if there are any discrepancies. There are utility programs which will do this automatically but the best way to solve this problem is to copy all the files on the diskette to another diskette (note that `FID` must be used, not `COPY`, since `COPY` copies an image of the diskette, bad VTOC and all).

If a file is deleted it can usually be recovered, providing that additional sector allocations have not occured since it was deleted. If another file was created after the `DELETE` command, DOS might have reused some or all of the sectors of the old file. The catalog can be quickly `ZAP`ped to move the track number of the T/S List from byte 20 of the file descriptive entry to byte 0. The file should then be copied to another disk and then the original deleted so that the VTOC freespace bit map will be updated.

`Freshly initialized disks are the best`

Chapter 5

The Structure of Dos

DOS MEMORY USE

DOS is an assembly language program which is loaded into RAM memory when the user boots his disk. If the diskette booted is a master diskette, the DOS image is loaded into the last possible part of RAM memory, dependent upon the size of the actual machine on which it is run. By doing this, DOS fools the active BASIC into believing that there is actually less RAM memory on the machine than there is. On a 48K Apple II with DOS active, for instance, BASIC believes that there is only about 38K of RAM. DOS does this by adjusting HIMEM after it is loaded to prevent BASIC from using the memory DOS is occupying. If a slave diskette is booted, DOS is loaded into whatever RAM it occupied when the slave diskette was `INIT`ialized. If the slave was created on a 16K Apple, DOS will be loaded in the 6 to 16K range of RAM, even if the machine now has 48K. In this case, the Apple will appear, for all intents an purposes, to have only 6K of RAM. If the slave was created on a 48K system, it

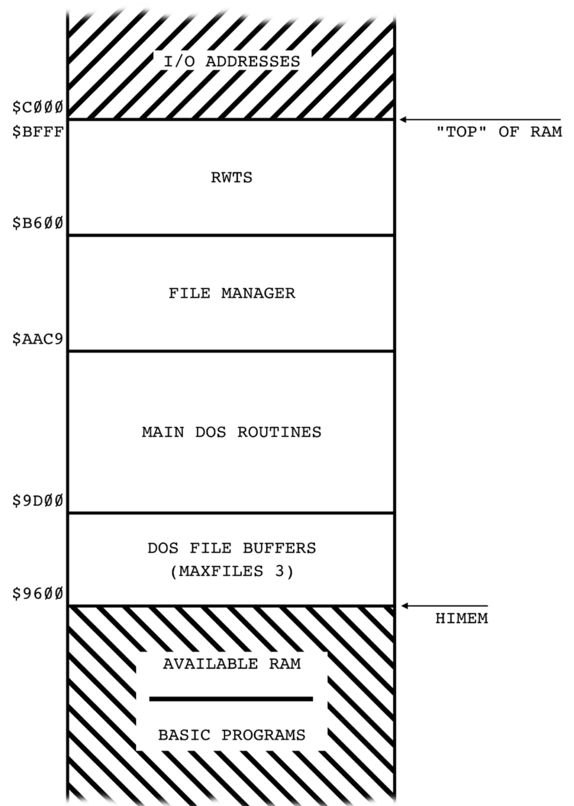


Figure 5.1 DOS Memory Use (48K Apple)

will not boot on less than 48K since the RAM DOS occupied does not exist on a smaller machine.

A diagram of DOS's memory for a 48K Apple II is given in Figure 5.1. As can be seen, there are four major divisions to the memory occupied by DOS. The first 1.75K is used for file buffers. With the default of `MAXFILES 3`, there are three file buffers set aside here. Each buffer occupies 595 bytes and corresponds to one potentially open file. File buffers are also used by DOS to `LOAD` and `SAVE` files, etc. If `MAXFILES` is changed from 3, the space occupied by the file buffers also changes. This affects the placement of HIMEM, moving it up or down with fewer or more buffers respectively.

The 3.5K above the file buffers is occupied by the main DOS routines. It is here that DOS's executable machine language code begins. The main routines are responsible for initializing DOS, interfacing to BASIC, interpreting commands, and managing the file buffers. All disk functions are passed on via subroutine calls to the file manager.

The file manager, occupying about 2.8K, is a collection of subroutines which perform almost any function needed to access a disk file. Functions include: `OPEN`, `CLOSE`, `READ`, `WRITE`, `POSITION`, `DELETE`, `CATALOG`, `LOCK`, `UNLOCK`, `RENAME`, `INIT`, and `VERIFY`. Although the file manager is a subroutine of DOS it may also be called by a user written assembly lanaguage program which is not part of DOS. This interface is generalized through a group of vectors in page 3 of RAM and is documented in the next chapter.

The last 2.5K of DOS is the Read/Write Track/Sector (RWTS) package. RWTS is the next step lower in protocol from the file manager - in fact it is called as a subroutine by the file manager. Where the file manager deals with files, RWTS deals with tracks and sectors on the diskette. A typical call to RWTS would be to read track 17 sector 0 or to write 256 bytes of data in memory onto track 5 sector E. An external interface is also provided for access to RWTS from a user written assembly language program and is described in the next chapter.

THE DOS VECTORS IN PAGE 3

In addition to the approximately 10K of RAM occupied by DOS in high memory, DOS maintains a group of what are called "vectors" in page 3 of low memory (`$3ØØ` through `$3FF`). These vectors allow access to certain places within the DOS collection of routines via a fixed location (`$3DØ` for instance). Because DOS may be loaded in various locations, depending upon the size of the machine and whether a slave or master diskette is booted, the addresses

of the externally callable subroutines within DOS will change. By putting the addresses of these routines in a vector at a fixed location, dependencies on DOS's location in memory are eliminated. The page 3 vector table is also useful in locating subroutines within DOS which may not be in the same memory location for different versions of DOS. Locations `$3ØØ` through `$3CF` were used by earlier versions of DOS during the boot process to load the Boot 1 program but are used by DOS 3.3 as a data buffer and disk code translate table. Presumably, this change was made to provide more memory for the first bootstrap loader (more on this later). The vector table itself starts at `$3DØ`.

DOS VECTOR TABLE ($3D0-$3FF)

```
ADDR  USAGE
3DØ   A JMP (jump or GOTO) instruction to the DOS warmstart
      routine. This routine reenters DOS but does not
      discard the current BASIC program and does not reset
      MAXFILES or other DOS environmental variables.
3D3   A JMP to the DOS coldstart routine. This routine
      reinitializes DOS as if it was rebooted, clearing the
      current BASIC file and resetting HIMEM.
3D6   A JMP to the DOS file manager subroutine to allow a
      user written assembly language program to call it.
3D9   A JMP to the DOS Read/Write Track/Sector (RWTS)
      routine to allow user written assembly language
      programs to call it.
3DC   A short subroutine which locates the input parameter
      list for the file manager to allow a user written
      program to set up input parameters before calling the
      file manager.
3E3   A short subroutine which locates the input parameter
      list for RWTS to allow a user written program to set
      up input parameters before calling RWTS.
3EA   A JMP to the DOS subroutine which "reconnects" the DOS
      intercepts to the keyboard and screen data streams.
3EF   A JMP to the routine which will handle a BRK machine
      language instruction. This vector is only supported by
      the Autostart ROM. Normally the vector contains the
      address of the monitor ROM subroutine which displays
      the registers.
3F2   LO/HI address of routine which will handle RESET for
      the Autostart ROM. Normally the DOS restart address is
      stored here but the user may change it if he wishes to
      handle RESET himself.
3F4   Power-up byte. Contains a "funny complement" of the
      RESET address with a $A5. This scheme is used to
      determine if the machine was just powered up or if
      RESET was pressed.  If a power-up occured, the
      Autostart ROM ignores the address at 3F2 (since it has
      never been initialized) and attempts to boot a
```

```
      diskette.  To prevent this from happening when you
      change $3F2 to handle your own RESETs, EOR (exclusive
      OR) the new value at $3F2 with a $A5 and store the
      result in the power-up byte.
3F5   A JMP to a machine language routine which is to be
      called when the '&' feature is used in Applesoft.
3F8   A JMP to a machine language routine which is to be
      called when a control-Y is entered from the monitor.
3FB   A JMP to a machine language routine which is to be
      called when a non-maskable interrupt occurs.
3FE   LO/HI address of a routine which is to be called when
      a maskable interrupt occurs.
```

WHAT HAPPENS DURING BOOTING

When an Apple is powered on its memory is essentially devoid of any programs. In order to get DOS running, a diskette is "booted". The term "boot" refers to the process of bootstrap loading DOS into RAM. Bootstrap loading involves a series of steps which load successively bigger pieces of a program until all of the program is in memory and is running. In the case of DOS, bootstrapping occurs in four stages. The location of these stages on the diskette and a memory map are given in Figure 5.2 and a description of the bootstrap process follows.

The first boot stage (let's call it Boot 0) is the execution of the ROM on the disk controller card. When the user types `PR#6` or `C600G` or `6(ctrl)P`, for instance, control is transfered to the disk controller ROM on the card in slot 6. This ROM is a machine language program of about 256 bytes in length. When executed, it "recalibrates" the disk arm by pulling it back to track 0 (the "clacketty-clack" noise that is heard) and then reads sector 0 from track 0 into RAM memory at location `$800` (DOS 3.3. Earlier versions used `$300`). Once this sector is read, the first stage boot jumps (GOTO's) `$800` which is the second stage boot (Boot 1).

Boot 1, also about 256 bytes long, uses part of the Boot 0 ROM as a subroutine and, in a loop, reads the next nine sectors on track 0 (sectors 1 through 9) into RAM. Taken together, these sectors contain the next stage of the bootstrap process, Boot 2. Boot 2 is loaded in one of two positions in memory, depending upon whether a slave or a master diskette is being booted. If the diskette is a slave diskette, Boot 2 will be loaded 9 pages (256 bytes per page) below the end of the DOS under which the slave was `INIT`ed. Thus, if the slave was created on a 32K DOS, Boot 2 will be loaded in the RAM from `$7700` to `$8000`. If a master diskette is being booted, Boot 2 will be loaded in the same place as for a 16K slave (`$3700` to `$4000`). In the process of

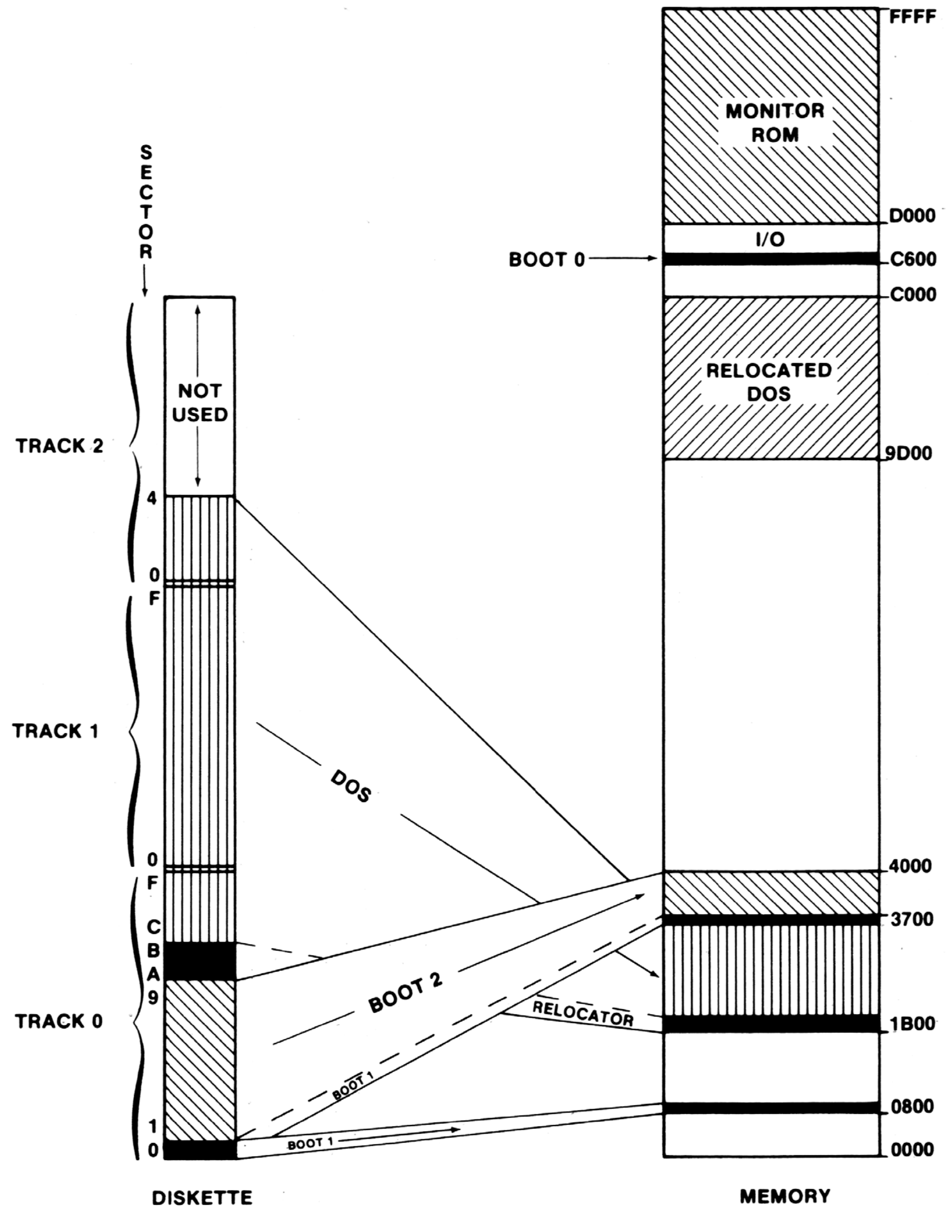


Figure 5.2 Bootstrap Process

loading Boot 2, Boot 1 is loaded a second time in the page in memory right below Boot 2 ($36ØØ for a master diskette). This is so that, should a new diskette be INITed, a copy of Boot 1 will be available in memory to be written to its track 0 sector 0. When Boot 1 is finished loading Boot 2, it jumps there to begin execution of the next stage of the bootstrap.

Boot 2 consists of two parts: a loader "main program"; and the RWTS subroutine package. Up to this point there has been no need to move the disk arm since all of the necessary sectors have been on track 0. Now, however, more sectors must be loaded, requiring arm movement to access additional tracks. Since this complicates the disk access, RWTS is called by the Boot 2 loader to move the arm and read the sectors it needs to load the last part of the bootstrap, DOS itself. Boot 2 now locates track 2 sector 4 and reads its contents into RAM just below the image of Boot 1 (this would be at $3500 for a master diskette). In a loop, Boot 2 reads 26 more sectors into memory, each one 256 bytes before the last. The last sector (track 0 sector A) is read into $1B00 for a master diskette. The 27 sectors which were read are the image of the DOS main routines and the file manager. With the loading of these routines, all of DOS has been loaded into memory. At this point, the bootstrap process for a slave diskette is complete and a jump is taken to the DOS coldstart address. If the diskette is a master, the image of DOS is only valid if the machine is a 16K Apple II. If more memory is present, the DOS image must be relocated into the highest possible RAM present in the machine. To do this, the master version of Boot 2 jumps to a special relocation program at $1B03. This relocator is 512 bytes in length and was automatically loaded as the two lowest pages of the DOS image. (In the case of a slave diskette, these pages contain binary zeros.) The relocator determines the size of the machine by systematically storing and loading on high RAM memory pages until it finds the last valid page. It then moves the DOS image from $1D00 to its final location ($9D00 for 48K) and, using tables built into the program, it modifies the machine language code so that it will execute properly at its new home. The relocator then jumps to the high memory copy of DOS and the old image is forgotten.

The DOS boot is completed by the DOS coldstart routine. This code initializes DOS, making space for the file buffers, setting HIMEM, building the page 3 vector table, and running the HELLO program.

Previous versions of DOS were somewhat more complicated in the implementation of the bootstrap. In these versions, Boot 1 was loaded at $300 and it, in turn, loaded Boot 2 at $3600, as does version 3.3. Unlike 3.3, however, 27 sectors of DOS were not always loaded. If the diskette was a slave diskette, only 25 sectors were loaded, and, on 13 sector diskettes, this meant the DOS image ended either with sector 8 or sector A of track 2 depending upon whether the diskette was a slave or master. In addition, Boot 1 had a different form of nibbilization (see chapter 3) than any other sector on the diskette, making its raw appearance in memory at $3600 non-executable.

The various stages of the bootstrap process will be covered again in greater detail in Chapter 8, *DOS Program Logic*.

You gotta tell me: how did you boot from the external //c drive?

Let me get this straight: you have a Floppy Emu and a Booti card?

Chapter 6

Using DOS from Assembly Language

CAVEAT

This chapter is aimed at the advanced assembly language programmer who wishes to access the disk without resorting to the `PRINT` statement scheme used with BASIC. Accordingly, the topics covered here may be beyond the comprehension (at least for the present) of a programmer who has never used assembly language.

DIRECT USE OF DISK DRIVE

It is often desirable or necessary to access the Apple's disk drives directly from assembly language, without the use of DOS. This is done using a section of 16 addresses that are latched toggles, interfacing directly to the hardware. There are eight two byte toggles that essentially represent pulling a TTL line high or low. Applications which could use direct disk access range from a user written operating system to DOS-independent utility programs. The device address assignments are given in Figure 6.1.

ADDRESS	LABEL	DESCRIPTION
$CØ8Ø	PHASEOFF	Stepper motor phase Ø off.
$CØ81	PHASEON	Stepper motor phase Ø on.
$CØ82	PHASE1OFF	Stepper motor phase 1 off.
$CØ83	PHASE1ON	Stepper motor phase 1 on.
$CØ84	PHASE2OFF	Stepper motor phase 2 off.
$CØ85	PHASE2ON	Stepper motor phase 2 on.
$CØ86	PHASE3OFF	Stepper motor phase 3 off.
$CØ87	PHASE3ON	Stepper motor phase 3 on.
$CØ88	MOTOROFF	Turn motor off.
$CØ89	MOTORON	Turn motor on.
$CØ8A	DRVØEN	Engage drive 1.
$CØ8B	DRV1EN	Engage drive 2.
$CØ8C	Q6L	Strobe Data Latch for I/O.
$CØ8D	Q6H	Load Data Latch.

```
$CØ8E  Q7L            Prepare latch for input.
$CØ8F  Q7H            Prepare latch for output.

       Q7L followed by Q6L = Read
       Q7L followed by Q6H = Sense Write Protect
       Q7H followed by Q6L = Write
       Q7H followed by Q6H = Load Write Latch
```

FIGURE 6.1 DOS HARDWARE ADDRESSES

The addresses are slot dependent and the offsets are computed by multiplying the slot number by 16. In hexadecimal this works out nicely and we can add the value `$sØ` (where s is the slot number) to the base address. If we wanted to engage disk drive number 1 in slot number 6, for example, we would add `$6Ø` to `$CØ8A` (device address assignment for engaging drive 1) for a result of `$CØEA`. However, since it is generally desirable to write code that is not slot dependent, one would normally use `$CØ8A,X` (where the X register contains the value `$sØ`).

In general, the above addresses need only be accessed with any valid 6502 instruction. However, in the case of reading and writing bytes, care must be taken to insure that the data will be in an appropriate register. All of the following would engage drive number 1. (Assume slot number 6)

```
LDA $CØEA
BIT $CØ8A,X (where X-reg contains $6Ø)
CMP $CØ8A,X (where X-reg contains $6Ø)
```

Below are typical examples demonstrating the use of the device address assignments. For more examples, see Appendix A. Slot 6 is assumed and the X-register contains `$6Ø`.

STEPPER PHASE OFF/ON

Basically, each of the four phases (0-3) must be turned on and then off again. Done in ascending order, this moves the arm inward. In descending order, this moves the arm outward. The timing between accesses to these locations is critical, making this a non-trivial exercise. It is recommended that the `SEEK` command in RWTS be used to move the arm. See the section on using RWTS immediately following.

MOTOR OFF/ON

```
        LDA $CØ88,X     Turn motor off.
        LDA $CØ89,X     Turn motor on.
```

NOTE: A sufficient delay should be provided to allow the motor time to come up to speed. Shugart recommends one second, but DOS is able to reduce this delay by watching the read latch until data starts to change.

ENGAGE DRIVE 1/2

```
        LDA $CØ8A,X     Engage drive 1.
        LDA $CØ8B,X     Engage drive 2.
```

READ A BYTE

```
READ    LDA $CØ8C,X
        BPL READ
```

NOTE: `$CØ8E,X` must already have been accessed to assure Read mode. The loop is necessary to ensure that the accumulator will contain valid data. If the data latch does not yet contain valid data the high bit will be zero.

SENSE WRITE PROTECT

```
        LDA $CØ8D,X
        LDA $CØ8E,X     Sense write protect.
        BMI ERROR       If high bit set, protected.
```

LOAD AND WRITE A BYTE

```
        LDA DATA
        STA $CØ8D,X     Write load.
        ORA $CØ8C,X     Write byte.
```

NOTE: `$CØ8F,X` must already have been accessed to ensure Write mode and a 100 microsecond delay should be invoked before writing.

Due to hardware constraints, data bytes must be written in 32 cycle loops. Below is an example for an immediate load of the accumulator, followed by a write. Timing is so critical that different routines may be necessary, depending on how the data is to be accessed, and code can not cross memory page boundaries without an adjustment.

```
        LDA #$D5        (2 cycles)
        JSR WRITE9      (6)
        LDA #$AA        (2)
        JSR WRITE9      (6)
        .
        .
        .
WRITE9  CLC             (2)
WRITE7  PHA             (3)
        PLA             (4)
WRITE   STA $CØ8D,X     (5)
        ORA $CØ8C,X     (4)
        RTS             (6)
```

CALLING READ/WRITE TRACK/SECTOR (RWTS)

Read/Write Track/Sector (RWTS) exists in every version of DOS as a collection of subroutines, occupying roughly the top third of the DOS program. The interface to RWTS is standardized and thoroughly documented by Apple and may be called by a program running outside of DOS.

There are two subroutines which must be called or whose function must be performed.

`JSR $3E3` When this subroutine is called, the Y and A registers are loaded with the address of the Input/Output control Block (IOB) used by DOS when accessing RWTS. The low order part of the address is in Y and the high order part in A. This subroutine should be called to locate the IOB and the results may be stored in two zero page locations to allow storing values in the IOB and retrieving output values after a call to RWTS. Of course, you may set up your own IOB as long as the Y and A registers point to your IOB upon calling RWTS.

`JSR $3D9` This is the main entry to the RWTS routine. Prior to making this call, the Y and A registers must be loaded with the address of an IOB describing the operation to be performed. This may be done by first calling `$3E3` as described above. The IOB must contain appropriate information as defined in the list on the facing page (offsets are given in hexadecimal):

INPUT/OUTPUT CONTROL BLOCK - GENERAL FORMAT

```
BYTE     DESCRIPTION
ØØ       Table type, must be $Ø1
Ø1       Slot number times 16 (sØ: s=slot.  Example: $6Ø)
Ø2       Drive number ($Ø1 or $Ø2)
Ø3       Volume number expected ($ØØ matches any volume)
Ø4       Track number ($ØØ through $22)
Ø5       Sector number ($ØØ through $ØF)
Ø6-Ø7    Address (LO/HI) of the Device Characteristics Table
Ø8-Ø9    Address (LO/HI) of the 256 byte buffer for
         READ/WRITE
ØA       Not used
ØB       Byte count for partial sector ($ØØ for 256 bytes)
ØC       Command code    $ØØ = SEEK
                         $Ø1 = READ
                         $Ø2 = WRITE
                         $Ø4 = FORMAT
ØD       Return code - The processor CARRY flag is set upon
                       return from RWTS if there is a
                       non-zero return code:
                         $ØØ = No errors
                         $Ø8 = Error during initialization
                         $1Ø = Write protect error
                         $2Ø = Volume mismatch error
                         $4Ø = Drive error
                         $8Ø = Read error (obsolete)
ØE       Volume number of last access (must be initialized)
ØF       Slot number of last access * 16 (must be initialized)
1Ø       Drive number of last access (must be initialized)
DEVICE CHARACTERISTICS TABLE
BYTE     DESCRIPTION
ØØ       Device type (should be $ØØ for Disk II)
Ø1       Phases per track (should be $Ø1 for Disk II)
Ø2-Ø3    Motor on time count (should be $EFD8 for Disk II)
```

NOTE: RWTS uses zero-page location `$48`, which is also used by the Apple monitor to hold the P-register value. Location `$48` should be set to zero after each call to RWTS.

RWTS IOB BY CALL TYPE

```
SEEK     Move disk arm to desired track
Input:   Byte ØØ     - Table type ($Ø1)
              Ø1     - Slot number * 16 (sØ: s=slot)
              Ø2     - Drive number ($Ø1 or $Ø2)
              Ø4     - Track number ($ØØ through $22)
```

```
                 Ø6/Ø7 - Pointer to the DCT
                 ØC    - Command code for SEEK ($ØØ)
                 ØF    - Slot number of last access * 16
                 1Ø    - Drive number of last access
Output: Byte ØD    - Return code (See previous definition)
                 ØF    - Current Slot number * 16
                 1Ø    - Current Drive number
--------------------------------------------------------------------
READ    Read a sector into a specified buffer
Input:  Byte ØØ    - Table type ($Ø1)
             Ø1    - Slot number * 16 (sØ: s=slot)
             Ø2    - Drive number ($Ø1 or $Ø2)
             Ø3    - Volume number ($ØØ matches any volume)
             Ø4    - Track number ($ØØ through $22)
             Ø5    - Sector number ($ØØ through $ØF)
             Ø6/Ø7 - Pointer to the DCT
             Ø8/Ø9 - Pointer to 256 byte user data buffer
             ØB    - Byte count per sector ($ØØ)
             ØC    - Command code for READ ($Ø1)
             ØE    - Volume number of last access
             ØF    - Slot number of last access * 16
             1Ø    - Drive number of last access
Output: Byte ØD    - Return code (See previous definition)
             ØE    - Current Volume number
             ØF    - Current Slot number * 16
             1Ø    - Current Drive number
--------------------------------------------------------------------
WRITE   Write a sector from a specified buffer
Input:  Byte ØØ    - Table type ($Ø1)
             Ø1    - Slot number * 16 (sØ: s=slot)
             Ø2    - Drive number ($Ø1 or $Ø2)
             Ø3    - Volume number ($ØØ matches any volume)
             Ø4    - Track number ($ØØ through $22)
             Ø5    - Sector number ($ØØ through $ØF)
             Ø6/Ø7 - Pointer to the DCT
             Ø8/Ø9 - Pointer to 256 byte user data buffer
             ØB    - Byte count per sector ($ØØ)
             ØC    - Command code for WRITE ($Ø2)
             ØE    - Volume number of last access
             ØF    - Slot number of last access * 16
             1Ø    - Drive number of last access
Output: Byte ØD    - Return code (See previous definition)
             ØE    - Current Volume number
             ØF    - Current Slot number * 16
             1Ø    - Current Drive number
--------------------------------------------------------------------
FORMAT  Initialize the diskette (does not put DOS on disk, create
        a VTOC/CATALOG, or store HELLO program)
Input:  Byte ØØ    - Table type ($Ø1)
             Ø1    - Slot number * 16 (sØ: s=slot)
             Ø2    - Drive number ($Ø1 or $Ø2)
             Ø3    - Volume number ($ØØ will default to 254)
```

```
                Ø6/Ø7 - Pointer to the DCT
                ØC    - Command code for FORMAT ($Ø4)
                ØE    - Volume number of last access
                ØF    - Slot number of last access * 16
                1Ø    - Drive number of last access
Output: Byte    ØD    - Return code (See previous definition)
                ØE    - Current Volume number
                ØF    - Current Slot number * 16
                1Ø    - Current Drive number
```

CALLING THE DOS FILE MANAGER

The DOS file manager exists in every version of DOS as a collection of subroutines occupying approximately the central third of the DOS program. The interface to these routines is generalized in such a way that they may be called by a program running outside of DOS. The definition of this interface has never been published by Apple (or anyone else, for that matter) but since the calls can be made through fixed vectors, and, the format of the parameter lists passed have not changed in all the versions of DOS, these routines may be relied upon as "safe". Indeed, the new `FID` utility program uses these routines to process files on the diskette.

There are two subroutines which must be called in order to access the file manager.

`JSR $3DC` When this subroutine is called, the Y and A registers are loaded with the address of the file manager parameter list. The low order part of the address is in Y and the high order part in A. This subroutine must be called at least once to locate this parameter list and the results may be stored in two zero page locations to allow the programmer to set input values in the parameter list and to locate output values there after file manager calls.

`JSR $3D6` This is the main entry to the file manager. Prior to making this call the parameter list, located using the call described above, must be completed appropriately, depending upon the type of call, and the X register must be set to either zero or non-zero as follows:

```
X = Ø - If file is not found, allocate it
X # Ø - If file is not found, do not allocate one
```

Normally, X should be zero on an `OPEN` call for a new file and non-zero for all other call types.

Three buffers must be provided to the file manager by the programmer, allocated in memory. These buffers, together, occupy 557 bytes of RAM, and

	Ø0	Ø1	Ø2	Ø3	Ø4	Ø5	Ø6	Ø7	Ø8	Ø9	ØA	ØB	ØC	ØD	ØE	ØF	1Ø	11
OPEN	Ø1		RECORD LENGTH OR ØØØØ		V	D	S	FILE TYPE	FILE NAME ADDRESS		RETURN CODE (OUTPUT)		ADDRESS OF FILE MANAGER WORK AREA BUFFER		ADDRESS OF T/S LIST BUFFER			
CLOSE	Ø2																DATA SECTOR BUFFER ADDRESS	
READ	Ø3	SUBCODE	RECORD NUMBER		BYTE OFFSET		RANGE LENGTH		ONE DATA BYTE	RANGE ADDR.								
WRITE	Ø4																	
DELETE	Ø5				V	D	S		FILE NAME ADDRESS									
CATALOG	Ø6					D	S											
LOCK	Ø7				V	D	S		FILE NAME ADDRESS						T/S LIST ADDRESS			
UNLOCK	Ø8				V	D	S											
RENAME	Ø9		NEW NAME ADDRESS		V	D	S											
POSITION	ØA		RECORD NUMBER		BYTE OFFSET													
INIT	ØB	DOS PAGE NO.			V	D	S											
VERIFY	ØC				V	D	S		FILE NAME ADDRESS						T/S LIST ADDRESS		DATA BUFF. ADDRESS	

Figure 6.2 File Manager Parameter List Required Input

must be passed to the file manager each time their associated file is used. A separate set of these buffers must be maintained for each open file. DOS maintains buffers for this purpose, as described in earlier chapters, in high RAM. These buffers may be "borrowed" from DOS if care is taken to let DOS know about it. A method for doing this will be outlined later.

The general format of the file manager parameter list is as follows:

FILE MANAGER PARAMETER LIST - GENERAL FORMAT

```
BYTE  DESCRIPTION
ØØ    Call type: Ø1=OPEN      Ø5=DELETE       Ø9=RENAME
                 Ø2=CLOSE     Ø6=CATALOG      ØA=POSITION
                 Ø3=READ      Ø7=LOCK         ØB=INIT
                 Ø4=WRITE     Ø8=UNLOCK       ØC=VERIFY
Ø1    Sub-call type for READ or WRITE:
                 ØØ=No operation (ignore call entirely)
                 Ø1=READ or WRITE one byte
                 Ø2=READ or WRITE a range of bytes
                 Ø3=POSITION then READ or WRITE one byte
                 Ø4=POSITION then READ/WRITE a range
Ø2-Ø9 Parameters specific to the call type used. See
      FILE MANAGER PARAMETER LIST BY CALL TYPE below.
ØA    Return code (note: not all return codes can occur
                 for any call type). The processor CARRY
                 flag is set upon return from the file
                 manager if there is a non-zero return code:
                 ØØ=No errors
                 Ø1=Not used ("LANGUAGE NOT AVAILABLE")
                 Ø2=Bad call type
                 Ø3=Bad sub-call type (greater than four)
                 Ø4=WRITE PROTECTED
                 Ø5=END OF DATA
                 Ø6=FILE NOT FOUND (was allocated if X=Ø)
                 Ø7=VOLUME MISMATCH
                 Ø8=DISK I/O ERROR
                 Ø9=DISK FULL
                 ØA=FILE LOCKED
ØB    Not used
ØC-ØD Address of a 45 byte buffer which will be used by the
      file manager to save its status between calls. This
      area is called the file manager workarea and need not
      be initialized by the caller but the space must be
      provided and this two byte address field initialized.
      (addresses are in low/high order format)
ØE-ØF Address of a 256 byte buffer which will be used by the
      file manager to maintain the current Track/Sector List
      sector for the open file. Buffer itself need not be
```

```
      initialized by the caller.
10-11 Address of a 256 byte buffer which will be used by the
      file manager to maintain the data sector buffer.
      Buffer need not be initialized by the caller.
```

FILE MANAGER PARAMETER LIST BY CALL TYPE

```
OPEN    Locates or creates a file. A call to POSITION should follow every
        OPEN.
        Input:  Byte 00    - 01
                     02/03 - Fixed record length or 0000 if variable
                     04    - Volume number or 00 for any volume
                     05    - Drive number to be used (01 or 02)
                     06    - Slot number to be used (01-07)
                     07    - File type (used only for new files)
                             $00 = TEXT
                             $01 = INTEGER BASIC
                             $02 = APPLESOFT BASIC
                             $04 = BINARY
                             $08 = RELOCATABLE
                             $10 = S TYPE FILE
                             $20 = A TYPE FILE
                             $40 = B TYPE FILE
                     08/09 - Address of file name (30 characters)
                             (Low/high format)
                     0C/0D - Address of file manager workarea buffer
                     0E/0F - Address of T/S List sector buffer
                     10/11 - Address of data sector buffer
        Output: Byte 07    - File type of file which was OPENed
                     0A    - Return code (see previous definitions)

CLOSE   Write out final sectors, update the Catalog. A CLOSE call is
        required eventually for every OPEN.
        Input:  Byte 00    - 02
                     0C/0D - Address of file manager workarea buffer
                     0E/0F - Address of T/S List sector buffer
                     10/11 - Address of data sector buffer
        Output: Byte 0A    - Return code

READ    Read one or a range of bytes from the file to memory.
WRITE   Write one or a range of bytes from memory to the file.
        Input:  Byte 00    - 03 (READ) 04 (WRITE)
                     01    - Subcode:
                             00 = No operation
                             01 = READ or WRITE one byte only
                             02 = READ or WRITE a range of bytes
                             03 = POSITION then READ/WRITE one byte
                             04 = POSITION then READ/WRITE range
```

```
                  Ø2/Ø3 - (Subcodes Ø3 or Ø4) Record number
                  Ø4/Ø5 - (Subcodes Ø3 or Ø4) Byte offset
                  Ø6/Ø7 - (Subcodes Ø2 or Ø4) Number of bytes in
                          range to be read or written. (Note: for
                          WRITE, this length must be one less
                          than the actual length to be written)
                  Ø8/Ø9 - (Subcodes Ø2 or Ø4) Address of range of
                          bytes to be written or address of
                          buffer to which bytes are to be read.
                  Ø8    - (WRITE, Subcodes Ø1 or Ø3) Single byte
                          to be written.
                  ØC/ØD - Address of file manager workarea buffer
                  ØE/ØF - Address of T/S List sector buffer
                  1Ø/11 - Address of data sector buffer
        Output: Byte Ø2/Ø3 - Record number of current file position
                  Ø4/Ø5 - Byte offset of current position in file
                  Ø8    - (READ, Subcodes Ø1 or Ø3) Byte read
                  ØA    - Return code

DELETE  Locate and delete a file, freeing its sectors.
        Input:  Byte ØØ    - Ø5
                  (remainder are the same as with OPEN call type)
        Output: Byte ØA    - Return code

CATALOG Produce a catalog listing on the output device.
        Input:  Byte ØØ    - Ø6
                  Ø5    - Drive
                  Ø6    - Slot
                  ØC/ØD - Address of file manager workarea buffer
                  ØE/ØF - Address of T/S List sector buffer
                  1Ø/11 - Address of data sector buffer
        Output: Byte ØA    - Return code

LOCK    Lock a file.
        Input:  Byte ØØ    - Ø7
                  (remainder are the same as with OPEN call type)
        Output: Byte ØA    - Return code

UNLOCK  Unlock a file.
        Input:  Byte ØØ    - Ø8
                  (remainder are the same as with OPEN call type)
        Output: Byte ØA    - Return code

RENAME  Rename a file.
        Input:  Byte ØØ    - Ø9
                  Ø2/Ø3 - Address of new file name (3Ø bytes)
                  (remainder are the same as with OPEN call type)
        Output: Byte ØA    - Return code
```

```
POSITION Calculate the location of a record and/or byte offset in the
         file. Position such that next READ or WRITE will be at that
         location in the file. A call to POSITION (either explicitly or
         implictly using subcodes of READ or WRITE) is required prior to
         the first READ or WRITE. Bytes Ø2 through Ø5 should be set to
         zeros for a normal position to the beginning of the file.
         Input:  Byte ØØ    - ØA
                      Ø2/Ø3 - Relative record number for files with a
                              fixed length record size or zero. First
                              record of file is record ØØØØ.
                      Ø4/Ø5 - Relative byte offset into record or of
                              entire file if record number is zero.
                      ØC/ØD - Address of file manager workarea buffer.
                      ØE/ØF - Address of T/S List sector buffer.
                      1Ø/11 - Address of data sector buffer.
         Output: Byte ØA    - Return code

 INIT    Initialize a slave diskette. This function formats a diskette and
         writes a copy of DOS onto tracks Ø-2. A VTOC and Catalog are also
         created. A HELLO program is not stored, however.
         Input:  Byte ØØ    - ØB
                      Ø1    - First page of DOS image to be copied to
                              the diskette. Normally $9D for a 48K
                              machine.
                      Ø4    - Volume number of new diskette.
                      Ø5    - Drive number (Ø1 or Ø2)
                      Ø6    - Slot number (Ø1-Ø7)
                      ØC/ØD - Address of file manager workarea buffer.
                      ØE/ØF - Address of T/S List sector buffer.
                      1Ø/11 - Address of data sector buffer.
         Output: Byte ØA    - Return code

 VERIFY  Verify that there are no bad sectors in a file by reading every
         sector.
         Input:  Byte ØØ    - ØC
                      (remainder are the same as the OPEN call type)
         Output: Byte ØA    - Return code
```

DOS BUFFERS

Usually it is desirable to use one of DOS's buffers when calling the file manager to save memory. DOS buffers consist of each of the three buffers used by the file manager (file manager workarea, T/S List sector, and data sector) as well as a 30 byte file name buffer and some link pointers. All together a DOS buffer occupies 595 bytes of memory. The address of the first DOS buffer is stored in the first two bytes of DOS (`$9DØØ` on a 48K Apple II).

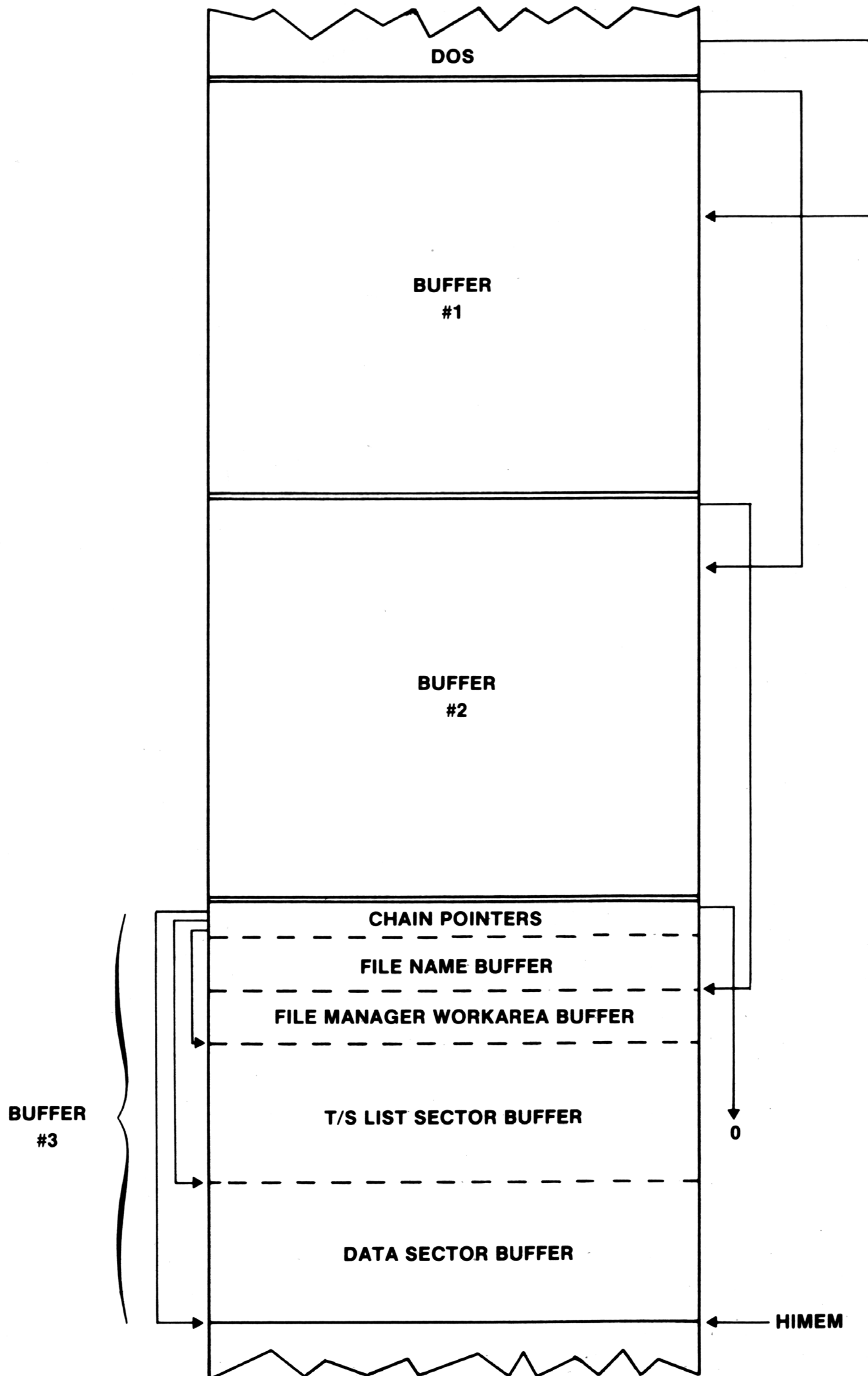


Figure 6.3 DOS File Buffers

The address of the next buffer is stored in the first and so on in a chain of linked elements. The link address to the next buffer in the last buffer is zeros. If the buffer is not being used by DOS, the first byte of the file name field is a hex `ØØ`. Otherwise, it contains the first character of the name of the open file. The assembly language programmer should follow these conventions to avoid having DOS reuse the buffer while it's in use. This means that the name of the file should be stored in the buffer to reserve it for exclusive use (or at least a non-zero byte stored on the first character) and later, when the user is through with the buffer, a `ØØ` should be stored on the file name to return it to DOS's use. If the later is not done, DOS will eventually run out of available buffers and will refuse even to do a `CATALOG` command. A diagram of the DOS buffers for `MAXFILES 3` is given in Figure 6.3 and the format of a DOS buffer is given below.

DOS BUFFER FORMAT

```
BYTE    DESCRIPTION
ØØØ/ØFF Data sector buffer (256 bytes in length)
1ØØ/1FF T/S List sector buffer (256 bytes in length)
2ØØ/22C File manager workarea buffer (45 bytes in length)
22D/24A File name buffer (3Ø bytes in length)
        First byte indicates whether this DOS buffer is
        being used. If hex ØØ, buffer is free for use.
24B/24C Address (Lo/High) of file manager workarea buffer
24D/24E Address of T/S List sector buffer
24F/25Ø Address of data sector buffer
251/252 Address of the file name field of the next buffer on
        the chain of buffers. If this is the last buffer on
        the chain then this field contains zeros.
```

THE FILE MANAGER WORKAREA

The file manager workarea contains the variables which, taken together, constitute all of the information the file manager needs to deal with an open file. Each time the file manager finishes processing a call, it copies all of its important variables into the file manager workarea buffer provided by the caller. Each subsequent time the file manager is called, the first thing it does is to copy the contents

The DOS File Manager

of the file manager workarea buffer back into its variables so that it may resume processing for the file where it left off on the previous call. Ordinarily, the programmer will have no need to worry about the contents of this workarea, since most of the useful information is present in the parameter list anyway. Occasionally, it is handy to know more about the open file. For these cases, the format of the file manager workarea is given below:

FILE MANAGER WORKAREA FORMAT

```
BYTE     DESCRIPTION
ØØ/Ø1    Track/Sector of first T/S List for file
Ø2/Ø3    Track/Sector of current T/S List for file
Ø4       Flags:
           8Ø=T/S List buffer changed and needs writing
           4Ø=Data buffer has been changed and needs writing
           Ø2=Volume freespace map changed and needs writing
Ø5/Ø6    Track/Sector of current data sector
Ø7       Sector offset into catalog to entry for this file
Ø8       Byte offset into catalog sector to entry for file
Ø9/ØA    Maximum data sectors represented by one T/S List
ØB/ØC    Offset of first sector in current T/S List
ØD/ØE    Offset of last sector in current T/S List
ØF/1Ø    Relative sector number last read
11/12    Sector size in bytes (256)
13/14    Current position in sectors (relative)
15       Current byte offset in this sector
16       Not used
17/18    Fixed record length
19/1A    Current record number
1B/1C    Byte offset into current record
1D/1E    Length of file in sectors
1F       Next sector to allocate on this track
2Ø       Current track being allocated
21/24    Bit map of available sectors on this track (rotated)
25       File type (8Ø=locked) Ø,1,2,4=T,I,A,B
26       Slot number times 16 (example: $6Ø=slot 6)
27       Drive number (Ø1 or Ø2)
28       Volume number (complemented)
29       Track
2A/2C    Not used
```

COMMON ALGORITHMS

Given below are several pieces of code which are used when working with DOS:

LOCATE A FREE DOS BUFFER

The following subroutine may be used to locate an unallocated DOS buffer for use with the DOS file manager.

```
FBUFF   LDA     $3D2     LOCATE DOS LOAD POINT
        STA     $1
        LDY     #Ø
        STY     $Ø
*
GBUFØ   LDA     ($Ø),Y   LOCATE NEXT DOS BUFFER
        PHA
        INY
        LDA     ($Ø),Y
        STA     $1
        PLA
        STA     $Ø
        BNE     GBUF     GOT ONE
        LDA     $1
        BEQ     NBUF     NO BUFFERS FREE
*
GBUF    LDY     #Ø       GET FILENAME
        LDA     ($Ø),Y
        BEQ     GOTBUF   ITS FREE
        LDY     #36      ITS NOT FREE
        BNE     GBUFØ    GO GET NEXT BUFFER
*
GOTBUF  CLC              INDICATE-GOT A FREE BUFFER
        RTS              RETURN TO CALLER
NBUF    SEC              INDICATE-NO FREE BUFFERS
        RTS              RETURN TO CALLER
```

WHICH VERSION OF DOS IS ACTIVE?

In case the program has version dependent code, a check of the DOS version may be required:

```
        CLC
        LDA     #Ø       ADD $16BE TO DOS LOAD POINT
        ADC     #$BE
        STA     $Ø
        LDA     $3D2
```

```
        ADC     #$16
        STA     $1
        LDY     #Ø
        LDA     ($Ø),Y  GET DOS VERSION NUMBER (2 OR 3)
```

IS DOS IN THE MACHINE?

The following series of instructions should be used prior to attempting to call RWTS or the file manager to insure that DOS is present on this machine.

```
        LDA     $3DØ    GET VECTOR JMP
        CMP     #$4C    IS IT A JUMP?
        BNE     NODOS   NO, DOS NOT LOADED
```

WHICH BASIC IS SELECTED?

Some programs depend upon either the Integer BASIC ROM or the Applesoft ROM. To find out which is active and select the one desired, the following subroutine can be called. First the A register is loaded with a code to indicate which BASIC is desired. `$2Ø` is used for Integer BASIC and `$4C` is used for Applesoft. To set up for Applesoft, for example:

```
        LDA     #$4C    CODE FOR APPLESOFT
        JSR     SETBSC  CALL SUBROUTINE
        BNE     ERROR   LANGUAGE NOT AVAILABLE
        .
        .
        .
SETBSC  CMP     $EØØØ   CORRECT BASIC ALREADY THERE?
        BEQ     RTS     YES
        STA     $CØ8Ø   NO, SELECT ROM CARD
        CMP     $EØØØ   NOW DO WE HAVE IT?
        BEQ     RTS     YES
        STA     $CØ81   NO, TRY ROM CARD OUT
        CMP     $EOOO   GOT IT NOW?
RTS     RTS             IN ANY CASE, EXIT TO CALLER
```

SEE IF A BASIC PROGRAM IS IN EXECUTION

To determine if there is a BASIC program running or if BASIC is in immediate command mode, use the following statements:

```
 ..IF INTEGER BASIC IS ACTIVE...
        LDA     $D9
        BMI     EXEC    PROGRAM EXECUTING
        BPL     NOEXEC  PROGRAM NOT EXECUTING
```

```
..IF APPLESOFT BASIC IS ACTIVE...
       LDX     $76     GET LINE NUMBER
       INX
       BEQ     NOEXEC  PROGRAM NOT EXECUTING
       LDX     $33     GET PROMPT CHARACTER
       CPX     #$DD    PROMPT IS A “]”?
       BEQ     NOEXEC  YES, NOT EXECUTING
       BNE     EXEC    ELSE, PROGRAM IS EXECUTING
```

PROCESS DOS ERROR FROM ASSEMBLY LANGUAGE

If you need to have DOS print out an error message from your assembly language program, use the following piece of code to do it properly:

```
PRERROR     LDA #$FØ     Hook output to COUT
            STA $36      instead of DOS
            LDA #$FD     at $FDFØ
            STA $37      DOS will put it’s own ptr back
            LDY #$ØA     See list below of error codes
            LDA ($Ø4),Y  $Ø4 points to FM parmlist
            JMP $A6D2
```

Here is the list of DOS error code offsets:

Error Offset	Text of Error Message
Ø	RETURN BELL RETURN
1	“LANGUAGE NOT AVAILABLE”
2	“RANGE ERROR” (Bad file manager opcode)
3	“RANGE ERROR” (Bad file manager subcode)
4	“WRITE PROTECTED
5	“END OF DATA”
6	“FILE NOT FOUND”
7	“VOLUME MISMATCH”
8	“I/O ERROR”
9	“DISK FULL”
1Ø	“FILE LOCKED”
11	“SYNTAX ERROR”
12	“NO BUFFERS AVAILABLE”
13	“FILE TYPE MISMATCH”
14	“PROGRAM TOO LARGE”
15	“NOT DIRECT COMMAND”

Chapter 7
Customizing DOS

Although DOS usually provides most of the functionality needed by the BASIC or assembly language programmer, at times a custom change is required. Making changes to your copy of DOS should only be undertaken when absolutely necessary, since new versions of DOS are released from time to time, and the job of moving several patches to a new version of DOS every few months can become a burden. In addition, wholesale modification of DOS without a clear understanding of the full implications of each change can result in an unreliable system.

SLAVE VS MASTER PATCHING

The usual procedure for making changes to DOS involves "patching" the object or machine language code in DOS. Once a desired change is identified, a few instructions are stored over other instructions within DOS to modify the program. There are three levels at which changes to DOS may be applied.

Custom DOS

1. A patch can be made to DOS in memory. If this is done, a later reboot will cause the change to "fall out" or be removed.
2. A patch of the first type can be made permanent by initializing a diskette while running the patched DOS. This procedure creates a slave diskette with a copy of DOS on tracks 0, 1, and 2 which contains the patch. Each time this newly created diskette is booted the patched version of DOS will be loaded. Also, any slave diskettes created by that diskette will also contain the patched version of DOS.
3. The patch is applied directly to a master diskette. This is somewhat more complicated. Either the patch may be made to the image of DOS on the first three tracks of a master diskette using a zap program, or

MASTER CREATE may be used to write the changed copy of DOS to a new diskette. The following procedure may be followed to do this:

```
BLOAD MASTER CREATE
Get into the monitor (CALL -151)
Store a $4C at location $80D (80D:4C)
Execute MASTER CREATE (800G)
When MASTER CREATE finishes loading the DOS image
   it will exit. You may use the monitor to make
   changes in the image. MASTER CREATE loads DOS
   into memory at $1200 such that Boot 2 (RWTS) is
   loaded first, followed by the main part of DOS
   starting at $1C00.
When all patches have been made, reenter MASTER CREATE
   at location $82D (82DG).
Complete the MASTER CREATE update normally. The
   resulting diskette will have the patches applied.
```

This procedure will work for versions 3.2, 3.2.1, and 3.3 of DOS.

AVOIDING RELOAD OF LANGUAGE CARD

A rather annoying addition to DOS 3.3 was a patch to the Boot 2 code to store a binary zero in the first byte of the Language Card, forcing DOS to reload BASIC (either Integer or Applesoft) for every boot, whether or not the machine was just powered up. When the machine is first powered up this patch is not necessary, since the first byte of the Language Card does not appear to DOS to be either BASIC, and it will reload the card anyway. On subsequent reboots, more often than not, a good copy of BASIC already resides in the Language Card and this patch results in a LANGUAGE NOT AVAILABLE error message after booting a slave diskette. Presumably the patch was added for version 3.3 to allow for the eventual possibility that a language like Pascal whose first byte of code just happens to match one of the BASICs would cause strange results in DOS. If the user always powers the machine off and on between using DOS and any other system, the patch may be removed as follows.

At $BFD3 (48K) is a STA instruction which stores a zero on the Language Card. This instruction must be made into three no-operation instructions:

```
BFD3:EA EA EA
```

A slave diskette may then be `INIT`ed using this modified version of DOS and that diskette will have the patch in its DOS. The address of the store instruction for a 32K DOS is `$7FD3` and for a 16K DOS is `$3FD3`.

INSERTING A PROGRAM BETWEEN DOS AND ITS BUFFERS

Once in a while it is useful to find a "safe" place to load a machine language program (a printer driver, perhaps) where BASIC and DOS can never walk over it, even if DOS is coldstarted. If the program is less than 200 bytes long, `$3ØØ` is a good choice. For larger programs, it is usually better to "tuck" the program in between DOS and its buffers (assuming the program is relocatable and will run at that location). To do this, load the program into low RAM, copy it to high RAM right below `$9DØØ` (for a 48K machine), over the top of DOS's buffers, change the first buffer address at `$9DØØ` to point below your program, (remember to allow 38 extra bytes for the filename and link fields) and `JMP` to `$3D3` (DOS Coldstart). This will cause DOS to rebuild its buffers below your program and "forget" about the memory your program occupies until the next time DOS is booted. Of course, BASIC can not get at that memory either, since its HIMEM is below the DOS buffers.

You can also do this from an Applesoft program, using the following piece of code:

```
Ø POKE 4Ø192, PEEK(4Ø192) - X
1 CALL 42964
```

With `X` equal to how many pages (256 bytes each) you'd like to reserve.

BRUN OR EXEC THE HELLO FILE

Ordinarily, when DOS finishes booting into memory, it performs a `RUN` command on the `HELLO` file in its file name buffer (left there by the `INIT` command which wrote DOS to the diskette). To change the `RUN` command to a `BRUN` or an `EXEC`, apply the following patch to DOS (48K):

```
9E42:34          (for BRUN)
..or..
9E42:14          (for EXEC)
```

REMOVING THE PAUSE DURING A LONG CATALOG

Normally, when a `CATALOG` command is done on a disk with many files, DOS will pause every time the screen fills with names to allow the user time to see them all. By pressing any key the `CATALOG` continues. If this pause is undesirable, apply the following patch to DOS (48K):

```
AE34:6Ø
```

WRITING DOS IN MEMORY TO A DISKETTE

If you'd like to write the current version of DOS onto a disk, use the code below. You might want to make changes to DOS before committing them to a disk, and using this method will allow you to experiment first.

```
LOAD HELLO
POKE -21921, Ø : POKE -18448, Ø : POKE -18447, 157 : POKE
-18453, Ø : CALL -18614
```

Load the `HELLO` program from the disk so DOS will use it for the startup program. Location `-21921` (`43615`) is the "Last Command Executed" byte in DOS. Poking `Ø` into this location tells DOS that `INIT` was the last command. (`AA5F: ØØ`)

Locations `-18448` (`47Ø88`) and `-18447` (`47Ø89`) tells the write routines to start at `$9DØØ`, the beginning of DOS in memory. (`B7FØ: ØØ 9D`)

Location `-18453` (`47Ø83`) is the Disk Volume Number, and putting `Ø` into this location will match any disk. (`B7EB: ØØ`)

The call to `-18614` (`46922`) will use DOS's write routine to write DOS onto the current disk using the last slot and drive accessed. (`B74A`)

Chapter 8

DOS Program Logic

This chapter will take a detailed look at the operation of the DOS program itself to aid the Apple user in understanding it and to help make intelligent use of its facilities. Each subroutine and group of variables or constants will be covered separately by storage address. The enterprising programmer may wish to create a disassembly of DOS on the printer and transfer the annotations given here directly to such a listing. Addresses used will be for DOS 3.3 and for a 48K master diskette version of DOS. Slot 6 is assumed. Unless specifically indicated by a `$` character, lengths are given in decimal, addresses in hexadecimal (base 16).

DISK II CONTROLLER CARD ROM - BOOT 0

```
ADDRESS
C6ØØ-C65B This routine is the first code executed when a disk
          is to be booted. It receives control via PR#6 or
          C6ØØG or 6 control-P.
          Dynamically build a translate table for converting
          disk codes to six bit hex at location $356-$3FF.
          Call an RTS instruction in the monitor ROM and
          extract the return address from the stack to find out
          the address of this controller card ROM.
          Use this address to determine the slot number of this
          drive by shifting $Csxx.
          Save the slot number times 16 ($sØ)
          Clear disk I/O latches, set read mode, select drive
          1, turn disk drive on.
          Pull disk arm back over 8Ø tracks to recalibrate the
          arm to track zero.
          Set up parms to read sector zero on track zero to
          location $8ØØ.
          Execution falls through into a general sector read
          subroutine at C65C.
C65C-C6FA This subroutine reads the sector number stored at
          $3D on the track indicated by $41 to the address
          stored at $26,$27.
          Look for D5/AA/96 sector address header on the disk.
          If D5/AA/AD is found and sector data was wanted, go
          to C6A6.
     C683 Handle a sector address block.
          Read three double bytes from the disk and combine
          them to obtain the volume, track, and sector number
```

of the sector being read from the disk at this time.
Store the track at $4Ø.
Compare the sector found to the sector wanted and the track found to the track wanted.
If no match, go back to C65C.
Otherwise, if sector is correct, go to C65D to find the sector data itself.

C6A6 Handle sector data block.
Read the 85 bytes of secondary data to $3ØØ-$355.
Read 256 bytes of primary data to the address stored at $26,$27.
Verify that the data checksum is valid.
If not, start over at C65C.
"Nibbilize" the primary and secondary data together into the primary data buffer ($26,$27).
Increment $27 (address page of read data) and $3D (sector number to be read) and check against $8ØØ to see if additional sectors need to be read.
If so, reload slot*16 and go back to C65C to read next sector. (This feature is not used when loading DOS but is used when loading from a BASICS diskette.)
Otherwise, go to $8Ø1 to begin executing the second stage of the bootstrap.

FIRST RAM BOOTSTRAP LOADER - BOOT 1

ADDRESS

Ø8Ø1-Ø84C This routine loads the second RAM loader, Boot 2, including RWTS, into memory and jumps to it.
If this is not the first entry to Boot 1, go to $81F.
Get slot*16 and shift down to slot number.
Create the address of the ROM sector read subroutine (C65C in our case) and store it at $3E,$3F.
Pick up the first memory page in which to read Boot 2 from location $8FE, add the length of Boot 2 in sectors from $8FF, and set that value as the first address to which to read (read last page first).

Ø81F Get sector to read, if zero, go to $839.
Translate theoretical sector number into physical sector number by indexing into skewing table at $84D.
Decrement theoretical sector number (8FF) for next iteration through.
Set up parameters for ROM subroutine (C65C) and jump to it. It will return to $8Ø1 when the sector has been read.

Ø839 Adjust page number at 8FE to locate entry point of Boot 2.
Perform a PR#Ø and IN#Ø by calling the monitor.
Initialize the monitor (TEXT mode, standard window, etc.)
Get slot*16 again and go to Boot 2 ($37ØØ for a master disk, $B7ØØ in its final relocated location).

DOS 3.3 MAIN ROUTINES

```
ADDRESS
9DØØ-9DØF Relocatable address constants
     9DØØ Address of first DOS buffer at its file name field.
     9DØ2 Address of the DOS keyboard intercept routine.
     9DØ4 Address of the DOS video intercept routine.
     9DØ6 Address of the primary file name buffer.
     9DØ8 Address of the secondary (RENAME) file name buffer.
     9DØA Address of the range length parameter used for LOAD.
     9DØC Address of the DOS load address ($9DØØ).
     9DØE Address of the file manager parameter list.
9D1Ø-9D1C DOS video (CSWL) intercept's state handler address
          table. States are used to drive the handling of DOS
          commands as they appear as output of PRINT statements
          and this table contains the address of the routine
          which handles each state from state Ø to state 6.
9D1E-9D55 Command handler entry point table. This table
          contains the address of a command handler subroutine
          for each DOS command in the following standard order:
               INIT      A54F
               LOAD      A413
               SAVE      A397
               RUN       A4D1
               CHAIN     A4FØ
               DELETE    A263
               LOCK      A271
               UNLOCK    A275
               CLOSE     A2EA
               READ      A51B
               EXEC      A5C6
               WRITE     A51Ø
               POSITION  A5DD
               OPEN      A2A3
               APPEND    A298
               RENAME    A281
               CATALOG   A56E
               MON       A233
               NOMON     A23D
               PR#       A229
               IN#       A22E
               MAXFILES  A251
               FP        A57A
               INT       A59E
               BSAVE     A331
               BLOAD     A35D
               BRUN      A38E
               VERIFY    A27D
9D56-9D61 Active BASIC entry point vector table. The addresses
          stored here are maintained by DOS such that they
          apply to the current version of BASIC running.
     9D56 Address of CHAIN entry point to BASIC.
     9D58 Address of RUN.
     9D5A Address of error handler.
     9D5C Address of BASIC coldstart.
     9D5E Address of BASIC warmstart.
     9D6Ø Address of BASIC relocate (APPLESOFT only).
9D62-9D6B Image of the entry point vector for INTEGER BASIC.
          This image is copied to 9D56 if INTEGER BASIC is made
```

```
              active.
9D6C-9D77 Image of the entry point vector for the ROM version
              of APPLESOFT.
9D78-9D83 Image of the entry point vector for the RAM version
              of APPLESOFT.
9D84-9DBE DOS coldstart entry routine.
              Get the slot and drive numbers and store as default
              values for command keywords.
              Copy APPLESOFT ROM or INTEGER BASIC entry point
              vector into current BASIC entry point vector.
              Remember which BASIC is active.
              Go to 9DD1.
9DBF-9DE9 DOS warmstart entry routine.
              Get the remembered BASIC type and set the ROM card
              as necessary (calls A5B2).
     9DD1 Remember whether entry is coldstart or warmstart
              Call A851 to replace DOS keyboard and video
              intercepts.
              Set NOMON C,I,O.
              Set video intercept handler state to Ø.
              Coldstart or warmstart the current BASIC (exit DOS).
              (DOS will next gain control when BASIC prints its
              input prompt character)
9DEA-9E5Ø First entry processing for DOS. This routine is
              called by the keyboard intercept handler when the
              first keyboard input request is made by BASIC after
              a DOS coldstart.
              If RAM APPLESOFT is active, copy its entry point
              vector to the active BASIC entry point vector and
              blank out the primary file name buffer so that no
              HELLO file will be run.
              Set MAXFILES to 3 by default.
              Call A7D4 to build the DOS file buffers.
              If an EXEC was active, close the EXEC file
              Set the video intercept state to Ø and indicate
              warmstart status by calling A75B.
              If the last command executed was not INIT (this DOS
              was not just booted), go to 9E45.
              Otherwise, copy an image of the DOS jump vector to
              $3DØ-$3FF.
              Point $3F2,$3F3 to DOS warmstart routine.
              Set the AUTOSTART ROM power-up byte since the RESET
              handler address was changed.
              Set the command index for RUN (to run the HELLO file)
              and go to A18Ø to execute it.
     9E45 See if there is a pending command.
              If so, go to A18Ø to execute it. Otherwise, return
              to caller.
9E51-9E7F An image of the DOS 3-page jump vector which the
              above routine copies to $3DØ-$3FF. See Chapter 5 for
              a description of its contents.
9E81-9EB9 DOS keyboard intercept routine.
              Call 9ED1 to save the registers at entry to DOS.
              If not coldstarting or reading a disk file,
              go to 9E9E.
              Get value in A register at entry and echo it on the
              screen (erases flashing cursor).
              If in read state (reading a file) go to A626 to get
              next byte from disk file.
```

```
          Otherwise, call 9DEA to do first entry processing.
          Put cursor on screen in next position.
          If EXECing, call A682 to get the next byte from the
          EXEC file.
          Set the video intercept state to 3 (input echo).
          Call 9FBA to restore the registers at entry to DOS.
          Call the true keyboard input routine.
          Save the input character so that it will be restored
          with the registers in the A register.
          Do the same with the new X register value.
          Exit DOS via 9FB3.
9EBA-9EBC A jump to the true KSWL handler routine.
9EBD-9EDØ DOS video intercept routine.
          Call 9ED1 to save the registers at entry to DOS.
          Get the video intercept state and, using it as an
          index into the state handler table (9D11), go to
          the proper handler routine, passing it the character
          being printed.
9ED1-9EEA Common intercept save registers routine.
          Save the A, X, Y, and S registers at AA59-AA5C.
          While in DOS, restore the true I/O handlers (KSWL and
          CSWL) to $36-$39.
          Return to caller.
9EEB-9F11 State Ø output handler. --start of line--
          If a RUN command was interrupted (by loading RAM
          APPLESOFT) go to 9F78 to complete it.
          If read flag is on (file being read) and output is a
          "?" character (BASIC INPUT), go to state 6 to skip
          it.
          If read flag is on and output is prompt character
          ($33) go to state 2 to ignore the line.
          Set state to 2 (ignore non-DOS command) just in case.
          If output character is not a control-D, go to
          state 2.
          Otherwise, set state to 1 (collect possible DOS
          command), set line index to zero, and fall through
          to state 1.
9F12-9F22 State 1 output handler. --collect DOS command--
          Using line index, store character in input buffer at
          $2ØØ.
          Increment line index.
          If character is not a carriage return, exit DOS
          via 9F95 (echo character on screen if MON I).
          Otherwise, go to command scanner at 9FCD.
9F23-9F2E State 2 output handler. --non-DOS command ignore--
          If the character is not a carriage return, exit DOS
          via 9FA4 (echo character on screen).
          Otherwise, set state back to Ø and exit DOS via
          9FA4.
9F2F-9F51 State 3 output handler. --INPUT statement handler--
          Set state to Ø in case INPUT ends.
          If character is not a carriage return, echo it on
          screen as long as EXEC is not in effect with NOMON I
          but exit DOS in any case. (KSWL will set state=3)
          Otherwise, call A65E to see if BASIC is executing a
          program or is in immediate mode. If EXEC is running
          or if BASIC is in immediate mode, go to state 1 to
          collect the possible DOS command.
          Otherwise, exit DOS, echoing the character as
```

```
            appropriate.
9F52-9F6Ø State 4 output handler. --WRITE data to a file--
            If the character is a carriage return, set state to
            5 (start of write data line).
            Call A6ØE to write the byte to the disk file.
            Exit DOS with echo on screen if MON O.
9F61-9F7Ø State 5 output handler. --Start of WRITE data line--
            If the character is a control-D, go to state Ø to
            immediately exit write mode.
            If the character is a line feed, write it and exit,
            staying in state 5.
            Otherwise, set the state to 4 and go to state 4.
9F71-9F77 State 6 output handler. --Skip prompt character--
            Set state to Ø.
            Exit DOS via 9F9D (echo if MON I).
9F78-9F82 Finish RUN command, interrupted by APPLESOFT RAM LOAD
            Reset the "RUN interrupted" flag.
            Call A851 to replace the DOS CSWL/KSWL intercepts.
            Go to A4DC to complete the RUN command.
9F83-9F94 DOS command scanner exit to BASIC routine.
            If first character of command line is control-D,
            go to echo exit (9F95).
            Otherwise, set things up so BASIC won't see the DOS
            command by passing a zero length line (only a
            carriage return). Fall through to echo exit.
9F95-9FBØ Echo character on screen (conditionally) and exit DOS
     9F95 Echo only if MON C set, otherwise, go to 9FB3.
     9F99 Echo only if MON O set, otherwise, go to 9FB3.
     9F9D Echo only if MON I set, otherwise, go to 9FB3.
     9FA4 Always echo character.
            Call 9FBA to restore registers at entry to DOS.
            Call 9FC5 to echo character on screen.
            Save contents of the registers after echoing.
            Fall through to DOS exit routine.
9FB3-9FC4 DOS exit routine and register restore.
            Call A851 to put back DOS KSWL/CSWL intercepts.
            Restore S (stack) register from entry to DOS.
     9FBA DOS register restore subroutine.
            Restore registers from first entry to DOS and return
            to caller.
9FC5-9FC7 A jump to the true CSWL routine.
9FC8-9FCC Skip a line on the screen.
            Load a carriage return into the A register and
            call 9FC5 to print it.
9FCD-A179 DOS command parse routine.
            Set the command index to -1 (none).
            Reset the pending command flag (none pending).
     9FD6 Add one to command index.
            If first character is a control-D, skip it.
            Flush to a non-blank (call A1A4).
            Compare command to command name in command name table
            at A884 for the current command index.
            If it doesn't match and if there are more entries
            left to check, go back to 9FD6.
            If it does match, go to AØ1B.
            Otherwise, if command was not found in the table,
            check to see if the first character was a control-D.
            If so, go to A6C4 to print "SYNTAX ERROR".
            Otherwise, call A75B to reset the state and warmstart
```

```
     flag and go to 9F95 to echo the command and exit.
     (the command must be for BASIC, not DOS)
AØ1B Compute an index into the operand table for the
     command which was entered.
     Call A65E to see if a BASIC program is executing.
     If not, and the command is not a direct type command,
     (according to the operand table) go to A6D2 to print
     "NOT DIRECT COMMAND".
     Otherwise, if the command is RUN, make the prompt
     character ($33) non-printing.
     Check the operand table to see if a first filename
     is a legal operand for this command.
     If not, go to AØAØ.
     Otherwise, clear the filename buffer (call AØ95).
     Flush to the next non-blank (call A1A4) and copy
     the filename operand to the first filename buffer.
     Skip forward to a comma if one was not found yet.
     If a second filename is legal for this command, use
     the code above to copy it into the second filename
     buffer.
     Check both filenames to see if they are blank.
     If one was required by the command but not given,
     give a syntax error or pass it through to BASIC.
     (As in the case of LOAD with no operands)
     If all is well, go to AØD1 to continue.
AØ95 A subroutine to blank both filename buffers.
AØAØ Indicate no filename parsed.
     Check operand table to see if a positional operand
     is expected.
     If not, go to AØD1 to continue.
     Otherwise, call A1B9 to convert the numeric operand.
     If omitted, give syntax error.
     If number converted exceeds 16, give "RANGE ERROR"
     If number is supposed to be a slot number, give
     "RANGE ERROR" if it exceeds 7.
     If number is not a slot number, give "RANGE ERROR" if
     it is zero. (MAXFILES Ø is a no-no)
AØD1 Set defaults for the keyword operands (V=Ø,L=Ø,B=Ø)
AØE8 Get the line offset index and flush to the next
     non-blank, skipping any commas found.
     If we are not yet to the end of the line, go to A1ØC.
     Check to see if any keywords were given which were
     not allowed for this command.
     If not, go to A17A to process the command.
A1ØC Lookup the keyword found on the command line in the
     table of valid keywords (A94Ø)
     If not in table, give "SYNTAX ERROR" message.
     Get its bit position in the keywords-given flag.
     If the keyword does not have an operand value, go to
     A164.
     Otherwise, indicate keyword found in flag.
     Convert the numeric value associated with keyword.
     Give "SYNTAX ERROR" message if invalid.
     Check to see if the number is within the acceptable
     range as given in the keyword valid range table at
     A955.
     Save the value of the keyword in the keyword values
     table starting at AA66.
     Go parse the next keyword. Go to AØE8.
```

```
     A164 Indicate C, I, or O keywords were parsed.
          Update the MON value in the keyword value table
          appropriately.
          Go parse the next keyword. Go to AØE8.
A17A-A17F Call A18Ø to process the command, then exit via echo
          at 9F83.
A18Ø-A192 Do command.
          Reset the video intercept state to zero.
          Clear the file manager parameter list.
          Using the command index, get the address of the
          command handling routine from the command handler
          routine table at 9D1E and go to it.
          Command handler will exit to caller of this routine.
A193-A1A3 Get next character on command line and check to see
          if it is a carriage return or a comma.
A1A4-A1AD Flush command line characters until a non-blank is
          found.
A1AE-A1B8 Clear the file manager parameter list at B5BA to
          zeros.
A1B9-A1D5 Convert numeric operand from command line. Call
          either A1D6 (decimal convert) or A2Ø3 (hex convert)
          depending upon the presence or lack thereof of a
          dollar sign ($).
A1D6-A2Ø2 Decimal convert subroutine.
A2Ø3-A228 Hexadecimal convert subroutine.
A229-A22D PR#n command handler.
          Load the parsed numeric value and exit via FE95 in
          the monitor ROM.
A22E-A232 IN#n command handler.
          Load the parsed numeric value and exit via FE8B in
          the monitor ROM.
A233-A23C MON command handler.
          Add new MON flags to old in AA5E and exit.
A23D-A25Ø NOMON command handler.
          If C was given, put out a carriage return since this
          line was echoed but its CR was not.
          Turn off the proper bits in AA5E and exit.
A251-A262 MAXFILES command handler.
          Turn off any EXEC file which is active.
          Close all open files (call A316).
          Set the new MAXFILES number at AA57.
          Go to A7D4 to rebuild the DOS file buffers and exit.
A263-A27Ø DELETE command handler.
          Load the delete file manager opcode (Ø5).
          Call the file manager open driver (A2AA) to perform
          the delete.
          Find the file buffer used to do the delete and free
          it (call A764).
          Exit to caller.
A271-A274 LOCK command handler.
          Load the lock file manager opcode (Ø7) and go to
          A277.
A275-A27C UNLOCK command handler.
          Load the unlock file manager opcode (Ø8).
     A277 Call the file manager open driver (A2AA) to perform
          the desired function.
          Exit to the caller via close (A2EA).
A27D-A28Ø VERIFY command handler.
          Load the verify file manager opcode (ØC) and go to
```

```
          A277 to perform function.
A281-A297 RENAME command handler.
          Store address of second file name in file manager
          parameter list.
          Load the rename file manager opcode (Ø9).
          Call the file manager driver at A2C8.
          Exit via close (A2EA).
A298-A2A2 APPEND command handler.
          Call A2A3 to OPEN the file.
          Read the file byte by byte until a zero is found.
          If append flag is on, add one to record number
          and turn flag off.
          Exit via a call to POSITION.
A2A3-A2A7 OPEN command handler.
          Set file type as TEXT.
          Go to A3D5 to open file.
A2A8-A2E9 Command handler common file management code.
          Set opcode to OPEN.
     A2AA If no L value was given on the command, use ØØØ1 and
          store record length value in file manager parmlist.
     A2C8 Close file if already open.
          Is there an available file buffer?
          If not, issue "NO FILE BUFFERS AVAILABLE" message.
          Point $4Ø,$41 at the free file buffer.
          Copy filename to file buffer (allocates the buffer)
          (A743).
          Copy buffer pointers to file manager parmlist (A74E).
          Finish filling in the file manager parmlist (A71A).
          Set operation code in parmlist.
          Exit through the file manager driver.
A2EA-A2FB CLOSE command handler.
          If no filename was given as part of command,
          go to A316 to close all files.
          Otherwise, find the open file buffer for filename
          (A764).
          If no such file open, exit to caller.
          Otherwise, close file and free buffer (A2FC).
          Go back through CLOSE command handler to make sure
          there are not more open buffers for the same file.
A2FC-A315 Close a file and free its file buffer.
          Find out if this buffer is EXEC's (A7AF).
          If so, turn EXEC flag off.
          Release the buffer by storing a $ØØ on its filename
          field.
          Copy file buffer pointers to the file manager
          parmlist.
          Set file manager opcode to CLOSE.
          Exit through the file manager driver routine.
A316-A33Ø Close all open files.
          Point to first file buffer (A792).
          Go to A32Ø.
     A31B Point to next file buffer on chain (A79A).
          If at end of chain, exit to caller.
     A32Ø Is this file buffer EXEC's?
          If so, skip it and go to A31B.
          Is it not in use (open)?
          If so, skip it and go to A31B.
          Otherwise, close it and free it (A2FC).
          Go to A316 to start all over.
```

```
A331-A35C BSAVE command handler.
          Insure that the A and L keywords were present on the
          command.
          If not, issue "SYNTAX ERROR" message.
          Open and verify a B type file (A3D5).
          Write the A keyword value as the first two bytes of
          the file.
          Write the L keyword value as the next two bytes of
          the file.
          Use the A value to exit by writing a range of bytes
          from memory to the file.
A35D-A38D BLOAD command handler.
          Open the file, ignoring its type.
          Insure the type is B.
          If not, issue "FILE TYPE MISMATCH" message.
          Otherwise, open B type file and test file type (A3D5)
          Read the A value from the first two bytes of file.
          If A keyword was not given, use the value just read.
          Read L value as next two bytes in file.
          Go to A471 to read range of bytes to memory from file
A38E-A396 BRUN command handler.
          Call BLOAD command handler to load file into memory.
          Replace DOS intercepts.
          Exit DOS by jumping to the A address value to begin
          execution of the binary program.
A397-A3D4 SAVE command handler.
          Get the active BASIC type (AAB6).
          If INTEGER, go to A3BC.
          If APPLESOFT, test $D6 flag to see if program is
          protected.
          If so, issue "PROGRAM TOO LARGE" message.
          Otherwise, open and test for A type file (A3D5).
          Compute program length (PGMEND-LOMEM).
          Write this two byte length to file.
          Exit by writing program image from LOMEM as a range
          of bytes (A3FF).
     A3BC Open and test for I type file (A3D5).
          Compute program length (HIMEM-PGMSTART).
          Write this two byte length to file.
          Exit by writing program image from PGMSTART as a
          range of bytes (A3FF).
A3D5-A3DF Open and test file type.
          Set file type wanted in file manager parmlist.
          Call A2A8 to open file.
          Go to A7C4 to check file type.
A3EØ-A3FE Write a 2 byte value to the open file.
          Store value to be written in file manager parmlist.
          Set write one byte opcodes.
          Call file manager driver.
          Call it again to write second byte and exit to caller.
A3FF-A4ØF Read/write a range of bytes.
          Set the address of the range in file manager parmlist.
          Set subcode to read or write a range of bytes.
          Call the file manager driver.
          Close the file.
          Exit through the VERIFY command handler to insure
          data was written ok.
A41Ø-A412 Issue "FILE TYPE MISMATCH" message.
A413-A479 LOAD command handler.
```

```
            Close all files (A316).
            Open the file in question.
            Is it an A or I type file?
            If not, issue "FILE TYPE MISMATCH" message.
            Which BASIC is active?
            If INTEGER, go to A45Ø.
            Select APPLESOFT BASIC (A4B1). This call could result
            in DOS losing control if the RAM version must be
            run.
            Read first two bytes of file as length of program.
            Add length to LOMEM (program start) to compute
            program end.
            Is program end beyond HIMEM?
            If so, close file and issue "PROGRAM TOO LARGE".
            Set program end and start of variables pointers.
            Read program as range of bytes to program start.
            Replace DOS intercepts (A851).
            Go to BASIC's relocation routine to convert a RAM
            APPLESOFT program to ROM and vice versa as needed.
       A45Ø Select INTEGER BASIC (A4B1).
            Read length of program (first two bytes in file).
            Compute program start (HIMEM-LENGTH).
            If zero or less than LOMEM, issue "PROGRAM TOO LARGE"
            message and close file.
            Set program start pointers.
            Read program into memory as a range of bytes.
            Exit to caller.
A47A-A4AA   Read two bytes from file (Address or Length).
            Set up parmlist to read two bytes to range length
            field (AA6Ø).
            Call file manager driver.
            Store value read as range length in file manager
            parmlist just in case it was a length.
A4AB-A4BØ   Close file and issue "PROGRAM TOO LARGE" message.
A4B1-A4DØ   Select desired BASIC.
            If desired BASIC is already active, exit to caller.
            Save current command index in case we must RUN
            APPLESOFT.
            If INTEGER, go to A59E to select it.
            Otherwise, copy primary file name to secondary
            buffer to save it in case RAM APPLESOFT is needed.
            Go to A57A to set APPLESOFT.
A4D1-A4E4   RUN command handler.
            If APPLESOFT is active, set RUN intercepted flag so
            that RUN can complete after APPLESOFT is loaded.
            Call LOAD command handler to load the program.
            Skip a line on the screen.
            Put DOS intercepts back.
            Go to the RUN entry point in the current BASIC.
A4E5-A4EF   INTEGER BASIC RUN entry point intercept.
            Delete all variables (CLR equivalent).
            Go to the CHAIN entry point in INTEGER BASIC.
A4FØ-A4FB   CHAIN command handler.
            Call the LOAD command handler to load the program.
            Skip a line.
            Replace DOS intercepts.
            Go to current BASIC's CHAIN entry point.
A4FC-A5Ø5   APPLESOFT ROM RUN entry point intercept.
            Call APPLESOFT to clear variables.
```

```
          Reset ONERR.
          Go to RUN entry point.
A5Ø6-A5ØD APPLESOFT RAM RUN entry point intercept.
          Call APPLESOFT to clear variables.
          Reset ONERR.
          Go to RUN entry point.
A51Ø-A51A WRITE command handler.
          Call READ/WRITE common code (A526).
          Set CSWL state to 5 (WRITE mode line start).
          Exit DOS (9F83).
A51B-A525 READ command handler.
          Call READ/WRITE common code (A526).
          Set READ mode flag in status flags (AA51).
          Exit DOS (9F83).
A526-A54E READ/WRITE common code.
          Locate the open file buffer for this file (A764).
          If not open, open it.
          Copy file buffer addresses to file manager parmlist.
          If R or B were given on command, copy to parmlist
          and issue a POSITION call to file manager.
          Exit to caller.
A54F-A56D INIT command handler.
          If V was given, use it. Otherwise, use 254.
          Store first page number of DOS in file manager
          parmlist.
          Call file manager driver to INIT diskette.
          Exit through SAVE to store greeting program on disk.
A56E-A579 CATALOG command handler.
          Call file manager with CATALOG opcode.
          Set new V value as default for future commands.
          Exit to caller.
A57A-A59D FP command handler.
          Set ROM card, if any, for APPLESOFT (A5B2).
          If successful, coldstart DOS (9D84).
          Otherwise, set status flag to indicate INTEGER BASIC
          is active.
          Set primary filename buffer to "APPLESOFT".
          Set flags to indicate RAM APPLESOFT and coldstart.
          Go to RUN command handler.
A59E-A5B1 INT command handler.
          Set ROM card, if any, for INTEGER BASIC (A5B2).
          If not successful, issue "LANGUAGE NOT AVAILABLE".
          Otherwise, clear RUN intercepted flag.
          Coldstart DOS (9D84).
A5B2-A5C5 Set ROM to desired BASIC.
          (This routine is passed a $4C for APPLESOFT or a $2Ø
          for INTEGER, since these bytes appear at $EØØØ in
          these BASICs. It will work regardless of which
          BASIC is onboard.)
          If desired BASIC is already available, exit.
          Try selecting ROM card.
          If desired BASIC is now available, exit.
          Try selecting onboard ROM.
          If desired BASIC is now available, exit.
          Otherwise, exit with error return code.
A5C6-A5DC EXEC command handler.
          Open the file (A2A3).
```

```
          Copy file buffer address to EXEC's buffer pointer at
          AAB4,AAB5.
          Set EXEC active flag (AAB3).
          Jump into POSITION command handler to skip R lines.
A5DD-A60D POSITION command handler.
          Locate the open file buffer (A764).
          If not found, open one as a TEXT file.
          Copy buffer pointers to file manager parmlist.
          If R was not given on command, exit.
     A5F2 Otherwise, test R value for zero and exit if so.
          Decrement R value by one.
          Read file byte by byte until a carriage return (end
          of line - $8D) is reached.
          If at end of file, issue "END OF FILE" message.
          Otherwise, go to A5F2 to skip next record.
A60E-A625 Write one data byte to file.
          Insure that BASIC is running a program (A65E).
          If not, close file and warmstart DOS.
          Set up file manager parmlist to write the data byte
          to the open file.
          Call file manager and exit.
A626-A65B Read one data byte from file.
          Insure that BASIC is running a program (A65E).
          If not, close file and warmstart DOS.
          Set CSWL intercept state to 6 (skip prompt character).
     A630 Read next file byte (A68C).
          If not at end of file, go to A644.
          Otherwise, close file.
          If state is not 3 (EXEC) issue "END OF DATA" message.
          Exit to caller.
     A644 If data byte is lower case character, turn its most
          significant bit off to fool GETIN routine in monitor.
          Store data byte in A register saved at entry to DOS.
          Using line index, turn high bit back on in previous
          data byte stored at $200 (input line buffer) to make
          it lower case if necessary.
          Exit DOS (9FB3).
A65E-A678 Test to see if BASIC is running a program or is in
          immediate command mode.
          If active BASIC is INTEGER, go to A672.
          If line number is greater than 65280 and prompt is
          "]" then APPLESOFT is in immediate mode.
          Otherwise, it is executing a program.
          Exit to caller with appropriate return code.
     A672 Check $D9 to determine whether BASIC is executing a
          program and exit with proper return code.
A679-A681 Close current file and warmstart DOS.
A682-A68B EXEC read one byte from file.
          Select EXEC file buffer.
          Copy file buffer addresses to file manager parmlist.
          Set state to 3 (input echo).
          Go to A62D to read a file byte.
A68C-A69C Read next text file byte.
          Set up file manager parmlist to read one byte.
          Call file manager driver.
          Return to caller with the data byte.
A69D-A6A7 Set $40,$41 to point to EXEC file buffer.
A6A8-A6C3 File manager driver routine.
          Call the file manager itself (AB06).
```

```
          If no errors, exit to caller.
          Otherwise, point $4Ø,$41 at file buffer.
          If found, release it by storing a zero on the file
          name field.
          If error was not "END OF DATA", print error message.
          Otherwise, pretend a $ØØ was read and return to
          caller.
A6C4-A6D4 Miscellaneous error messages.
     A6C4 "COMMAND SYNTAX ERROR"
     A6C8 "NO FILE BUFFERS AVAILABLE"
     A6CC "PROGRAM TOO LARGE"
     A6DØ "FILE TYPE MISMATCH"
A6D5-A7Ø1 Error handler.
          Set warmstart flag and clear status (BFE6).
          If APPLESOFT ONERR is active, go to A6EF.
          Otherwise, print RETURN BELL RETURN.
          Print text of error message (A7Ø2).
          Print another RETURN.
     A6EF Replace DOS intercepts.
          If a BASIC program is in execution, pass error code
          to BASIC's error handler.
          Otherwise, warmstart BASIC.
A7Ø2-A719 Print text of error message.
          Using the error number as an index, print the message
          text from the message table (A971) byte by byte.
          Last character has most significant bit on.
A71A-A742 Complete file manager parameter list.
          Copy Volume value to parmlist.
          Copy Drive value to parmlist.
          Copy Slot value to parmlist.
          Copy address of primary filename buffer to parmlist.
          Save file buffer address in $4Ø,$41.
          Return to caller.
A743-A74D Copy primary filename to file buffer filename field.
A74E-A75A Copy current buffer pointers to file manager parmlist
          Copy file manager workarea buffer pointer.
          Copy T/S List sector buffer pointer.
          Copy data sector buffer address.
          Copy next file buffer link address.
          Return to caller.
A75B-A763 Reset state to Ø and set warmstart flag.
A764-A791 Locate an open or free file buffer.
          Assume there are no free file buffers by zeroing $45.
          Point $4Ø,$41 at first buffer on chain.
          Go to A773.
     A76E Point $4Ø,$41 at next buffer on chain.
          If at end of chain, exit with file not open code.
     A773 Get first byte of filename field.
          If zero (file buffer free), save file buffer address
          at $44,$45 as an available buffer and go to A76E.
          Otherwise, see if name in primary filename buffer
          matches the name in this file buffer.
          If not, go to A76E to get next buffer.
          If so, return to caller with open file found code.
A792-A799 Point $4Ø,$41 at first file buffer on chain.
A79A-A7A9 Point $4Ø,$41 at next file buffer on chain.
A7AA-A7AE Get first byte of file name in file buffer.
A7AF-A7C3 See if current buffer belongs to EXEC.
          Is EXEC active?
```

```
          If not, exit.
          If so, does current buffer address match EXEC's?
          Return to caller with appropriate code.
A7C4-A7D3 Check file type.
          Does file type of open file match desired file type?
          If so, exit.
          Otherwise, turn lock bit off and test again.
          If ok, exit.
          Otherwise, close file and issue "FILE TYPE MISMATCH".
A7D4-A850 Initialize (build) DOS file buffer chain.
          Set $40,$41 to point to first buffer.
          Set counter to MAXFILES value.
     A7E5 Store zero on filename field to mark as free.
          Set up link pointers in buffer to point to file
          manager workarea (45 bytes prior to filename field).
          Set up link pointer to T/S List sector buffer (-256
          bytes from file manager workarea buffer).
          Set up link pointer to data sector buffer 256 bytes
          before that.
          Decrement counter.
          If zero, go to A82D to set HIMEM.
          Otherwise, set link to next file buffer as 38 bytes
          prior to data sector buffer.
          Go to A7E5 to set up next buffer.
     A82D Set link of last buffer to $0000.
          If INTEGER BASIC is active, go to A846.
          Otherwise, set APPLESOFT's HIMEM and STRING START
          pointers in zeropage to point just below the last
          buffer.
          Exit to caller.
     A846 Set INTEGER BASIC's HIMEM and PROGRAM START pointers
          to point just below the last buffer.
          Exit to caller.
A851-A883 Replace DOS keyboard/video intercept vectors.
          Is DOS keyboard (KSWL) vector still set?
          If so, go to A86A.
          Otherwise, save current KSWL vector ($38,$39) at
          AA55,AA56 and replace with DOS intercept routine's
          address.
     A86A Is DOS video (CSWL) vector still set?
          If so, exit to caller.
          Otherwise, save current CSWL vector ($36,$37) at
          AA53,AA54 and replace with DOS intercept routine's
          address.
          Exit to caller.
A884-A908 DOS command name text table.
          This table consists of the ASCII name for each DOS
          command in order of command index values, with the
          last character of each indicated by the MSB being
          on. Commands in order are:
            INIT,LOAD,SAVE,RUN,CHAIN,DELETE,LOCK,UNLOCK,CLOSE,
            READ,EXEC,WRITE,POSITION,OPEN,APPEND,RENAME,
            CATALOG,MON,NOMON,PR#,IN#,MAXFILES,FP,INT,BSAVE,
            BLOAD,BRUN,VERIFY.
          Example: INIT is $49 $4E $49 $D4 (I N I T)
A909-A940 Command valid keywords table.
          This table is used to determine which keywords are
          required or may be given for any DOS command.
          Each command has a two byte entry with 16 flags,
```

indicating which keywords may be given. The flag bit settings are as follows:

BIT	MEANING
Ø	Filename legal but optional
1	Command has no positional operand
2	Filename #1 expected
3	Filename #2 expected
4	Slot number positional operand expected
5	MAXFILES value expected as positional operand
6	Command may only be issued from within a program
7	Command may create a new file if file not found
8	C, I, O keywords legal
9	V keyword legal
1Ø	D keyword legal
11	S keyword legal
12	L keyword legal
13	R keyword legal
14	B keyword legal
15	A keyword legal

Thus, for a typical command, OPEN, where the value is $2378, bits 2, 6, 7, 9, 1Ø, 11, and 12 are set so the command has one filename operand, may only be issued from within a program, may create a new file, and the V, D, S, and L keywords are legal.

The command entries are:

INIT	217Ø
LOAD	AØ7Ø
SAVE	A17Ø
RUN	AØ7Ø
CHAIN	2Ø7Ø
DELETE	2Ø7Ø
LOCK	2Ø7Ø
UNLOCK	2Ø7Ø
CLOSE	6ØØØ
READ	22Ø6
EXEC	2Ø74
WRITE	22Ø6
POSITION	22Ø4
OPEN	2378
APPEND	227Ø
RENAME	3Ø7Ø
CATALOG	4Ø7Ø
MON	4Ø8Ø
NOMON	4Ø8Ø
PR#	Ø8ØØ
IN#	Ø8ØØ
MAXFILES	Ø4ØØ
FP	4Ø7Ø
INT	4ØØØ
BSAVE	2179
BLOAD	2Ø71
BRUN	2Ø71
VERIFY	2Ø7Ø

A941-A94A Keyword name table.

This table contains all the ASCII names of the DOS keywords in standard order. Each keyword name occupies one byte:

V,D,S,L,R,B,A,C,I,O

A94B-A954 Keyword flag bit positions table.

```
          This table gives the bit positions for each keyword
          into the second byte of the command valid keyword
          table above and in the flag (AA65) which indicates
          which keywords were present on the command line.
          The bit positions are:
               V - 40
               D - 20
               S - 10
               L - 08
               R - 04
               B - 02
               A - 01
               C - C0 ...
               I - A0 ... not used in valid keyword table
               O - 90 ...
A955-A970 Keyword value valid range table.
          This table indicates the range any keyword value
          may legally have. Each keyword has a four byte entry,
          two bytes of minimum value, and two bytes of maximum
          value. Values are:
               KEYWORD   MIN  MAX
                  V       0   254
                  D       1     2
                  S       1     7
                  L       1  32767
                  R       0  32767
                  B       0  32767
                  A       0  65535
          C, I, and O do not appear in this table since they
          do not have numeric values.
A971-AA3E Error message text table.
          This table contains the text for each error code in
          order of error code number:
          NUMBER    TEXT
            0       RETURN BELL RETURN
            1       "LANGUAGE NOT AVAILABLE"
            2       "RANGE ERROR" (Bad file manager opcode)
            3       "RANGE ERROR" (Bad file manager subcode)
            4       "WRITE PROTECTED
            5       "END OF DATA"
            6       "FILE NOT FOUND"
            7       "VOLUME MISMATCH"
            8       "I/O ERROR"
            9       "DISK FULL"
           10       "FILE LOCKED"
           11       "SYNTAX ERROR"
           12       "NO BUFFERS AVAILABLE"
           13       "FILE TYPE MISMATCH"
           14       "PROGRAM TOO LARGE"
           15       "NOT DIRECT COMMAND"
AA3F-AA4F Error message text offset index table.
          This table contains the offset in bytes to the text
          of any given error message in the table above.
          Entries are one byte each for each error code number.
AA4F-AA65 DOS main routines variables.
     AA4F Current file buffer address (2 bytes).
     AA51 Status flags: $01=READ state, $00=Warmstart,
                        $80=Coldstart,  $40=APPLESOFT RAM
     AA52 DOS CSWL intercept state number.
```

```
     AA53 Address of true CSWL handler (2 bytes).
     AA55 Address of true KSWL handler (2 bytes).
     AA57 MAXFILES value.
     AA59 Save area for S, X, Y, and A registers when DOS is
          entered (4 bytes).
     AA5D Command line index value (offset into line).
     AA5E MON flags: (C=$4Ø, I=$2Ø, O=$1Ø)
     AA5F Index of last command times 2.
     AA6Ø Range length for LOAD and BLOAD (2 bytes).
     AA62 Index of pending command, if any.
     AA63 Scratch variable (counter, message index, etc.)
     AA64 Index of current keyword.
     AA65 Keywords present on command line flags.
AA66-AA74 Keyword values parsed from command and defaulted.
     AA66 Volume (2 bytes)
     AA68 Drive (2 bytes)
     AA6A Slot (2 bytes)
     AA6C Length (2 bytes)
     AA6E Record (2 bytes)
     AA7Ø Byte (2 bytes)
     AA72 Address (2 bytes)
     AA74 MON value (one byte)
AA75-AA92 Primary file name buffer
AA93-AABØ Secondary (RENAME) file name buffer
AAB1-AACØ DOS main routines constants and variables.
     AAB1 MAXFILES default ($Ø3).
     AAB2 Control-D ($84).
     AAB3 EXEC file active flag ($ØØ=not active).
     AAB4 EXEC file buffer address (2 bytes).
     AAB6 Active BASIC flag: $ØØ=INTEGER, $4Ø=APPLESOFT ROM,
                             $8Ø=APPLESOFT RAM
     AAB7 RUN intercepted flag.
     AAB8 "APPLESOFT" characters in ASCII (9 bytes)
AAC1-AAC8 File manager constants.
     AAC1 Address of RWTS paramter list (B7E8).
     AAC3 Address of VTOC sector buffer (B3BB).
     AAC5 Address of directory sector buffer (B4BB).
     AAC7 Address of last byte of DOS plus one. (CØØØ)
AAC9-AAE4 File manager function routine entry point table.
          This table contains a two byte function handler
          routine address for each of the 14 file manager
          opcodes in opcode order.
AAE5-AAFØ File manager read subcode handler entry point table.
          This table contains a two byte function handler
          routine address for each of the 6 read subcodes.
AAF1-AAFC File manager write subcode handler entry point table.
          This table contains a two byte function handler
          routine address for each of the 6 write subcodes.
AAFD-ABØ5 File manager external entry point (from $3D6).
          Is X register zero?
          If so, allow new files by simulating an INIT command
          index.
          Otherwise, require old file by simulating a LOAD
          command index.
          Fall through to main file manager entry point.
ABØ6-AB1E File manager main entry.
          Save S register at B39B.
          Restore file manager workarea from file buffer (AE6A)
          Make sure opcode does not exceed 13.
```

```
          If it does, return with code=2 (invalid opcode).
          Use opcode as index into file manager function
          routine entry point table and go to proper handler
          via RTS.
AB1F-AB21 Return with return code=2 (bad opcode).
AB22-AB27 OPEN function handler.
          Call common open code (AB28).
          Exit file manager.
AB28-ABDB Common open routine.
          Initialize file manager workarea by resetting
          variables to their defaults (ABDC).
          Set sector length to 256.
          Insure record length is non-zero. If zero, use 1.
          Store record length in file manager workarea.
          Locate or allocate a directory entry for the file
          (B1C9).
          If file already exists, go to ABA6.
          Otherwise, save directory index for free entry.
          Using last command index and valid keywords table,
          determine whether current command may create a new
          file.
          If so, go to AB64.
          Otherwise, if running "APPLESOFT", set return code
          to "LANGUAGE NOT AVAILABLE" and exit.
          If not running "APPLESOFT" set return code to "FILE
          NOT FOUND" and exit.
     AB64 Set sector count in directory entry to 1 (there will
          only be a T/S List sector initially).
          Allocate a sector for a T/S List (B244).
          Store sector number of this sector in directory
          entry and in first and current T/S List sector number
          in file manager workarea.
          Store track number in both places also.
          Move file type desired to directory entry.
          Write directory sector back to catalog (BØ37).
          Select T/S List buffer (AFØC).
          Zero it (B7D6).
          And write it back (AF3A).
          Set return code to 6 ("FILE NOT FOUND").
     ABA6 Place track/sector of T/S List in directory entry in
          first T/S List variable in file manager workarea.
          Copy file type from directory to parmlist to pass it
          back to caller and to file manager workarea.
          Copy number of sectors in file to workarea.
          Save directory offset to entry in workarea.
          Set end of data pointer to "infinity".
          Set number of data bytes represented by one T/S List
          sector to 122*256 (3Ø.5K) in workarea.
          Go read first T/S List sector (AF5E).
ABDC-ACØ5 Initialize file manager workarea.
          Zero entire 45 bytes of workarea.
          Save complemented volume number in workarea.
          Save drive number in workarea.
          Save slot*16 in workarea.
          Set track number to $11 (catalog track).
          Return to caller.
ACØ6-AC39 CLOSE function handler.
          Checkpoint data buffer to disk if needed (AF1D).
          Checkpoint T/S List buffer if needed (AF34).
```

```
          Release any sectors which were preallocated but not
          used (B2C3).
          If VTOC does not need to be re-read, exit.
          Otherwise, re-read VTOC sector (AFF7).
          Flush through directory sectors in the catalog until
          we reach the one which contains the entry for this
          file.
          Get the index to the entry.
          Update the sector count in the entry to reflect the
          new file's length.
          Checkpoint the directory sector back to the disk.
          Exit file manager.
AC3A-AC57 RENAME function handler.
          Call common code to locate/open the file.
          If file is locked, exit with "FILE LOCKED" return
          code.
          Set $42,$43 to point to new name.
          Copy new name to directory entry.
          Write back directory sector to disk.
          Exit file manager.
AC58-AC69 READ function handler.
          Insure subcode does not exceed 5. If so, exit with
          return code=3.
          Use subcode as index into READ subcode handler entry
          point table.
          Go to proper handler of subcode.
AC6A-AC6C Return code = 3, subcode bad
AC6D-AC6F "FILE LOCKED" error return
AC7Ø-AC86 WRITE function handler.
          If file is locked, exit with "FILE LOCKED" error.
          Insure subcode does not exceed 5. If so, exit with
          return code=3.
          Use subcode as index into WRITE subcode handler entry
          point table.
          Go to proper handler of subcode.
AC87-AC89 POSITION AND READ ONE BYTE subcode handler
          Call position routine.
          Fall through to next subcode handler.
AC8A-AC92 READ ONE BYTE subcode handler.
          Read next file byte (ACA8).
          Store in parmlist for pass back to caller.
          Exit the file manager.
AC93-AC95 POSITION AND READ A RANGE OF BYTES subcode handler.
          Call position routine.
          Fall through to next subcode handler.
AC96-ACA7 READ A RANGE OF BYTES subcode handler.
          Decrement and check length (B1B5).
          Read a byte (ACA8).
          Point $42,$43 at range address and add one to address
          Store byte read at address.
          Loop back to AC96. (Length check will exit file
          manager when length is zero.)
ACA8-ACB8 Read a data byte.
          Read next data sector if necessary (BØB6).
          If at end of file, exit with "END OF DATA" error.
          Otherwise, load data byte from data sector buffer.
          Increment record number/byte offset into file (B15B).
          Increment file position offset (B194).
          Return with data byte read.
```

```
ACBB-ACBD POSITION AND WRITE ONE BYTE subcode handler.
          Call position routine.
          Fall through to next subcode handler.
ACBE-ACC6 WRITE ONE BYTE subcode handler.
          Find data byte to be written.
          Write it to file (ACDA).
          Exit file manager.
ACC7-ACC9 POSITION AND WRITE A RANGE OF BYTES subcode handler.
          Call position routine.
          Fall through to next subcode handler.
AACA-ACD7 WRITE A RANGE OF BYTES subcode handler.
          Copy and advance range address pointer.
          Get next byte to write.
          Write it to file (ACDA).
          Test and decrement length (B1B5).
          Loop back to AACA.
ACDA-ACEC Write a data byte.
          Read the proper data sector (if necessary) (BØB6).
          Store data byte to be written in sector buffer.
          Flag data sector buffer as requiring rewrite.
          Increment record number/byte offset into file (B15B).
          Exit via file position offset increment routine
          (B194).
ACEF-ACF5 LOCK function handler.
          Set mask byte to $8Ø (lock).
          Go to common code (ACFB).
ACF6-ACFA UNLOCK function handler.
          Set mask byte to $ØØ (unlock).
          Fall through to common code.
ACFB-AD11 LOCK/UNLOCK common code.
          Locate/open file (AB28).
          Get index into directory to entry.
          Update file type byte to lock ($8X) or unlock ($ØX).
          Write directory sector back to disk.
          Exit file manager.
AD12-AD17 POSITION function handler.
          Call position routine.
          Exit file manager.
AD18-AD2A VERIFY function handler.
          Locate/open file (AB28).
     AD1B Read next data sector.
          If at end of file, exit file manager.
          Otherwise, increment sector position.
          And loop back to AD1B.
AD2B-AD88 DELETE function handler.
          Locate/open file (AB28).
          Using directory index, determine if file is locked.
          If so, exit with "FILE LOCKED" error code.
          Copy T/S List sector's track number from directory to
          workarea and to last character of file name in the
          directory entry itself.
          Store a $FF over T/S List sector's track number in
          directory entry to mark file deleted.
          Copy T/S List sector's sector number to workarea.
          Write directory sector back to disk.
     AD54 Read next T/S List sector (AF5E).
          If no more exist, write VTOC and exit file manager.
          Otherwise, select T/S List buffer (AFØC).
          Index to first T/S pair.
```

```
     AD5E If track number is zero or minus, skip it.
          Otherwise, free the data sector by updating the VTOC
          bit map (AD89).
          Index to next T/S pair.
          If more, go to AD5E.
          Get T/S of next T/S List sector from this one.
          Free this T/S List sector (AD89).
          Go process next one, if any (go to AD54).
          Otherwise, write VTOC and exit file manager.
AD89-AD97 Free a sector.
          Call B2DD to deallocate sector in VTOC bit map.
          Zero the sector allocation area of the workarea.
          Return to caller.
AD98-AE2E CATALOG function handler.
          Initialize file manager workarea (ABDC).
          Set V value to zero (complimented=$FF).
          Read the VTOC sector (AFF7).
          Set up a counter for 22 lines before waiting for
          the keyboard.
          Skip 2 lines on the screen.
          Print "DISK VOLUME ".
          Convert Volume number and print it (AE42).
          Skip 2 more lines.
     ADCA Read next directory sector.
          If no more exist, exit file manager.
          Set index to first entry.
     ADD1 Get track number.
          If zero, exit file manager.
          If minus, skip entry (deleted file).
          Print "*" if file is locked (check file type byte).
          Use file type as index into file type name table at
          B3A7 and print single character found there.
          Print a blank.
          Convert and print the number of sectors in the file.
          Print a blank.
          Index to filename.
          Print file name.
          Skip to next line.
          Advance index to next directory entry.
          If there are more, go to ADD1.
          If not, go to ADCA to read next directory sector.
          Exit when finished.
AE2F-AE41 Skip a line on CATALOG printout.
          Output a carriage return.
          Decrement line counter.
          If still nonzero, exit.
          Otherwise, wait for keyboard keypush.
          Then reset counter to 21 lines.
          And return to caller.
AE42-AE69 Convert the number stored at $44 to a three character
          printable number and print it.
AE6A-AE7D Restore file manager workarea from file buffer.
          Select file manager workarea buffer.
          Set return code in parmlist to zero initially.
          Copy 45 byte saved image of file manager workarea in
          file buffer to real file manager workarea.
          Exit to caller.
AE7E-AE8D Save file manager workarea in file buffer.
          Select file manager workarea buffer.
```

```
          Copy 45 byte workarea to file buffer.
          Exit to caller.
AE8E-AFØ7 INIT function handler.
          Initialize the file manager workarea (ABDC).
          Call RWTS to format the diskette (BØ58).
          Copy V value to VTOC buffer.
          Start track to allocate next value at $11.
          And direction of allocation as $Ø1 (forward).
          Zero VTOC bit map (all sectors in use).
          Skipping the first three tracks and track $11, copy
          the 4 byte bit mask (B3AØ) to each track entry in
          the VTOC bit map to free the sectors. This leaves the
          first three tracks and the catalog track marked in
          use.
          Zero the directory sector buffer.
          Point to directory sector buffer.
          Set track $11 in RWTS parmlist.
          Set up link from this directory sector to next (track
          $11, sector-1).
          Call RWTS to write directory sector.
          Write each sector on track in this way except for
          sector zero.
          On last sector (sector 1) zero link pointer.
          Point RWTS parms at DOS load point (B7C2).
          Write DOS image onto tracks Ø-2 (B74A).
          Exit file manager.
AFØ8-AF1C Select a buffer by setting $42,$43 to point to it.
     AFØ8 Select file manager workarea buffer in file buffer.
     AFØC Select T/S List sector buffer in file buffer.
     AF1Ø Select data sector buffer in file buffer.
          Exit to caller when $42,$43 are set.
AF1D-AF33 Checkpoint write data sector buffer to disk.
          Test flag to see if buffer was changed since last
          read/write.
          If not, exit to caller.
          Otherwise, set up RWTS pointer (AFE4).
          Call RWTS to write sector.
          Reset flag to indicate data sector no longer in need
          of a checkpoint.
          Exit to caller.
AF34-AF4A Checkpoint write T/S List sector buffer to disk.
          Test flag to see if buffer was changed since last
          read/write.
          If not, exit to caller.
          Otherwise, set up RWTS pointer (AF4B).
          Call RWTS to write sector.
          Reset flag to indicate T/S List sector no longer in
          need of checkpoint.
          Exit to caller.
AF4B-AF5D Prepare for RWTS call with a T/S List sector.
          Copy address of T/S List buffer to RWTS parmlist.
          Get track/sector of sector.
          Exit to caller.
AF5E-AFDB Read a T/S List sector to file buffer.
          (CARRY flag is set at entry to indicate whether the
          first T/S List for the file is wanted (C=Ø) or the
          next (C=1).
          Memorize carry flag entry code.
          Checkpoint current T/S List sector if necessary.
```

```
          Set up for RWTS (AF4B).
          Select T/S List buffer (AFØC).
          Is first or next wanted?
          If first, go to AFB5 to continue.
          Otherwise, get link to next T/S List from this one.
          If link is non-zero, use it to find next one and go
          to AFB5.
          Otherwise, we are out of T/S Lists for this file.
          If we are reading file, exit with error code.
          Otherwise, allocate a new sector (B244).
          Point old T/S List sector to new one's track/sector.
          Write old T/S List sector back to disk.
          Zero the buffer to form new T/S List sector.
          Compute and store the relative sector number of the
          first sector listed in this sector at +5,+6 into the
          buffer.
          Set RWTS opcode to write new T/S List sector to disk.
     AFB5 Set RWTS opcode to read old T/S List (unless we just
          allocated it above).
          Set track and sector and call RWTS to read old list
          or write new list.
          Compute relative sector number of last sector (plus
          one) in this list and store in workarea.
          Exit to caller with normal return code.
AFDC-AFE3 Read a data sector.
          Set up for RWTS (AFE4).
          Set RWTS READ opcode and go to RWTS driver to do it.
AFE4-AFF6 Prepare for RWTS with data sector.
          Copy address of data sector buffer to RWTS parmlist.
          Get its track/sector.
          And exit to caller.
AFF7-BØ1Ø Read/write the VTOC buffer.
     AFF7 Read VTOC entry, go to AFFD.
     AFFB Write VTOC entry, fall through.
     AFFD Common code.
          Copy VTOC sector buffer address to RWTS parmlist.
          Get its track number and use sector $ØØ.
          Exit through RWTS driver.
BØ11-BØ36 Read a directory sector.
          (If CARRY flag is zero on entry, read first directory
          sector. If CARRY is one, read next.)
          Memorize entry code.
          Set buffer pointers (BØ45).
          First or next?
          If first, get track/sector of directory sector from
          VTOC at offset +1,+2.
          Otherwise, get track/sector from directory sector at
          offset +1,+2. If track is zero, exit with error code
          (end of directory).
          Call RWTS to read sector.
          Exit with normal return code.
BØ37-BØ44 Write directory sector.
          Set buffer pointers.
          Find its track/sector in workarea.
          Exit through RWTS to write it.
BØ45-BØ51 Prepare for RWTS for directory buffer.
          Copy directory buffer address to RWTS parmlist.
          Exit to caller.
BØ52-BØB3 Read/Write Track/Sector (RWTS) driver.
```

```
            Set track/sector in RWTS parmlist.
      BØ58  Set command code (read, write, etc.).
            If writing, set flag (B5D5).
            Set volume number expected in parmlist.
            Set slot*16 in parmlist.
            Set drive in parmlist.
            Set sector size in parmlist.
            Set IOB type in parmlist ($Ø1).
            Call RWTS, passing parmlist pointer.
            Copy true volume found to file manager parmlist.
            Reset volume expected field in RWTS parmlist.
            If an error did not occur, exit to caller.
            Otherwise, get return code.
            Translate vol mismatch to rc=7, write protected to
            rc=4 and all other errors to rc=8 (I/O error).
            Exit file manager now.
BØB6-B133   Read next data sector (if necessary).
            Is the current file position in the current data
            sector now in memory?
            If so, go to B12C.
            Otherwise, checkpoint data sector buffer.
            Is the current file position prior to or after this
            T/S List's domain?
            If not, go to BØF3.
            Otherwise, read each T/S List for the file, starting
            with the first, until the proper one is found.
            If it is never found, exit with error (ran off end of
            file reading).
      BØF3  Data is in this T/S List sector.
            Compute the displacement to the proper entry in this
            T/S List sector.
            Select the T/S List buffer.
            Get the track of the data sector wanted.
            If non-zero, go to B114.
            Otherwise, if not writing, exit with error (no data
            to read there).
            If writing, allocate a new sector and store its
            track/sector location in the list at this point
            (B134).
            Go to B12Ø.
      B114  Read old data sector, using the track/sector found
            in the T/S List entry.
      B12Ø  Save number of sector last read in workarea.
      B12C  Select data buffer.
            Get byte offset and exit normally to caller.
B134-B15A   Add a new data sector to file.
            Allocate a sector (B244).
            Put track/sector numbers in T/S List entry.
            Select data buffer and zero it.
            Set flags to indicate that the T/S List sector and
            the data sector buffer require checkpoints.
            Exit to caller.
B15B-B193   Increment record number and byte offset into file.
            Copy current record number and byte offset to file
            manager parameter list to pass back to caller.
            Increment byte offset in workarea.
            If byte offset equals record length, set byte offset
            back to zero and increment record number.
            Return to caller.
```

```
B194-B1A1 Increment file position offset.
          Increment byte offset into current sector by one.
          If at end of sector, increment sector number by one.
          Return to caller.
B1A2-B1B4 Copy and advance range address.
          Copy range address from file manager parmlist to $42.
          Increment range address in parmlist for next time
          through.
          Return to caller.
B1B5-B1C8 Decrement range length.
          Decrement range length in file manager parmlist by
          one.
          If zero, exit file manager.
          Otherwise, exit to caller.
B1C9-B21B Locate or allocate a directory entry in the catalog.
          Read the VTOC sector (AFF7).
          Set $42,$43 to point to file name we are looking for.
          Set pass number to one (locate file).
     B1D8 Initialize directory sector offset (first sector).
     B1E1 Increment sector offset.
          Read directory sector.
          If at end of directory, go to B23A.
          Set entry index to first file entry.
     B1EB Get track.
          If deleted, skip entry, go to B217.
          If empty, end of directory, go to B212.
          Advance index to filename in directory.
          Compare against filename wanted.
          If they match, return entry index and exit.
          If not, advance index to next entry in sector and
          loop back to B1EB.
          If at end of sector, go to B1E1 to get next sector.
     B212 If pass number is one, go to B1D8 to start second
          pass.
     B217 If pass number is one, go to B2ØB to skip entry.
          If second pass, fall through to allocate entry.
B21C-B22F Copy file name to directory entry.
          Advance index to file name field in directory entry.
          Copy 3Ø byte filename to directory entry.
          Reload directory index and return to caller.
B23Ø-B239 Advance index to next directory entry in sector.
          Add 35 (length of entry) to index.
          Test for end of sector and return to caller.
B23A-B243 Switch to second pass in directory scan.
          If on pass one, switch to pass 2 and go to B1D8.
          If on pass two, exit file manager with "DISK FULL"
          error.
B244-B2C2 Allocate a disk sector.
          Is there a track currently allocated to this file?
          If not, go to B26A to find a track with free sectors.
     B249 Otherwise, decrement sector number to get next
          possible free sector number.
          If there are no more sectors on this track, go to
          B265 to find a new track.
          Otherwise, rotate the track bit mask by one position
          and get the bit for this sector.
          If the sector is in use, loop back to B249.
          Otherwise, add one to file's sector count.
          Pass back sector number (track number is at B5F1).
```

```
           And return to caller.
      B265 Indicate no track is being used at present.
      B26A Reset allocation flag to allow at least one complete
           search of all tracks for some space.
           Read VTOC sector.
      B272 Get last track allocated from and add direction value
           to get next track to examine (+1 or -1).
           Are we back to track Ø?
           If so, go to B284.
           Otherwise, are we past track 34?
           If so, reverse direction and go to B28E.
      B284 Is this the second time we have come to track Ø ?
           (check allocation flag).
           If so, exit with "DISK FULL" error.
           Otherwise, set allocation flag to remember this.
           Set direction to forward (+1).
      B28E Begin at directory track (17 + or - 1).
           Compute bit map index (tracknumber*4).
           Copy track bit map from VTOC to workarea, watching
           to see if all four bytes are zero (track is full).
           In any case, set all four bytes in VTOC to zero
           (allocate all sectors).
           If no free sectors in the track, go to B272 to try
           next track.
           Otherwise, write VTOC to disk to insure file's
           integrity.
           Set sector number to last sector in track.
           Go to B249 to allocate one of its free sectors to
           the file.
 B2C3-B2DC Release pre-allocated sectors in current track and
           checkpoint the VTOC.
           Has a track been allocated to the file?
           If not, exit to caller.
           Otherwise, read VTOC.
           Get next sector which could have been used (number
           of times track map was shifted during allocation).
           Call B2DD to shift track bit map back and merge it
           back into the VTOC bit map.
           Indicate no track has been allocated.
           Exit to caller.
 B2DD-B2FF Free one or more sectors by shifting mask in file
           manager's allocation area back into VTOC bit map.
           (If CARRY is set, current sector is freed also)
           Rotate entire 4 byte track bit mask once.
           Repeat for as many sectors as were allocated.
           Compute index into VTOC for this track's map.
           If zero, exit.
           Merge ("OR") file manager's bits with those already
           in VTOC, freeing sectors which were never used by
           the file.
           Return to caller.
 B3ØØ-B35E Calculate file position.
           Set record number passed in file manager parmlist
           in workarea and in sector offsets.
           Clear sector offset high part.
           Perform a 16 bit multiply as follows:
           3 byte file position = record number times record
           length.
           Add the byte offset from the parmlist into the three
```

```
          byte file position value (B5E4,B5E5,B5E6).
          Return to caller.
B35F-B37D Error exits.
     B35F RC=1 "LANGUAGE NOT AVAILABLE"
     B363 RC=2 "RANGE ERROR" (bad opcode)
     B367 RC=3 "RANGE ERROR" (bad subcode)
     B36B RC=4 "WRITE PROTECTED"
     B36F RC=5 "END OF DATA"
     B373 RC=6 "FILE NOT FOUND"
     B377 RC=9 "DISK FULL" (all files closed)
     B37B RC=A "FILE LOCKED"
B37F-B396 Exit file manager.
     B37F Exit with no errors.
          Get return code of zero.
          Clear carry flag and go to B386.
     B385 Set carry flag to indicate error.
     B386 Save return code in parmlist.
          Clear monitor status register ($48) after RWTS has
          probably tromped on it.
          Save file manager workarea to file buffer (AE7E).
          Restore processor status and stack register.
          Exit to original caller of file manager.
B397-B3A3 File manager scratch space.
     B397 Track/sector of current directory sector (2 bytes).
     B39B S register save area.
     B39C Directory index.
     B39D Catalog line counter/Directory lookup flag/Etc.
     B39E LOCK/UNLOCK mask/Allocation flag/Etc.
     B3AØ Four byte mask used by INIT to free an entire track
          in the VTOC bit map.
B3A4-B3A6 Decimal conversion table (1,1Ø,1ØØ).
B3A7-B3AE File type name table used by CATALOG.
          File types are: T,I,A,B,S,R,A,B, corresponding to
          hex values: $ØØ, $Ø1, $Ø2, $Ø4, $Ø8, $1Ø, $2Ø, and
          $4Ø respectively.
B3AF-B3BA ASCII text "DISK VOLUME " backwards. Used by CATALOG.
B3BB-B4BA VTOC sector buffer.
     B3BC Track/sector of first directory sector.
     B3BE DOS release number (1, 2, or 3).
     B3C1 Volume number of diskette.
     B3E2 Number of entries in each T/S List sector.
     B3EB Track to allocate next.
     B3EC Direction of track allocation (+1 or -1)
     B3EF Number of tracks on a disk.
     B3FØ Number of sectors on a disk.
     B3F1 Sector size in bytes (2 bytes)
     B3F3 Track Ø bit map
     B3F7 Track 1 bit map
          etc.
     B47B Track 34 bit map
B4BB-B5BA DIRECTORY sector buffer.
     B4BC Track/sector of next directory sector.
     B4C6 First directory entry and
          Track of T/S List
     B4C7 Sector of T/S List
     B4C8 File type and lock bit
     B4C9 Filename field (3Ø bytes)
     B4E7 Size of file in sectors (including T/S List(s)).
B5BB-B5DØ File manager parameter list.
```

```
     B5BB Opcode
     B5BC Subcode
     B5BD Eight bytes of variable parameters depending on
          opcode.
     B5C5 Return code.
     B5C7 Address of file manager workarea buffer.
     B5C9 Address of T/S List sector buffer.
     B5CB Address of data sector buffer.
     B5CD Address of next DOS buffer on chain (not used).
B5D1-B5FD File manager workarea.
     B5D1 1st T/S List sector's track/sector.
     B5D3 Current T/S List sector's track/sector.
     B5D5 Flags: 8Ø=T/S List needs checkpoint
                 4Ø=Data sector needs checkpoint
                 2Ø=VTOC sector needs checkpoint
                 Ø2=Last operation was write
     B5D6 Current data sector's track/sector.
     B5D8 Directory sector index for file entry.
     B5D9 Index into directory sector to directory entry for
          file.
     B5DA Number of sectors described by one T/S List.
     B5DC Relative sector number of first sector in list.
     B5DE Relative sector number +1 of last sector in list.
     B5EØ Relative sector number of last sector read.
     B5E2 Sector length in bytes.
     B5E4 File position (3 bytes) sector offset, byte offset
          into that sector.
     B5E8 Record length from OPEN.
     B5EA Record number.
     B5EC Byte offset into record.
     B5EE Number of sectors in file.
     B5FØ Sector allocation area (6 bytes).
          Next sector to allocate (shift count)
          Track being allocated
          Four byte bit map of track being allocated, rotated
          to next sector to allocate.
     B5F6 File type.
     B5F7 Slot number times 16.
     B5F8 Drive number.
     B5F9 Volume number (complemented).
     B5FA Track number.
B5FE-B5FF Not used.
B6ØØ-B6FF Start of Boot 2/RWTS image.
     B6ØØ Boot 1 image which can be written to INITed disks
          on track Ø, sector Ø.
     B65D DOS 3.3 patch area.
     B65D APPEND patch flag.
     B65E APPEND patch. Come here when file manager driver
          gets an error other than end of data.
          Locate and free the file buffer.
          Clear the APPEND flag.
          Get the error number and go print error (A6D2).
     B671 APPEND patch. Come here from APPEND command handler
          to increment record number if APPEND flag is set and
          to clear the flag. Exit through POSITION.
     B686 VERIFY patch. Come here from I/O a range of bytes
          routine to exit through VERIFY after SAVE or BSAVE.
     B692 APPEND patch. Come here from file manager driver if
          return code was END OF DATA.
```

```
          Test the file position for zero.
          If non-zero, set APPEND flag on and return to caller.
          If zero (at start of file), copy record number and
          byte offset to file manager parmlist and return a
          zero data byte to caller.
     B6FE Page address of first page in Boot 2.
     B6FF Number of sectors (pages) in Boot 2.
B7ØØ-B749 DOS 2nd stage boot loader.
          Set RWTS parmlist to read DOS from disk.
          Call Read/Write group of pages ($B793).
          Create new stack.
          Call SETVIC ($FE93) and SETKBD ($FE89).
          Exit to DOS coldstart ($9D84).
B74A-B78C Put DOS on tracks Ø-2.
          Set RWTS parmlist to write DOS to disk.
          Call Read/Write group of pages ($B793).
          Exit to caller.
B78D-B792 Unused.
B793-B7B4 Read/Write a group of pages.
          call RWTS through external entry point ($B7B5).
          Exit to caller.
B7B5-B7C1 Disable interrupts and call RWTS.
B7C2-B7D5 Set RWTS parameters for writing DOS.
B7D6-B7DE Zero current buffer.
          Zero 256 bytes pointed to by $42,$43.
          Exit to caller.
B7DF-B7E7 DOS 2nd stage boot loader parmlist.
     B7DF Unused.
     B7EØ Number of pages in 2nd DOS load.
     B7E1 Number of sectors to read/write.
     B7E2 Number of pages in 1st DOS load.
     B7E3 INIT DOS page counter.
     B7E4 Pointer to RWTS parmlist (2 bytes).
     B7E6 Pointer to 1st stage boot location (2 bytes).
B7E8-B7F8 RWTS parmlist.
     B7E8 Table type. Must be $Ø1.
     B7E9 Slot number times 16.
     B7EA Drive number ($Ø1 or $Ø2).
     B7EB Volume number expected (Ø matches any volume).
     B7EC Track number ($ØØ to $22).
     B7ED Sector number ($ØØ to $ØF).
     B7EE Pointer to Device Characteristics Table (2 bytes).
     B7FØ Pointer to user data buffer for READ/WRITE (2 bytes).
     B7F2 Unused.
     B7F3 Byte count for partial sector (use $ØØ for 256).
     B7F4 Command code: Ø=SEEK, 1=READ, 2=WRITE, 4=FORMAT.
     B7F5 Error code:(valid if carry set) $1Ø=Write protect,
          $2Ø=Volume mismatch, $4Ø=Drive error, $8Ø=Read error.
     B7F6 Volume number found.
     B7F7 Slot number found.
     B7F8 Drive number found.
B7F9-B7FA Unused.
B7FB-B7FE Device Characteristics Table (DCT).
     B7FB Device type (should be $ØØ).
     B7FC Phases per track (should be $Ø1).
     B7FD Motor on time count (2 bytes - should be $EF, $D8).
B7FF      Unused.
B8ØØ-B829 PRENIBBLE routine.
          Converts 256 (8 bit) bytes to 342 (6 bit) "nibbles"
```

```
          of the form ØØXXXXXX.
          Pointer to page to convert stored at $3E,$3F.
          Data stored at primary and secondary buffers.
          On entry: $3E,$3F contain pointer to user data.
          On exit:  A-reg:unknown
                    X-reg:$FF
                    Y-reg:$FF
                    Carry set
          Exit to caller.
B82A-B8B7 WRITE routine.
          Writes prenibbilized data from primary and secondary
          buffers to disk.
          Calls Write a byte subroutine.
          Writes 5 bytes autosync, starting data marks
          ($D5/$AA/$AD), 342 bytes data, one byte checksum, and
          closing data marks ($DE/$AA/$EB).
          Uses Write Translate Table ($ba29).
          On entry: X-reg:Slot number times 16
          On exit:  Carry set if error
                    If no error:
                    A-reg:unknown
                    X-reg:unchanged
                    Y-reg:$ØØ
                    Carry clear
                    Uses $26,$27,$678
          Exit to caller.
B8B8-B8C1 Write a byte subroutine.
          Timing critical code used to write bytes at 32 cycle
          intervals.
          Exit to caller.
B8C2-B8DB POSTNIBBLE routine.
          Converts 342 (6 bit) "nibbles" of the form ØØXXXXXX
          to 256 (8 bit) bytes.
          Nibbles stored at primary and secondary buffers.
          Pointer to data page stored at $3E,$3F.
          On entry: X-reg:Slot number times 16
                    $36,$37:pointer to user data
                    $26:byte count in secondary buffer ($ØØ)
          On exit:  A-reg:unknown
                    X-reg:unknown
                    Y-reg:byte count in secondary buffer
                    Carry set
          Exit to caller.
B8DC-B943 READ routine.
          Read a sector of data from disk and store it at
          primary and secondary buffers.  (First uses secondary
          buffer high to low, then primary low to high)
          On entry: X-reg:Slot times 16
                    Read mode (Q6L,Q7L)
          On exit:  Carry set if error.
                    If no error:
                    A-reg:$AA
                    X-reg:unchanged
                    Y-reg:$ØØ
                    Carry clear
                    Uses $26
          Exit to caller.
B944-B99F RDADR routine.
          Read an Address Field.
```

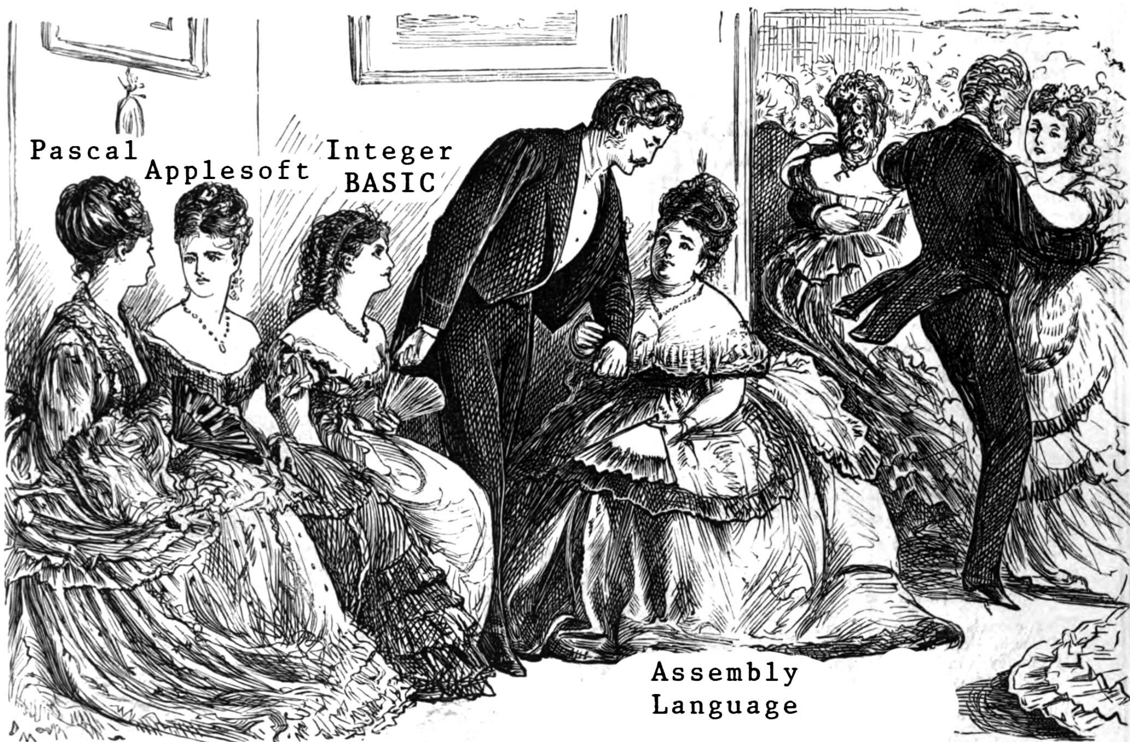
```
          Reads starting address marks ($D5/$AA/$96), address
          information (volume/track/sector/checksum), and
          closing address marks ($DE/$AA).
          On entry: X-reg:Slot number times 16
                    Read mode (Q6L,Q7L)
          On exit:  Carry set if error.
                    If no error:
                    A-reg:$AA
                    X-reg:unchanged
                    Y-reg:$ØØ
                    Carry clear
                    $2F: Volume number found
                    $2E: Track number found
                    $2D: Sector number found
                    $2C: Checksum found
                    Uses $26,$27
          Exit to caller.
B9AØ-B9FC SEEKABS routine.
          Move disk arm to desired track.
          Calls arm move delay subroutine ($B9FD).
          On entry: X-reg:Slot number times 16
                    A-reg:Desired track (halftrack for single
                          phase disk).
                    $478:Current track.
          On exit:  A-reg:unknown
                    X-reg:unchanged
                    Y-reg:unknown
                    $2A and $478:Final track
                    $27:Prior track (if seek needed)
                    Uses: $26,$27,$2A,$2B
          Exit to caller.
B9FD-BA1Ø Arm move delay subroutine.
          Delays a specified number of 1ØØ Usec intervals.
          On entry: A-reg:number of 1ØØ Usec intervals.
                    $46,$47:Should contain motor on time count
                    ($EF,$D8) from Device Characteristics Table
                    $478:Current track.
          On exit:  A-reg:$ØØ
                    X-reg:$ØØ
                    Y-reg:unchanged
                    Carry set
          Exit to caller.
BA11-BA28 Arm move delay table.
          Contains values of 1ØØ μsec intervals used during
          Phase-on and Phase-off of stepper motor.
BA29-BA68 Write Translate Table.
          Contains 6 bit "nibbles" used to convert 8 bit bytes.
          Values range from $96 to $FF.
          Codes with more than one pair of adjacent zeros or
          with no adjacent ones are excluded.
BA69-BA95 Unused.
BA96-BAFF Read Translate Table.
          Contains 8 bit bytes used to convert 6 bit "nibbles".
          Values range from $96 to $FF.
          Codes with more than one pair of adjacent zeros or
          with no adjacent ones are excluded.
BBØØ-BBFF Primary Buffer.
BCØØ-BC55 Secondary Buffer.
BC56-BCC3 Write Address Field during initialization.
```

```
          Calls Write double byte subroutine.
          Writes number of autosync bytes contained in Y-reg,
          starting address marks ($D5/$AA/$96), address
          information (volume/track/sector/checksum), closing
          address marks ($DE/$AA/$EB).
          On entry: X-reg:Slot number times 16
                    Y-reg:number of autosync to write
                    $3E: $AA
                    $3F: sector number
                    $41: volume number
                    $44: track number
          On exit:  A-reg:unknown
                    X-reg:unchanged
                    Y-reg:$ØØ
                    Carry set
          Exit to caller.
BCC4-BCDE Write double byte subroutine.
          Timing critical code that encodes address information
          into even and odd bits and writes it at 32 cycle
          intervals.
          Exit to caller.
BCDF-BCFF Unused.
BDØØ-BD18 Main entry to RWTS.
          Upon entry, store Y-reg and A-reg at $48,$49 as
          pointers to the IOB.
          Initialize maximum number of recals at 1 and seeks
          at 4.
          Check if the slot number has changed.  If not,
          branch to SAMESLOT at $BD34.
BD19-BD33 Update slot number in IOB and wait for old drive
          to turn off.
BD34-BD53 SAMESLOT
          Enter read mode and read with delays to see if disk
          is spinning.
          Save result of test and turn on motor just in case.
BD54-BD73 Move pointers in IOB to zero page for future use.
          Device Characteristics Table pointer at $3C,$3D
          and data buffer pointer at $3E,$3F.
          Set up $47 (motor on time) with $D8 from DCT.
          Check if the drive number has changed.  If not,
          branch to $BD74.
          If so, change test results to show drive off.
BD74-BD8F Select appropriate drive and save drive being used
          as high bit of $35.  1=drive 1, Ø=drive 2.
          Get test results. If drive was on, branch to $BD9Ø.
          Wait for capacitor to discharge using MSWAIT
          subroutine at $BAØØ.
BD9Ø-BDAA Get destination track and go to it using MYSEEK
          subroutine at $BE5A.
          Check test result again and if drive was on,
          branch to TRYTRK at $BDAB.
          Delay for motor to come up to speed.
BDAB-BDBB TRYTRK
          Get command code.
          If null, exit through ALLDONE at $BE46, turning drive
          off and returning to caller.
          If =4, branch to FORMDSK at $BEØD.
          Otherwise, move low bit into carry (set=read,
          clear=write) and save value on status reg.
```

```
          If write operation, data is prenibbilized via a call
          to PRENIB16 at $B8ØØ.
BDBC-BDEC Initialize maximum retries at 48 and read an
          Address Field via RDADR16 at $B944.
          If read was good, branch to RDRIGHT at $BDED.
          If bad read, decrement retries, and, if still some
          left try again. Else, prepare to recalibrate.
          Decrement recal count.  If no more, then indicate
          drive error via DRVERR at $BEØ4.
          Otherwise, reinitialize reseeks at 4 and recalibrate
          arm.  Move to desired track and try again.
BDED-BEØ3 RDRIGHT
          Verify on correct track.  If so branch to RTTRK
          at $BE1Ø.
          If not, set correct track via SETTRK subroutine at
          $BE95 and decrement reseek count.
          If not zero then reseek track.  If zero, then recal.
BEØ4-BEØA DRVERR
          Clean up stack and status reg.
          Load A-reg with $4Ø (drive error)
          Goto HNDLERR at $BE48.
BEØB-BEØC Used to branch to ALLDONE at $BE46.
BEØD-BFØF FORMDSK
          Jump to DSKFORM at $BEAF.
BE1Ø-BE25 RTTRK
          Check volume number found against volume number
          wanted.
          If no volume was specified, then no error.
          If specified volume doesn't match, load A-reg with
          $2Ø (volume mismatch error) and exit via HNDLERR
          at $BE48.
BE26-BE45 CRCTVOL
          Check to see if sector is correct.
          Use ILFAV table at $BFB8 for software sector
          interleaving.
          If wrong sector, try again by branching back to
          TRYADR at $BDC1.
          If sector correct, find out what operation to do.
          If write, branch to WRIT at $BE51.
          Otherwise, read data via READ16 ($B8DC).
          If read is good, then postnibble data via POSTNB16
          ($B8C2) and return to caller with no error.
BE46-BE47 ALLDONE
          Skip over set carry instruction in HNDLERR.
BE48-BE5Ø HNDLERR
          Set carry.
          Store A-reg in IOB as return code.
          Turn off motor.
          Return to caller.
BE51-BE59 WRITE
          Write a sector using WRITE16 ($B82A).
          If the write was good, exit via ALLDONE ($BE46).
          If bad write, load A-reg with $1Ø (write protect
          error) and exit via HNDLERR ($BE48).
BE5A-BE8D MYSEEK
          Provides necessary housekeeping before going to
          SEEKABS routine.
          Determines number of phases per track and stores
          track information in appropriate slot dependent
```

```
          location.
BE8E-BE94 XTOY routine.
          Put slot in Y-reg by transferring X-reg divided
          by 16 into Y-reg.
BE95-BEAE Set track number.
BEAF-BFØC INIT command handler
          Provides setup for initializing a disk.
          Get the desired volume number from the IOB.
          Zero both the primary and secondary buffers.
          Recalibrate the disk arm to track Ø.
          Set the number of sync bytes to be written between
          sectors to $28 (4Ø.).
          Call TRACK WRITE routine for the actual formatting.
          Allow 48 retries during initialization.
          Double check that the first sector found is zero
          after calling TRACK WRITE.
          Increment the track number after successfully
          formatting a track.
          Loop back until 35 tracks are done.
BFØD-BF61 TRACK WRITE routine.
          Start with sector zero.
          Preceed it with 128 self-sync bytes.
          Follow them with sectors Ø through 15 in sequence.
          Set retry count for verifying the track at 48.
          Fill the sector initilization map with positive
          numbers.
          Loop through a delay period to bypass most of the
          initial self-sync bytes.
          Read the first Address Field found.
          If the read is good and sector zero was found,
          enter the VERIFY TRACK routine.
          Decrement the sync count by 2 (until it reaches 16
          at which time it is decremented by 1).
          If sync count is greater than or equal to 5, exit
          via $BF71.
          If not, set carry and return to caller.
BF62-BF87 VERIFY TRACK routine.
          This routine reads all 16 sectors from the track that
          was just formatted.
          If an error occurs during the read of either the
          Address Field or the Data Field, the number of
          retries is decremented.
          The routine continues reading until retries is zero.
          Calls Sector Map routine ($BF88).
BF88-BFA7 Sector Map routine.
          This routine marks the sector initialization map as
          each sector is verified.
          If an error occurs, the routine exits through $BF6C,
          which decrements the number of retries and continues
          if that value is greater than zero.
          Upon completion of track zero, the sync count is
          decremented by two if it is at least 16.
BFA8-BFB7 Sector Initialization Map used to mark sectors as
          they are initialized.
          Contains a $3Ø prior to initialization of a track.
          Value changed to $FF as each sector is completed.
BFB8-BFC7 Sector Translate Table
          Sector interleaving done with software.
BFC8-BFD6 Patch area starts here.
```

```
          Patch from $B741 to zero language card during boot.
          Call SETVID ($FE93).
          Unprotect Language Card (if present).
          Store $ØØ at $EØØØ.
          Exit through SETKBD ($FE89) and DOS coldstart.
BFD9-BFDB Unused.
BFDC-BFE5 Patch called from $AØE2.
          Set three additional defaults (Byte offset=Ø).
          Return to caller.
BFE6-BFEC Patch called from $A6D5.
          Call $A75B to reset state and set warmstart flag.
          Mark RUN not interrupted.
          Return to caller.
BFED-BFFF Patch called from $B377.
          Call $AE7E to save file manager workarea.
          Restore stack.
          Close all open files ($A316).
          Save stack again.
          Exit through $B385 ("DISK FULL ERROR").
```



DOS ZERO PAGE USAGE

BYTE	USE
24	Cursor horizontal (DOS)
26,27	Sector read buffer address (ROM)
	Scratch space (RWTS)
28,29	BASL/BASH (DOS)
2A	Segment merge counter (ROM,BOOT)
	Scratch space (RWTS)
2B	BOOT slot*16 (ROM)
	Scratch space (RWTS)
2C	Checksum from sector header (RWTS)
2D	Sector number from sector header (RWTS)
2E	Track number from sector header (RWTS)
2F	Volume number from sector header (RWTS)
33	Prompt character (DOS)
35	Drive number in high bit (RWTS)
36,37	CSWL,CSWH (DOS)
38,39	KSWL,KSWH (DOS)
3C	Workbyte (ROM)
	Merge workbyte (BOOT)
	Device characteristics table address (RWTS)
3D	Sector number (ROM)
	Device characteristics table address (RWTS)
3E,3F	Address of ROM sector-read subroutine (BOOT)
	Buffer address (RWTS)
4Ø,41	DOS image address (BOOT)
	File buffer address (DOS)
41	Format track counter (RWTS)
42,43	Buffer address (DOS)
44,45	Numeric operand (DOS)
46,47	Scratch space (RWTS)
48,49	IOB address (RWTS)
4A,4B	INTEGER BASIC LOMEM address (DOS)
	Format diskette workspace (RWTS)
4C,4D	INTEGER BASIC HIMEM address (DOS)
67,68	APPLESOFT BASIC PROGRAM START (DOS)
69,6A	APPLESOFT BASIC VARIABLES START (DOS)
6F,7Ø	APPLESOFT BASIC STRING START (DOS)
73,74	APPLESOFT BASIC HIMEM address (DOS)
76	APPLESOFT BASIC line number high (DOS)
AF,BØ	APPLESOFT BASIC PROGRAM END (DOS)
CA,CB	INTEGER BASIC PROGRAM START (DOS)
CC,CD	INTEGER BASIC VARIABLES END (DOS)
D6	APPLESOFT BASIC PROGRAM protection flag (DOS)
D8,D9	INTEGER BASIC line number (DOS)
	APPLESOFT BASIC ONERR (DOS)

APPLEVISION on endless repeat...

Appendix A
Example Programs

This section is intended to supply the reader with utility programs which can be used to examine and repair diskettes. These programs are provided in their source form to serve as examples of the programming necessary to interface practical programs to DOS. The reader who does not know assembly language may also benefit from these programs by entering them from the monitor in their binary form and saving them to disk for later use. It should be pointed out that the use of 16 sector diskettes is assumed, although most of the programs can be easily modified to work under any version of DOS. It is recommended that, until the reader is completely familiar with the operation of these programs, it would be well advised to use them only on an "expendable" diskette. None of the programs can physically damage a diskette, but they can, if used improperly, destroy the data on a diskette, requiring it to be re-`INIT`ialized.

Five programs are provided:

DUMP   TRACK DUMP UTILITY

This is an example of how to directly access the disk drive through its I/O select addresses. `DUMP` may be used to dump any given track in its raw, prenibbilized form, to memory for examination. This can be useful both to understand how disks are formatted and in diagnosing clobbered diskettes.

ZAP   DISK UPDATE UTILITY

This program is the backbone of any attempt to patch a diskette directory back together. It is also useful in examining the structure of files stored on disk and in applying patches to files or DOS directly. `ZAP` allows its user to read, and optionally write, any sector on a diskette. As such, it serves as a good example of a program which calls Read/Write Track/Sector (RWTS).

INIT REFORMAT A SINGLE TRACK

This program will initialize a single track on a diskette. Any volume number (`$ØØ-$FF`) may be specified. `INIT` is useful in restoring a track whose sectoring has been damaged without reinitializing the entire diskette. DOS 3.3 and 48K is assumed.

FTS FIND T/S LISTS UTILITY

`FTS` may be used when the directory for a diskette has been destroyed. It searches every sector on a diskette for what appear to be Track/Sector Lists, printing the track and sector location of each it finds. Knowing the locations of the T/S Lists can help the user patch together a new catalog using `ZAP`.

COPY CONVERT FILES

`COPY` is provided as an example of direct use of the DOS File Manager package from assembly language. The program will read an input `B`-type file and copy its contents to an output `T`-type file. Although it could be used, for example, to convert files used by the Programma PIE editor for use by the Apple Toolkit assembler, it is not included as a utility program but rather as an example of the programming necessary to access the File Manager.

STORING THE PROGRAMS ON DISKETTE

The enterprising programmer may wish to type the source code for each program into an assembler and assemble the programs onto disk. The Apple Toolkit assembler was used to produce the listings presented here, and interested programmers should consult the documentation for that assembler for more information on the pseudo-opcodes used. For the non-assembly language programmer, the binary object code of each program may be entered from the monitor using the following procedure.

The assembly language listings consist of columns of information as follows:

```
The address of some object code
The object code which should be stored there
The statement number
The statement itself
```

For example...

```
Ø8ØØ:2Ø DC Ø3  112 COPY    JSR  LOCFPL      FIND PARMLIST
```

indicates that the binary code "2ØDCØ3" should be stored at 0800 and that this is statement 112. To enter a program in the monitor, the reader must type in each address and its corresponding object code. The following is an example of how to enter the DUMP program:

```
CALL -151       (Enter the monitor from BASIC)
Ø8ØØ:2Ø E3 Ø3
Ø8Ø3:84 ØØ
Ø8Ø5:85 Ø1
Ø8Ø7:A5 Ø2
 ...etc...
Ø879:85 3F
Ø87B:4C B3 FD
BSAVE DUMP,A$8ØØ,L$7E   (Save program to disk)
```

Note that if a line (such as line 4 in DUMP) has no object bytes associated with it, it may be ignored. When the program is to be run...

```
BLOAD DUMP      (Load program)
CALL -151       (Get into monitor)
Ø2:11 N 8ØØG    (Store track to dump, run program)
```

The BSAVE commands which must be used with the other programs are:

```
BSAVE ZAP,A$9ØØ,L$6C
BSAVE INIT,A$8ØØ,L$89
BSAVE FTS,A$9ØØ,L$DC
BSAVE COPY,A$8ØØ,L$1EC
```

DUMP – TRACK DUMP UTILITY

The `DUMP` program will dump any track on a diskette in its raw, pre-nibbilized format, allowing the user to examine the sector address and data fields and the formatting of the track. This allows the curious reader to examine diskettes to better understand the concepts presented in the preceeding chapters. `DUMP` may also be used to examine most protected disks to see how they differ from normal ones and to diagnose diskettes with clobbered sector address or data fields with the intention of recovering from disk I/O errors. The `DUMP` program serves as an example of direct use of the Disk II hardware from assembly language, with little or no use of DOS.

To use `DUMP`, first store the number of the track you wish dumped at location `$Ø2`, then begin execution at `$8ØØ`. `DUMP` will return to the monitor after displaying the first part of the track in hexadecimal on the screen. The entire track image is stored, starting at `$1ØØØ`. For example:

```
CALL -151       (Get into the monitor from BASIC)
BLOAD DUMP      (Load the DUMP program)
```

...Now insert the diskette to be dumped...

```
Ø2:11 N 8ØØG    (Store a 11 (track 17, the catalog
                 track) in $Ø2, N terminates the store
                 command, go to location $8ØØ)
```

The output might look like this...

```
1ØØØ- D5 AA 96 AA AB AA BB AB   (Start of sector address)
1ØØ8- AA AB BA DE AA E8 CØ FF
1Ø1Ø- 9E FF FF FF FF FF D5 AA   (Start of sector data)
1Ø18- AD AE B2 9D AC AE 96 96   (Sector data)
 ...etc...
```

Quite often, a sector with an I/O error will have only one bit which is in error, either in the address or data header or in the actual data itself. A particularly patient programmer can, using `DUMP` and perhaps a half hour of hand “nibbilizing” determine the location of the error and record the data on paper for later entry via `ZAP`. A thorough understanding of Chapter 3 is necessary to accomplish this feat.

```
                1     ************************************
                2     *                                  *
                3     * DUMP:THIS PROGRAM WILL ALLOW ITS *
                4     *      USER TO DUMP AN ENTIRE      *
                5     *      TRACK IN ITS RAW FORM INTO  *
                6     *      MEMORY FOR EXAMINATION.     *
                7     *                                  *
                8     * INPUT: $Ø2 = TRACK TO BE READ    *
                9     *                                  *
                1Ø    * OUTPUT:$1ØØØ = ADDRESS OF TRACK  *
                11    *                IMAGE             *
                12    *                                  *
                13    * ENTRY POINT: $8ØØ                *
                14    *                                  *
                15    * PROGRAMMER: DON D WORTH 2/19/81  *
                16    *                                  *
                17    ************************************

                19    *  ZPAGE DEFINITIONS

                21    PTR      EQU   $Ø          WORK POINTER
                22    TRACK    EQU   $2          TRACK TO BE READ/WRITTEN
                23    A1L      EQU   $3C         MONITOR POINTER
                24    A2L      EQU   $3E         MONITOR POINTER
                25    PREG     EQU   $48         MONITOR STATUS REGISTER

                27    *  OTHER ADDRESSES

                29    BUFFER   EQU   $1ØØØ       TRACK IMAGE AREA
                3Ø    LOCRPL   EQU   $3E3        LOCATE RWTS PARMLIST SUBRTN
                31    RWTS     EQU   $3D9        RWTS SUBROUTINE
                32    COUT     EQU   $FDED       PRINT ONE CHAR SUBROUTINE
                33    XAM      EQU   $FDB3       MONITOR HEX DUMP SUBRTN

                35    *  DISK I/O SELECTS

                37    DRVSMØ   EQU   $CØ8Ø       STEP MOTOR POSITIONS
                38    DRVSM1   EQU   $CØ81
                39    DRVSM2   EQU   $CØ82
                4Ø    DRVSM3   EQU   $CØ83
                41    DRVSM4   EQU   $CØ84
                42    DRVSM5   EQU   $CØ85
                43    DRVSM6   EQU   $CØ86
                44    DRVSM7   EQU   $CØ87
                45    DRVOFF   EQU   $CØ88       TURN DRIVE OFF AFTER 6 REVS
                46    DRVON    EQU   $CØ89       TURN DRIVE ON
                47    DRVSL1   EQU   $CØ8A       SELECT DRIVE 1
                48    DRVSL2   EQU   $CØ8B       SELECT DRIVE 2
                49    DRVRD    EQU   $CØ8C       READ DATA LATCH
                5Ø    DRVWR    EQU   $CØ8D       WRITE DATA LATCH
                51    DRVRDM   EQU   $CØ8E       SET READ MODE
                52    DRVWRM   EQU   $CØ8F       SET WRITE MODE

                54    *  RWTS PARMLIST DEFINITION

                56             ORG   $Ø
ØØØØ: ØØ        57    RPLIOB   DS    1           IOB TYPE ($Ø1)
ØØØ1: ØØ        58    RPLSLT   DS    1           SLOT*16
ØØØ2: ØØ        59    RPLDRV   DS    1           DRIVE
ØØØ3: ØØ        6Ø    RPLVOL   DS    1           VOLUME
ØØØ4: ØØ        61    RPLTRK   DS    1           TRACK
ØØØ5: ØØ        62    RPLSEC   DS    1           SECTOR
ØØØ6: ØØ ØØ     63    RPLDCT   DS    2           ADDRESS OF DCT
ØØØ8: ØØ ØØ     64    RPLBUF   DS    2           ADDRESS OF BUFFER
ØØØA: ØØ ØØ     65    RPLSIZ   DS    2           SECTOR SIZE
```

```
ØØØC: ØØ        66   RPLCMD    DS    1          COMMAND CODE
                67   RPLCNL    EQU   $ØØ           NULL COMMAND
                68   RPLCRD    EQU   $Ø1           READ COMMAND
                69   RPLCWR    EQU   $Ø2           WRITE COMMAND
                7Ø   RPLCFM    EQU   $Ø4           FORMAT COMMAND
ØØØD: ØØ        71   RPLRCD    DS    1          RETURN CODE
                72   RPLRWP    EQU   $1Ø           WRITE PROTECTED
                73   RPLRVM    EQU   $2Ø           VOLUME MISMATCH
                74   RPLRDE    EQU   $4Ø           DRIVE ERROR
                75   RPLRRE    EQU   $8Ø           READ ERROR
ØØØE: ØØ        76   RPLTVL    DS    1          TRUE VOLUME
ØØØF: ØØ        77   RPLPSL    DS    1          PREVIOUS SLOT
ØØ1Ø: ØØ        78   RPLPDR    DS    1          PREVIOUS DRIVE
                79             ORG   $8ØØ

                81   *  USE RWTS TO POSITION THE ARM TO THE DESIRED TRACK

Ø8ØØ: 2Ø E3 Ø3  83   DUMP      JSR   LOCRPL     LOCATE RWTS PARMLIST
Ø8Ø3: 84 ØØ     84             STY   PTR        AND SAVE POINTER
Ø8Ø5: 85 Ø1     85             STA   PTR+1

Ø8Ø7: A5 Ø2     87             LDA   TRACK      GET TRACK TO READ/WRITE
Ø8Ø9: AØ Ø4     88             LDY   #RPLTRK    STORE IN RWTS LIST
Ø8ØB: 91 ØØ     89             STA   (PTR),Y

Ø8ØD: A9 ØØ     91             LDA   #RPLCNL    NULL OPERATION
Ø8ØF: AØ ØC     92             LDY   #RPLCMD    AND STORE IN LIST
Ø811: 91 ØØ     93             STA   (PTR),Y

Ø813: A9 ØØ     95             LDA   #Ø         ANY VOLUME WILL DO
Ø815: AØ Ø3     96             LDY   #RPLVOL
Ø817: 91 ØØ     97             STA   (PTR),Y
Ø819: 2Ø E3 Ø3  98             JSR   LOCRPL     RELOAD POINTER TO PARMS
Ø81C: 2Ø D9 Ø3  99             JSR   RWTS       CALL RWTS
Ø81F: A9 ØØ     1ØØ            LDA   #Ø
Ø821: 85 48     1Ø1            STA   PREG       FIX P REG SO DOS IS HAPPY

                1Ø3  *  PREPARE TO DUMP TRACK TO MEMORY

Ø823: AØ Ø1     1Ø5            LDY   #RPLSLT    GET SLOT*16
Ø825: B1 ØØ     1Ø6            LDA   (PTR),Y
Ø827: AA        1Ø7            TAX
Ø828: BD 89 CØ  1Ø8            LDA   DRVON,X    KEEP DRIVE ON
Ø82B: BD 8E CØ  1Ø9            LDA   DRVRDM,X   INSURE READ MODE

Ø82E: A9 ØØ     111            LDA   #<BUFFER   POINT AT DATA
Ø83Ø: 85 ØØ     112            STA   PTR
Ø832: A9 1Ø     113            LDA   #>BUFFER
Ø834: 85 Ø1     114            STA   PTR+1
Ø836: AØ ØØ     115            LDY   #Ø

                117  *  START DUMPING AT THE BEGINNING OF A SECTOR ADDRESS
                118  *  FIELD OR A SECTOR DATA FIELD

Ø838: BD 8C CØ  12Ø  LOOP1     LDA   DRVRD,X    WAIT FOR NEXT BYTE
Ø83B: 1Ø FB     121            BPL   LOOP1
Ø83D: C9 FF     122            CMP   #$FF       AUTOSYNC?
Ø83F: DØ F7     123            BNE   LOOP1      NO, DON'T START IN MIDDLE
Ø841: BD 8C CØ  124  LOOP2     LDA   DRVRD,X    WAIT FOR NEXT BYTE
Ø844: 1Ø FB     125            BPL   LOOP2
Ø846: C9 FF     126            CMP   #$FF       TWO AUTOSYNCS?
Ø848: DØ EE     127            BNE   LOOP1      NOT YET
Ø84A: BD 8C CØ  128  LOOP3     LDA   DRVRD,X
Ø84D: 1Ø FB     129            BPL   LOOP3
Ø84F: C9 FF     13Ø            CMP   #$FF       STILL AUTOSYNCS?
```

```
Ø851: FØ F7     131          BEQ   LOOP3      YES, WAIT FOR DATA BYTE
Ø853: DØ Ø5     132          BNE   LOOP4      ELSE, START STORING DATA

                134  *  ONCE ALIGNED, BEGIN COPYING THE TRACK TO MEMORY.
                135  *  COPY AT LEAST TWICE ITS LENGTH TO INSURE WE GET IT
                136  *  ALL.

Ø855: BD 8C CØ  138  LOOPD   LDA   DRVRD,X    WAIT FOR NEXT DATA BYTE
Ø858: 1Ø FB     139          BPL   LOOPD
Ø85A: 91 ØØ     14Ø  LOOP4   STA   (PTR),Y    STORE IN MEMORY
Ø85C: E6 ØØ     141          INC   PTR        BUMP POINTER
Ø85E: DØ F5     142          BNE   LOOPD
Ø86Ø: E6 Ø1     143          INC   PTR+1
Ø862: A5 Ø1     144          LDA   PTR+1
Ø864: C9 4Ø     145          CMP   #$4Ø       DONE AT LEAST A TRACK?
Ø866: 9Ø ED     146          BCC   LOOPD      NO, CONTINUE
Ø868: BD 88 CØ  147          LDA   DRVOFF,X   TURN DRIVE OFF

                149  *  WHEN FINISHED, DUMP SOME OF TRACK IN HEX ON SCREEN

Ø86B: A9 ØØ     151  EXIT    LDA   #<BUFFER   DUMP 8ØØ.8AF
Ø86D: 85 3C     152          STA   A1L
Ø86F: A9 1Ø     153          LDA   #>BUFFER
Ø871: 85 3D     154          STA   A1L+1
Ø873: A9 AF     155          LDA   #<BUFFER+$AF
Ø875: 85 3E     156          STA   A2L
Ø877: A9 1Ø     157          LDA   #>BUFFER+$AF
Ø879: 85 3F     158          STA   A2L+1
Ø87B: 4C B3 FD  159          JMP   XAM        EXIT VIA HEX DISPLAY

--End assembly, 143 bytes, Errors: Ø
```

ZAP – DISK UPDATE UTILITY

The next step up the ladder from DUMP is to access data on the diskette at the sector level. The ZAP program allows its user to specify a track and sector to be read into memory. The programmer can then make changes in the image of the sector in memory and subsequently use ZAP to write the modified image back over the sector on disk. ZAP is particularly useful when it is necessary to patch up a damaged directory. Its use in this regard will be covered in more detail when FTS is explained.

To use ZAP, store the number of the track and sector you wish to access in $Ø2 and $Ø3 respectively. Tracks may range from $ØØ to $22 and sectors from $ØØ to $ØF. For example, the Volume Table of Contents (VTOC) for the diskette may be examined by entering $11 for the track and $ØØ for the sector. $Ø4 should be initialized with either a $Ø1 to indicate that the sector is to be read into memory, or $Ø2 to ask that memory be written out to the sector. Other values for location $Ø4 can produce damaging results ($Ø4 in location $Ø4 will INIT your diskette!). When these three memory locations have been set up, begin execution at $9ØØ. ZAP will read or write the sector into or from the 256 bytes starting at $8ØØ. For example:

```
CALL -151                  (Get into the monitor from BASIC)
BLOAD ZAP                  (Load the ZAP program)
```

...Now insert the diskette to be zapped...

```
Ø2:11 ØØ Ø1 N 9ØØG  (Store a 11 (track 17, the catalog
                     track) in $Ø2, a ØØ (sector Ø) at $Ø3,
                     and a Ø1 (read) at $Ø4.  N ends the
                     store command and 9ØØG runs ZAP.)
```

The output might look like this...

```
Ø8ØØ- Ø4 11 ØF Ø3 ØØ ØØ Ø1 ØØ   (Start of VTOC)
Ø8Ø8- ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø81Ø- ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø818- ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
 ...etc...
```

In the above example, if the byte at offset 3 (the version of DOS which INITed this diskette) is to be changed, the following would be entered...

```
8Ø3:Ø2          (Change Ø3 to Ø2)
Ø4:Ø2 N 9ØØG    (Change ZAP to write mode and do it)
```

Note that ZAP will remember the previous values in $Ø2, $Ø3, and $Ø4.

If something is wrong with the sector to be read (an I/O error, perhaps), ZAP will print an error message of the form:

RC=1Ø

A return code of 1Ø, in this case, means that the diskette was WRITE PROTECTED and a write operation was attempted. Other error codes are 2Ø (VOLUME MISMATCH), 4Ø (DRIVE ERROR), and 8Ø (READ ERROR). Refer to the documentation on RWTS given in Chapter 6 for more information on these errors.

```
                 1    ************************************
                 2    *                                  *
                 3    * ZAP: THIS PROGRAM WILL ALLOW ITS *
                 4    *      USER TO READ/WRITE          *
                 5    *      INDIVIDUAL SECTORS FROM/TO  *
                 6    *      THE DISKETTE                *
                 7    *                                  *
                 8    * INPUT: $Ø2  = TRACK TO BE READ   *
                 9    *        $Ø3  = SECTOR TO BE READ  *
                 1Ø   *               /WRITTEN           *
                 11   *        $Ø4  = $Ø1 - READ SECTOR  *
                 12   *               $Ø2 - WRITE SECTOR *
                 13   *        $8ØØ = ADDRESS OF SECTOR  *
                 14   *               DATA BUFFER        *
                 15   *                                  *
                 16   * ENTRY POINT: $9ØØ                *
                 17   *                                  *
                 18   * PROGRAMMER: DON D WORTH 2/15/81  *
                 19   *                                  *
                 2Ø   ************************************

                 22   BELL     EQU   $87        BELL CHARACTER

                 24   *        ZPAGE DEFINITIONS

                 26   PTR      EQU   $Ø         WORK POINTER
                 27   TRACK    EQU   $2         TRACK TO BE READ/WRITTEN
                 28   SECTOR   EQU   $3         SECTOR TO BE READ/WRITTEN
                 29   OPER     EQU   $4         OPERATION TO BE PERFORMED
                 3Ø   READ     EQU   1             READ OPERATION
                 31   WRITE    EQU   2             WRITE OPERATION
                 32   A1L      EQU   $3C        MONITOR POINTER
                 33   A2L      EQU   $3E        MONITOR POINTER
                 34   PREG     EQU   $48        MONITOR STATUS REGISTER

                 36   *        OTHER ADDRESSES

                 38   BUFFER   EQU   $8ØØ       SECTOR DATA BUFFER
                 39   LOCRPL   EQU   $3E3       LOCATE RWTS PARMLIST SUBRTN
                 4Ø   RWTS     EQU   $3D9       RWTS SUBROUTINE
                 41   COUT     EQU   $FDED      PRINT ONE CHAR SUBROUTINE
                 42   PRBYTE   EQU   $FDDA      PRINT ONE HEX BYTE SUBRTN
                 43   XAM      EQU   $FDB3      MONITOR HEX DUMP SUBRTN

                 45   *        RWTS PARMLIST DEFINITION

                 47            ORG   $Ø
ØØØØ: ØØ         48   RPLIOB   DS    1          IOB TYPE ($Ø1)
ØØØ1: ØØ         49   RPLSLT   DS    1          SLOT*16
ØØØ2: ØØ         5Ø   RPLDRV   DS    1          DRIVE
```

```
ØØØ3: ØØ        51   RPLVOL   DS    1           VOLUME
ØØØ4: ØØ        52   RPLTRK   DS    1           TRACK
ØØØ5: ØØ        53   RPLSEC   DS    1           SECTOR
ØØØ6: ØØ ØØ     54   RPLDCT   DS    2           ADDRESS OF DCT
ØØØ8: ØØ ØØ     55   RPLBUF   DS    2           ADDRESS OF BUFFER
ØØØA: ØØ ØØ     56   RPLSIZ   DS    2           SECTOR SIZE
ØØØC: ØØ        57   RPLCMD   DS    1           COMMAND CODE
                58   RPLCNL   EQU   $ØØ            NULL COMMAND
                59   RPLCRD   EQU   $Ø1            READ COMMAND
                6Ø   RPLCWR   EQU   $Ø2            WRITE COMMAND
                61   RPLCFM   EQU   $Ø4            FORMAT COMMAND
ØØØD: ØØ        62   RPLRCD   DS    1           RETURN CODE
                63   RPLRWP   EQU   $1Ø            WRITE PROTECTED
                64   RPLRVM   EQU   $2Ø            VOLUME MISMATCH
                65   RPLRDE   EQU   $4Ø            DRIVE ERROR
                66   RPLRRE   EQU   $8Ø            READ ERROR
ØØØE: ØØ        67   RPLTVL   DS    1           TRUE VOLUME
ØØØF: ØØ        68   RPLPSL   DS    1           PREVIOUS SLOT
ØØ1Ø: ØØ        69   RPLPDR   DS    1           PREVIOUS DRIVE
                7Ø            ORG   $9ØØ

                72   *       FILL IN RWTS LIST

Ø9ØØ: 2Ø E3 Ø3  74   ZAP      JSR   LOCRPL      LOCATE RWTS PARMLIST
Ø9Ø3: 84 ØØ     75            STY   PTR         AND SAVE POINTER
Ø9Ø5: 85 Ø1     76            STA   PTR+1

Ø9Ø7: A5 Ø2     78            LDA   TRACK       GET TRACK TO READ/WRITE
Ø9Ø9: AØ Ø4     79            LDY   #RPLTRK     STORE IN RWTS LIST
Ø9ØB: 91 ØØ     8Ø            STA   (PTR),Y

Ø9ØD: A5 Ø3     82            LDA   SECTOR      GET SECTOR TO READ/WRITE
Ø9ØF: C9 1Ø     83            CMP   #16         BIGGER THAN 16 SECTORS?
Ø911: 9Ø Ø4     84            BCC   SOK         NO
Ø913: A9 ØØ     85            LDA   #Ø
Ø915: 85 Ø3     86            STA   SECTOR      YES, PUT IT BACK TO ZERO
Ø917: AØ Ø5     87   SOK      LDY   #RPLSEC
Ø919: 91 ØØ     88            STA   (PTR),Y     STORE IN RWTS LIST

Ø91B: AØ Ø8     9Ø            LDY   #RPLBUF
Ø91D: A9 ØØ     91            LDA   #<BUFFER    STORE BUFFER PTR IN LIST
Ø91F: 91 ØØ     92            STA   (PTR),Y
Ø921: C8        93            INY
Ø922: A9 Ø8     94            LDA   #>BUFFER
Ø924: 91 ØØ     95            STA   (PTR),Y

Ø926: A5 Ø4     97            LDA   OPER        GET COMMAND CODE
Ø928: AØ ØC     98            LDY   #RPLCMD     AND STORE IN LIST
Ø92A: 91 ØØ     99            STA   (PTR),Y

Ø92C: A9 ØØ     1Ø1           LDA   #Ø          ANY VOLUME WILL DO
Ø92E: AØ Ø3     1Ø2           LDY   #RPLVOL
Ø93Ø: 91 ØØ     1Ø3           STA   (PTR),Y

                1Ø5  *       NOW CALL RWTS TO READ/WRITE THE SECTOR

Ø932: 2Ø E3 Ø3  1Ø7           JSR   LOCRPL      RELOAD POINTER TO PARMS
Ø935: 2Ø D9 Ø3  1Ø8           JSR   RWTS        CALL RWTS
Ø938: A9 ØØ     1Ø9           LDA   #Ø
Ø93A: 85 48     11Ø           STA   PREG        FIX P REG SO DOS IS HAPPY
Ø93C: 9Ø 1B     111           BCC   EXIT        ALL IS WELL
                113  *       ERROR OCCURED, PRINT "RC=XX"

Ø93E: A9 87     115           LDA   #BELL       BEEP THE SPEAKER
Ø94Ø: 2Ø ED FD  116           JSR   COUT
```

```
Ø943: A9 52      117          LDA   #'R'        PRINT THE "RC="
Ø945: 2Ø ED FD   118          JSR   COUT
Ø948: A9 43      119          LDA   #'C'
Ø94A: 2Ø ED FD   12Ø          JSR   COUT
Ø94D: A9 3D      121          LDA   #'='
Ø94F: 2Ø ED FD   122          JSR   COUT
Ø952: AØ ØD      123          LDY   #RPLRCD
Ø954: B1 ØØ      124          LDA   (PTR),Y     GET RWTS RETURN CODE
Ø956: 2Ø DA FD   125          JSR   PRBYTE      PRINT RETURN CODE IN HEX

                 127  *       WHEN FINISHED, DUMP SOME OF SECTOR IN HEX

Ø959: A9 ØØ      129  EXIT    LDA   #<BUFFER    DUMP 8ØØ.8B7
Ø95B: 85 3C      13Ø          STA   A1L
Ø95D: A9 Ø8      131          LDA   #>BUFFER
Ø95F: 85 3D      132          STA   A1L+1
Ø961: A9 AF      133          LDA   #<BUFFER+$AF
Ø963: 85 3E      134          STA   A2L
Ø965: A9 Ø8      135          LDA   #>BUFFER+$AF
Ø967: 85 3F      136          STA   A2L+1
Ø969: 4C B3 FD   137          JMP   XAM         EXIT VIA HEX DISPLAY

--End assembly, 125 bytes, Errors: Ø
```

INIT – REFORMAT A SINGLE TRACK

Occasionally the sectoring information on a diskette can become damaged so that one or more sectors can no longer be found by DOS. To correct this problem requires that the sector address and data fields be re-formatted for the entire track thus affected. `INIT` can be used to selectively reformat a single track, thus avoiding a total re-`INIT` of the diskette. Before using `INIT`, the user should first attempt to write on the suspect sector (using `ZAP`). If RWTS refuses to write to the sector (`RC=4Ø`), then `INIT` must be run on the entire track. To avoid losing data, all other sectors on the track should be read and copied to another diskette prior to reformatting. After `INIT` is run they can be copied back to the repaired diskette and data can be written to the previously damaged sector.

To run `INIT`, first store the number of the track you wish reformatted at location `$Ø2`, the volume number of the disk at location `$Ø3`, and then begin execution at `$8ØØ`. `INIT` will return to the monitor upon completion. If the track can not be formatted for some reason (eg. physical damage or problems with the disk drive itself) a return code is printed. For example:

```
CALL -151      (Get into the monitor from BASIC)
BLOAD INIT     (Load the INIT program)
```

...Now insert the disk to be `INIT`-ed...

```
Ø2:11 FE N 8ØØG    (Store a 11 (track 17, the catalog
                    track) in $Ø2, a volume number of
                    $FE (254) in $Ø3, N terminates the
                    store command, go to location $8ØØ)
```

WARNING: DOS 3.3 must be loaded in the machine before running `INIT` and a 48K Apple is assumed. `INIT` will not work with other versions of DOS or other memory sizes.

```
1    *************************************
2    *                                   *
3    * INIT: THIS PROGRAM WILL ALLOW ITS *
4    *       USER TO INITIALIZE A        *
5    *       SINGLE TRACK WITH ANY VOLUME *
6    *       NUMBER DESIRED.             *
7    *                                   *
8    * INPUT: $Ø2 = TRACK TO BE INITIALIZED *
9    *        $Ø3 = VOLUME NUMBER        *
1Ø   *                                   *
11   * ENTRY POINT: $8ØØ                 *
12   *                                   *
13   * PROGRAMMER: PIETER LECHNER 2/19/81 *
14   *                                   *
15   *************************************

17   *       ZPAGE DEFINITIONS
```

```
                19   PTR      EQU   $Ø         WORK POINTER
                2Ø   TRACK    EQU   $2         TRACK TO BE READ/WRITTEN
                21   VOLUME   EQU   $3         VOLUME NUMBER
                22   SECFND   EQU   $2D        SECTOR FOUND BY RDADR16
                23   AA       EQU   $3E        ZPAGE CONSTANT FOR TIMING
                24   VOL      EQU   $41        VOLUME USED BY WRADR16
                25   TRK      EQU   $44        TRACK USED BY WRADR16
                26   SYNCNT   EQU   $45        SYNC COUNT USED BY DSKF2
                27   PREG     EQU   $48        MONITOR P REGISTER SAVEAREA
                28   BELL     EQU   $87        ASCII BELL

                3Ø   *       OTHER ADDRESSES

                32   LOCRPL   EQU   $3E3       LOCATE RWTS PARMLIST SUBRTN
                33   RWTS     EQU   $3D9       RWTS SUBROUTINE
                34   RTRYCNT  EQU   $578       RETRY COUNT FOR DSKF2
                35   NBUF1    EQU   $BBØØ      PRIMARY SECTOR BUFFER
                36   NBUF2    EQU   $BCØØ      SECONDARY SECTOR BUFFER
                37   READ16   EQU   $B8DC      READ DATA FIELD ROUTINE
                38   RDADR16  EQU   $B944      READ ADDRESS FIELD ROUTINE
                39   DSKF2    EQU   $BFØD      FORMAT ONE TRACK ROUTINE
                4Ø   COUT     EQU   $FDED      MONITOR CHARACTER OUTPUT
                41   PRBYTE   EQU   $FDDA      MONITOR HEX OUTPUT

                43   *       DISK I/O SELECTS

                45   DRVSMØ   EQU   $CØ8Ø      STEP MOTOR POSITIONS
                46   DRVSM1   EQU   $CØ81
                47   DRVSM2   EQU   $CØ82
                48   DRVSM3   EQU   $CØ83
                49   DRVSM4   EQU   $CØ84
                5Ø   DRVSM5   EQU   $CØ85
                51   DRVSM6   EQU   $CØ86
                52   DRVSM7   EQU   $CØ87
                53   DRVOFF   EQU   $CØ88      TURN DRIVE OFF AFTER 6 REVS
                54   DRVON    EQU   $CØ89      TURN DRIVE ON
                55   DRVSL1   EQU   $CØ8A      SELECT DRIVE 1
                56   DRVSL2   EQU   $CØ8B      SELECT DRIVE 2
                57   DRVRD    EQU   $CØ8C      READ DATA LATCH
                58   DRVWR    EQU   $CØ8D      WRITE DATA LATCH
                59   DRVRDM   EQU   $CØ8E      SET READ MODE
                6Ø   DRVWRM   EQU   $CØ8F      SET WRITE MODE

                62   *       RWTS PARMLIST DEFINITION

                64            ORG   $Ø
ØØØØ: ØØ        65   RPLIOB   DS    1          IOB TYPE ($Ø1)
ØØØ1: ØØ        66   RPLSLT   DS    1          SLOT*16
ØØØ2: ØØ        67   RPLDRV   DS    1          DRIVE
ØØØ3: ØØ        68   RPLVOL   DS    1          VOLUME
ØØØ4: ØØ        69   RPLTRK   DS    1          TRACK
ØØØ5: ØØ        7Ø   RPLSEC   DS    1          SECTOR
ØØØ6: ØØ ØØ     71   RPLDCT   DS    2          ADDRESS OF DCT
ØØØ8: ØØ ØØ     72   RPLBUF   DS    2          ADDRESS OF BUFFER
ØØØA: ØØ ØØ     73   RPLSIZ   DS    2          SECTOR SIZE
ØØØC: ØØ        74   RPLCMD   DS    1          COMMAND CODE
                75   RPLCNL   EQU   $ØØ           NULL COMMAND
                76   RPLCRD   EQU   $Ø1           READ COMMAND
                77   RPLCWR   EQU   $Ø2           WRITE COMMAND
                78   RPLCFM   EQU   $Ø4           FORMAT COMMAND
ØØØD: ØØ        79   RPLRCD   DS    1          RETURN CODE
                8Ø   RPLRWP   EQU   $1Ø           WRITE PROTECTED
                81   RPLRVM   EQU   $2Ø           VOLUME MISMATCH
                82   RPLRDE   EQU   $4Ø           DRIVE ERROR
```

```
                83   RPLRRE   EQU   $8Ø            READ ERROR
ØØØE: ØØ        84   RPLTVL   DS    1           TRUE VOLUME
ØØØF: ØØ        85   RPLPSL   DS    1           PREVIOUS SLOT
ØØ1Ø: ØØ        86   RPLPDR   DS    1           PREVIOUS DRIVE
                87            ORG   $8ØØ

                89   *       USE RWTS TO POSITION THE ARM TO THE DESIRED TRACK

Ø8ØØ: 2Ø E3 Ø3  91   DUMP     JSR   LOCRPL      LOCATE RWTS PARMLIST
Ø8Ø3: 84 ØØ     92            STY   PTR         AND SAVE POINTER
Ø8Ø5: 85 Ø1     93            STA   PTR+1

Ø8Ø7: A5 Ø2     95            LDA   TRACK       GET TRACK TO READ/WRITE
Ø8Ø9: AØ Ø4     96            LDY   #RPLTRK     STORE IN RWTS LIST
Ø8ØB: 91 ØØ     97            STA   (PTR),Y

Ø8ØD: A9 ØØ     99            LDA   #RPLCNL     NULL OPERATION
Ø8ØF: AØ ØC     1ØØ           LDY   #RPLCMD     AND STORE IN LIST
Ø811: 91 ØØ     1Ø1           STA   (PTR),Y

Ø813: A9 ØØ     1Ø3           LDA   #Ø          ANY VOLUME WILL DO
Ø815: AØ Ø3     1Ø4           LDY   #RPLVOL
Ø817: 91 ØØ     1Ø5           STA   (PTR),Y
Ø819: 2Ø E3 Ø3  1Ø6           JSR   LOCRPL      RELOAD POINTER TO PARMS
Ø81C: 2Ø D9 Ø3  1Ø7           JSR   RWTS        CALL RWTS
Ø81F: BD 89 CØ  1Ø8           LDA   DRVON,X     LEAVE DRIVE ON

                11Ø  *       ESTABLISH ENVIRONMENT FOR DSKF2 ROUTINE

Ø822: A5 Ø2     112           LDA   TRACK       PASS TRACK TO DSKF2
Ø824: 85 44     113           STA   TRK
Ø826: A5 Ø3     114           LDA   VOLUME      AND VOLUME
Ø828: 85 41     115           STA   VOL
Ø82A: A9 AA     116           LDA   #$AA        STORE CONSTANT FOR ZPAGE..
Ø82C: 85 3E     117           STA   AA          TIMING
Ø82E: A9 28     118           LDA   #$28        START WITH 4Ø SYNCS..
Ø83Ø: 85 45     119           STA   SYNCNT      BETWEEN SECTORS
Ø832: AØ 56     12Ø           LDY   #$56
Ø834: A9 ØØ     121           LDA   #$ØØ
Ø836: 99 FF BB  122  ZNBUF2   STA   NBUF2-1,Y   ZERO SECONDARY BUFFER
Ø839: 88        123           DEY
Ø83A: DØ FA     124           BNE   ZNBUF2
Ø83C: 99 ØØ BB  125  ZNBUF1   STA   NBUF1,Y     AND PRIMARY BUFFER
Ø83F: 88        126           DEY
Ø84Ø: DØ FA     127           BNE   ZNBUF1

                129  *       INITIALIZE TRACK

Ø842: 2Ø ØD BF  131           JSR   DSKF2       FORMAT TRACK AND VERIFY
Ø845: A9 Ø8     132           LDA   #$Ø8        IN CASE OF ERROR...
Ø847: BØ 19     133           BCS   HNDERR      ERROR?

                135  *       READ SECTOR ZERO TO VERIFY FORMATTING

Ø849: A9 3Ø     137           LDA   #$3Ø        NO, DOUBLE CHECK TRACK
Ø84B: 8D 78 Ø5  138           STA   RTRYCNT     ALLOW 48 RETRIES
Ø84E: 38        139  NOGOOD   SEC
Ø84F: CE 78 Ø5  14Ø           DEC   RTRYCNT     COUNT RETRIES
Ø852: FØ ØE     141           BEQ   HNDERR
Ø854: 2Ø 44 B9  142           JSR   RDADR16     READ AN ADDRESS FIELD
Ø857: BØ F5     143           BCS   NOGOOD      ERROR, TRY AGAIN
Ø859: A5 2D     144           LDA   SECFND      IS THIS SECTOR ZERO?
Ø85B: DØ F1     145           BNE   NOGOOD      NO, TRY AGAIN
Ø85D: 2Ø DC B8  146           JSR   READ16      YES, READ DATA FIELD
Ø86Ø: 9Ø 1F     147           BCC   DONETRK     ALL IS WELL, DONE.
```

```
Ø862: AØ ØD     148  HNDERR   LDY   #RPLRCD    ELSE, PHONEY UP A RC
Ø864: 91 ØØ     149           STA   (PTR),Y

                151  *        ERROR OCCURED, PRINT "RC=XX"

Ø866: A9 87     153           LDA   #BELL      BEEP THE SPEAKER
Ø868: 2Ø ED FD  154           JSR   COUT
Ø86B: A9 52     155           LDA   #'R'       PRINT THE "RC="
Ø86D: 2Ø ED FD  156           JSR   COUT
Ø87Ø: A9 43     157           LDA   #'C'
Ø872: 2Ø ED FD  158           JSR   COUT
Ø875: A9 3D     159           LDA   #'='
Ø877: 2Ø ED FD  16Ø           JSR   COUT
Ø87A: AØ ØD     161           LDY   #RPLRCD
Ø87C: B1 ØØ     162           LDA   (PTR),Y    GET RWTS RETURN CODE
Ø87E: 2Ø DA FD  163           JSR   PRBYTE     PRINT RETURN CODE IN HEX

                165  *        WHEN DONE, EXIT TO CALLER

Ø881: BD 88 CØ  167  DONETRK  LDA   DRVOFF,X   TURN DRIVE OFF
Ø884: A9 ØØ     168           LDA   #$ØØ
Ø886: 85 48     169           STA   PREG       CLEAR P REGISTER FOR DOS
Ø888: 6Ø        17Ø           RTS              ; RETURN TO CALLER

--End assembly, 154 bytes, Errors: Ø
```

FTS – FIND T/S LISTS UTILITY

From time to time one of your diskettes will develop an I/O error smack in the middle of the catalog track. When this occurs, any attempt to use the diskette will result in an `I/O ERROR` message from DOS. Generally, when this happens, the data stored in the files on the diskette is still intact; only the pointers to the files are gone. If the data absolutely must be recovered, a knowledgeable Apple user can reconstruct the catalog from scratch. Doing this involves first finding the T/S Lists for each file, and then using `ZAP` to patch a catalog entry into track 16 for each file which was found. `FTS` is a utility which will scan a diskette for T/S Lists. Although it may flag some sectors which are not T/S Lists as being such, it will never miss a valid T/S List. Therefore, after running `FTS` the programmer must use `ZAP` to examine each track/sector printed by `FTS` to see if it is really a T/S List. Additionally, `FTS` will find every T/S List image on the diskette, even some which were for files which have since been deleted. Since it is difficult to determine which files are valid and which are old deleted files, it is usually necessary to restore all the files and copy them to another diskette, and later delete the duplicate or unwanted ones.

To run `FTS`, simply load the program and start execution at `$9ØØ`. `FTS` will print the track and sector number of each sector it finds which bears a resemblance to a T/S List. For example:

```
CALL -151      (Get into the monitor from BASIC)
BLOAD FTS      (Load the FTS program)
```

...Now insert the disk to be scanned...

```
9ØØG           (Run the FTS program on this diskette)
```

The output might look like this...

```
T=12 S=ØF
T=13 S=ØF
T=14 S=ØD
T=14 S=ØF
```

Here, only four possible files were found. `ZAP` should now be used to read track `$12`, sector `$ØF`. At `+$ØC` is the track and sector of the first sector in the file. This sector can be read and examined to try to identify the file and its type. Usually a BASIC program can be identified, even though it is stored in tokenized form, from the text strings contained in the `PRINT` statements. An ASCII conversion chart (see page 8 in the *Apple II Reference Manual*) can be used to decode these character strings. Straight `T`-type files will also contain ASCII

text, with each line separated from the others with `$8D` (carriage returns). B-type files are the hardest to identify, unless the address and length stored in the first 4 bytes are recognizable. If you cannot identify the file, assume it is Applesoft BASIC. If this assumption turns out to be incorrect, you can always go back and `ZAP` the file type in the `CATALOG` to try something else. Given below is an example `ZAP` to the `CATALOG` to create an entry for the file whose T/S List is at `T=12 S=ØF`.

```
CALL -151
BLOAD ZAP
```

...insert disk to be ZAPped...

```
8ØØ:ØØ N 8Ø1<8ØØ.8FEM   (Zero sector area of memory)
8ØB:12 ØF Ø2            (Track 12, Sector ØF, Type-A)
:C1 AØ AØ AØ AØ AØ AØ   (Name is "A")
:AØ AØ AØ AØ AØ AØ AØ   (fill name out with 29 blanks)
:AØ AØ AØ AØ AØ AØ AØ
:AØ AØ AØ AØ AØ AØ AØ
:AØ AØ
Ø2:11 ØF Ø2 N 9ØØG      (Write new sector image out as
                         first (and only) catalog sector)
```

The file should immediately be copied to another diskette and then the process repeated for each T/S List found by `FTS` until all of the files have been recovered. As each file is recovered, it may be `RENAME`d to its previous name. Once all the files have been copied to another disk, and successfully tested, the damaged disk may be re-`INIT`ialized.

```
1    **********************************
2    *                                *
3    * FTS: THIS PROGRAM SCANS THE    *
4    *      ENTIRE DISKETTE FOR WHAT  *
5    *      APPEAR TO BE TRACK/SECTOR *
6    *      LISTS AND PRINTS THE      *
7    *      TRACK AND SECTOR OF EACH  *
8    *      ONE IT FINDS.             *
9    *                                *
1Ø   * INPUT: NONE                    *
11   *                                *
12   * ENTRY POINT: $9ØØ              *
13   *                                *
14   * PROGRAMMER: DON D WORTH 2/15/81 *
15   *                                *
16   **********************************

18   BELL     EQU   $87        BELL CHARACTER
19   RETURN   EQU   $8D        CARRIAGE RETURN

21   *        ZPAGE DEFINITIONS

23   PTR      EQU   $Ø         WORK POINTER
24   A1L      EQU   $3C        MONITOR POINTER
```

```
                25   A2L      EQU   $3E        MONITOR POINTER
                26   PREG     EQU   $48        MONITOR STATUS REGISTER

                28   *      OTHER ADDRESSES

                3Ø   BUFFER   EQU   $8ØØ       SECTOR DATA BUFFER
                31   LOCRPL   EQU   $3E3       LOCATE RWTS PARMLIST SUBRTN
                32   RWTS     EQU   $3D9       RWTS SUBROUTINE
                33   COUT     EQU   $FDED      PRINT ONE CHAR SUBROUTINE
                34   PRBYTE   EQU   $FDDA      PRINT ONE HEX BYTE SUBRTN

                36   *      RWTS PARMLIST DEFINITION

                38            ORG   $Ø
ØØØØ: ØØ        39   RPLIOB   DS    1          IOB TYPE ($Ø1)
ØØØ1: ØØ        4Ø   RPLSLT   DS    1          SLOT*16
ØØØ2: ØØ        41   RPLDRV   DS    1          DRIVE
ØØØ3: ØØ        42   RPLVOL   DS    1          VOLUME
ØØØ4: ØØ        43   RPLTRK   DS    1          TRACK
ØØØ5: ØØ        44   RPLSEC   DS    1          SECTOR
ØØØ6: ØØ ØØ     45   RPLDCT   DS    2          ADDRESS OF DCT
ØØØ8: ØØ ØØ     46   RPLBUF   DS    2          ADDRESS OF BUFFER
ØØØA: ØØ ØØ     47   RPLSIZ   DS    2          SECTOR SIZE
ØØØC: ØØ        48   RPLCMD   DS    1          COMMAND CODE
                49   RPLCNL   EQU   $ØØ           NULL COMMAND
                5Ø   RPLCRD   EQU   $Ø1           READ COMMAND
                51   RPLCWR   EQU   $Ø2           WRITE COMMAND
                52   RPLCFM   EQU   $Ø4           FORMAT COMMAND
ØØØD: ØØ        53   RPLRCD   DS    1          RETURN CODE
                54   RPLRWP   EQU   $1Ø           WRITE PROTECTED
                55   RPLRVM   EQU   $2Ø           VOLUME MISMATCH
                56   RPLRDE   EQU   $4Ø           DRIVE ERROR
                57   RPLRRE   EQU   $8Ø           READ ERROR
ØØØE: ØØ        58   RPLTVL   DS    1          TRUE VOLUME
ØØØF: ØØ        59   RPLPSL   DS    1          PREVIOUS SLOT
ØØ1Ø: ØØ        6Ø   RPLPDR   DS    1          PREVIOUS DRIVE
                61            ORG   $9ØØ

                63   *      START TRACK/SECTOR JUST PAST DOS (TRACK 3)

Ø9ØØ: 2Ø E3 Ø3  65   FTS      JSR   LOCRPL     LOCATE RWTS PARMLIST
Ø9Ø3: 84 ØØ     66            STY   PTR        AND SAVE POINTER
Ø9Ø5: 85 Ø1     67            STA   PTR+1

Ø9Ø7: A9 Ø3     69            LDA   #3         FIRST NON-DOS TRACK
Ø9Ø9: AØ Ø4     7Ø            LDY   #RPLTRK    STORE IN RWTS LIST
Ø9ØB: 91 ØØ     71            STA   (PTR),Y

Ø9ØD: AØ Ø8     73            LDY   #RPLBUF
Ø9ØF: A9 ØØ     74            LDA   #<BUFFER   STORE BUFFER PTR IN LIST
Ø911: 91 ØØ     75            STA   (PTR),Y
Ø913: C8        76            INY
Ø914: A9 Ø8     77            LDA   #>BUFFER
Ø916: 91 ØØ     78            STA   (PTR),Y

Ø918: A9 Ø1     8Ø            LDA   #RPLCRD    GET COMMAND CODE FOR READ
Ø91A: AØ ØC     81            LDY   #RPLCMD    AND STORE IN LIST
Ø91C: 91 ØØ     82            STA   (PTR),Y

Ø91E: A9 ØØ     84            LDA   #Ø         ANY VOLUME WILL DO
Ø92Ø: AØ Ø3     85            LDY   #RPLVOL
Ø922: 91 ØØ     86            STA   (PTR),Y

                88   *      NEW TRACK, START SECTOR AT ZERO
```

```
Ø924: AØ Ø5     9Ø   NEWTRK   LDY   #RPLSEC
Ø926: A9 ØØ     91            LDA   #Ø
Ø928: 91 ØØ     92            STA   (PTR),Y

                94   *       NOW CALL RWTS TO READ THE SECTOR

Ø92A: 2Ø E3 Ø3  96   NEWSEC   JSR   LOCRPL     RELOAD POINTER TO PARMS
Ø92D: 2Ø D9 Ø3  97            JSR   RWTS       CALL RWTS
Ø93Ø: A9 ØØ     98            LDA   #Ø
Ø932: 85 48     99            STA   PREG       FIX P REG SO DOS IS HAPPY
Ø934: 9Ø 26     1ØØ           BCC   SCAN       ALL IS WELL

                1Ø2  *       ERROR OCCURED, PRINT "RC=XX"

Ø936: 2Ø B3 Ø9  1Ø4           JSR   PRTTS      PRINT TRACK/SECTOR
Ø939: A9 87     1Ø5           LDA   #BELL      BEEP THE SPEAKER
Ø93B: 2Ø ED FD  1Ø6           JSR   COUT
Ø93E: A9 52     1Ø7           LDA   #'R'       PRINT THE "RC="
Ø94Ø: 2Ø ED FD  1Ø8           JSR   COUT
Ø943: A9 43     1Ø9           LDA   #'C'
Ø945: 2Ø ED FD  11Ø           JSR   COUT
Ø948: A9 3D     111           LDA   #'='
Ø94A: 2Ø ED FD  112           JSR   COUT
Ø94D: AØ ØD     113           LDY   #RPLRCD
Ø94F: B1 ØØ     114           LDA   (PTR),Y    GET RWTS RETURN CODE
Ø951: 2Ø DA FD  115           JSR   PRBYTE     PRINT RETURN CODE IN HEX
Ø954: A9 8D     116           LDA   #RETURN
Ø956: 2Ø ED FD  117           JSR   COUT
Ø959: 4C 8E Ø9  118           JMP   NXTSEC     GO ON

                12Ø  *       NO ERROR, SEE IF SECTOR LOOKS LIKE A T/S LIST

Ø95C: A2 ØØ     122  SCAN     LDX   #Ø
Ø95E: BD ØØ Ø8  123  SCLPØ    LDA   BUFFER,X   MAKE SURE ITS NOT ALL ZERO
Ø961: DØ Ø5     124           BNE   SCAN1
Ø963: E8        125           INX
Ø964: DØ F8     126           BNE   SCLPØ
Ø966: FØ 26     127           BEQ   NXTSEC     IF IT IS, SKIP IT

Ø968: A2 Ø5     129  SCAN1    LDX   #5         START AT OFFSET 5
Ø96A: BD ØØ Ø8  13Ø  SCLP1    LDA   BUFFER,X
Ø96D: DØ 1F     131           BNE   NXTSEC     HEADER OF T/S MUST BE ZERO
Ø96F: E8        132           INX
Ø97Ø: EØ ØC     133           CPX   #12        AT THE T/S PAIRS YET?
Ø972: 9Ø F6     134           BCC   SCLP1      NO, KEEP CHECKING

Ø974: BD ØØ Ø8  136  SCLP2    LDA   BUFFER,X   GET TRK
Ø977: C9 23     137           CMP   #35        MUST BE Ø-34
Ø979: BØ 13     138           BCS   NXTSEC
Ø97B: E8        139           INX
Ø97C: BD ØØ Ø8  14Ø           LDA   BUFFER,X   GET SECTOR
Ø97F: C9 1Ø     141           CMP   #16        MUST BE Ø-15
Ø981: BØ ØB     142           BCS   NXTSEC
Ø983: E8        143           INX
Ø984: DØ EE     144           BNE   SCLP2

Ø986: 2Ø B3 Ø9  146           JSR   PRTTS      ALL CONDITIONS MET
Ø989: A9 8D     147           LDA   #RETURN
Ø98B: 2Ø ED FD  148           JSR   COUT

                15Ø  *       BUMP SECTOR NUMBER OR TRACK AND CONTINUE

Ø98E: AØ Ø5     152  NXTSEC   LDY   #RPLSEC
Ø99Ø: B1 ØØ     153           LDA   (PTR),Y    GET LAST SECTOR
Ø992: 18        154           CLC
```

```
Ø993: 69 Ø1      155          ADC   #1        BUMP BY ONE
Ø995: 91 ØØ      156          STA   (PTR),Y   AND PUT IT BACK IN LIST
Ø997: C9 1Ø      157          CMP   #16       TOO BIG?
Ø999: BØ Ø3      158          BCS   NXTTRK
Ø99B: 4C 2A Ø9   159          JMP   NEWSEC    NO, GO READ IT

Ø99E: AØ Ø4      161  NXTTRK  LDY   #RPLTRK
Ø9AØ: B1 ØØ      162          LDA   (PTR),Y   GET LAST TRACK
Ø9A2: 18         163          CLC
Ø9A3: 69 Ø1      164          ADC   #1        BUMP BY ONE
Ø9A5: 91 ØØ      165          STA   (PTR),Y   AND PUT IT BACK IN LIST
Ø9A7: C9 11      166          CMP   #$11      CATALOG TRACK?
Ø9A9: FØ F3      167          BEQ   NXTTRK    YES, SKIP OVER THAT ONE
Ø9AB: C9 23      168          CMP   #35       DONE ALL 35 TRACKS?
Ø9AD: BØ Ø3      169          BCS   EXIT      YES, LEAVE
Ø9AF: 4C 24 Ø9   17Ø          JMP   NEWTRK    NO, GO READ FIRST SECTOR
Ø9B2: 6Ø         171  EXIT    RTS

                 173  *      PRTTS: PRINT "T=XX S=XX"

Ø9B3: A9 54      175  PRTTS   LDA   #'T'      PRINT "T"
Ø9B5: 2Ø ED FD   176          JSR   COUT
Ø9B8: AØ Ø4      177          LDY   #RPLTRK
Ø9BA: B1 ØØ      178          LDA   (PTR),Y
Ø9BC: 2Ø CC Ø9   179          JSR   PRTEQ     PRINT "=XX "

Ø9BF: A9 53      181          LDA   #'S'      PRINT "S"
Ø9C1: 2Ø ED FD   182          JSR   COUT
Ø9C4: AØ Ø5      183          LDY   #RPLSEC
Ø9C6: B1 ØØ      184          LDA   (PTR),Y
Ø9C8: 2Ø CC Ø9   185          JSR   PRTEQ     PRINT "=XX "
Ø9CB: 6Ø         186          RTS

Ø9CC: 48         188  PRTEQ   PHA
Ø9CD: A9 3D      189          LDA   #'='
Ø9CF: 2Ø ED FD   19Ø          JSR   COUT
Ø9D2: 68         191          PLA
Ø9D3: 2Ø DA FD   192          JSR   PRBYTE
Ø9D6: A9 2Ø      193          LDA   #'        '
Ø9D8: 2Ø ED FD   194          JSR   COUT
Ø9DB: 6Ø         195          RTS

--End assembly, 237 bytes, Errors: Ø
```

COPY – CONVERT FILES

The COPY program demonstrates the use of the DOS File Manager subroutine package from assembly language. COPY will read as input a Binary type file, stripping off the address and length information, and write the data out as a newly created Text type file. The name of the input file is assumed to be "INPUT", although this could just as easily have been inputted from the keyboard, and the name of the output file is "OUTPUT". COPY is a single drive operation, using the last drive which was referenced.

To run COPY, load it and begin execution at $8ØØ:

```
CALL -151     (Get into the monitor from BASIC)
BLOAD COPY    (Load the COPY program)
```

...Now insert the disk containing INPUT...

```
8ØØG          (Run the COPY program)
```

When COPY finishes, it will return to BASIC. If any errors occur, the return code passed back from the File Manager will be printed. Consult the documentation on the File Manager parameter list in Chapter 6 for a list of these return codes.

```
1    *************************************
2    *                                   *
3    * COPY:THIS PROGRAM DEMONSTRATES    *
4    *      THE USE OF THE DOS FILE      *
5    *      MANAGER BY COPYING A BINARY  *
6    *      FILE TO A TEXT FILE.         *
7    *                                   *
8    * INPUT: INPUT FILE NAME IS "INPUT"  *
9    *        OUTPUT FILE NAME IS "OUTPUT" *
1Ø   *                                   *
11   * ENTRY POINT: $8ØØ                 *
12   *                                   *
13   * PROGRAMMER: DON D WORTH 2/19/81   *
14   *                                   *
15   *************************************
17   BELL     EQU   $87         BELL CHARACTER

19   *       ZPAGE DEFINITIONS

21   PTR      EQU   $Ø          WORK POINTER
22   BUFP     EQU   $2          BUFFER POINTER
23   EBYTE    EQU   $4
24   A1L      EQU   $3C         MONITOR POINTER
25   A2L      EQU   $3E         MONITOR POINTER

27   *       OTHER ADDRESSES

29   BUFFER   EQU   $1ØØØ       DATA BUFFER
3Ø   DOSWRM   EQU   $3DØ        DOS WARMSTART ADDRESS
31   LOCRPL   EQU   $3E3        LOCATE RWTS PARMLIST SUBRTN
```

```
                 32  LOCFPL   EQU   $3DC        LOCATE FILE MGR PARMLIST SUB
                 33  FM       EQU   $3D6        FILE MANAGER ENTRY POINT
                 34  COUT     EQU   $FDED       PRINT ONE CHAR SUBROUTINE
                 35  PRBYTE   EQU   $FDDA       PRINT ONE HEX BYTE SUBRTN

                 37  *       FILE MANAGER PARMLIST DEFINITION

                 39           ORG   $Ø
ØØØØ: ØØ         4Ø  FMOCOD   DS    1           OPERATION CODE
                 41  FMOCOP   EQU   $Ø1            OPEN
                 42  FMOCCL   EQU   $Ø2            CLOSE
                 43  FMOCRD   EQU   $Ø3            READ
                 44  FMOCWR   EQU   $Ø4            WRITE
                 45  FMOCDE   EQU   $Ø5            DELETE
                 46  FMOCCA   EQU   $Ø6            CATALOG
                 47  FMOCLO   EQU   $Ø7            LOCK
                 48  FMOCUN   EQU   $Ø8            UNLOCK
                 49  FMOCRE   EQU   $Ø9            RENAME
                 5Ø  FMOCPO   EQU   $ØA            POSITION
                 51  FMOCIN   EQU   $ØB            INIT
                 52  FMOCVE   EQU   $ØC            VERIFY
ØØØ1: ØØ         53  FMSBCD   DS    1           SUBCODE
                 54  FMSBNO   EQU   $ØØ            NO OPERATION
                 55  FMSBON   EQU   $Ø1            READ/WRITE ONE BYTE
                 56  FMSBRA   EQU   $Ø2            READ/WRITE RANGE OF BYTES
                 57  FMSBPO   EQU   $Ø3            POSITION AND DO ONE BYTE
                 58  FMSBPR   EQU   $Ø4            POSITION AND DO RANGE
ØØØ2: ØØ ØØ ØØ   59  FMPRMS   DS    8           SPECIFIC PARAMETERS
ØØØ5: ØØ ØØ ØØ ØØ ØØ

                 61  *       OPEN PARMS
                 62           ORG   FMPRMS
ØØØ2: ØØ ØØ      63  FMRCLN   DS    2           RECORD LENGTH
ØØØ4: ØØ         64  FMVOL    DS    1           VOLUME
ØØØ5: ØØ         65  FMDRV    DS    1           DRIVE
ØØØ6: ØØ         66  FMSLT    DS    1           SLOT
ØØØ7: ØØ         67  FMTYPE   DS    1           TYPE
                 68  FMTYPT   EQU   Ø              TEXT
                 69  FMTYPI   EQU   1              INTEGER
                 7Ø  FMTYPA   EQU   2              APPLESOFT
                 71  FMTYPB   EQU   4              BINARY
ØØØ8: ØØ ØØ      72  FMNAME   DS    2           ADDRESS OF FILE NAME

                 74  *       READ/WRITE PARMS
                 75           ORG   FMPRMS
ØØØ2: ØØ ØØ      76  FMRCNM   DS    2           RECORD NUMBER
ØØØ4: ØØ ØØ      77  FMOFFS   DS    2           BYTE OFFSET
ØØØ6: ØØ ØØ      78  FMRALN   DS    2           RANGE LENGTH
ØØØ8: ØØ ØØ      79  FMRAAD   DS    2           RANGE ADDRESS
                 8Ø  FMDATA   EQU   FMRAAD      DATA BYTE READ/WRITTEN

                 82  *       RENAME PARMS
                 83           ORG   FMPRMS
ØØØ2: ØØ ØØ      84  FMNNAM   DS    2           ADDRESS OF NEW NAME

                 86  *       INIT PARMS
                 87           ORG   FMPRMS
                 88  FMPAGE   EQU   FMSBCD      FIRST PAGE OF DOS IMAGE

                 9Ø  *       COMMON PARMS
                 91           ORG   FMPRMS+8
ØØØA: ØØ         92  FMRC     DS    1           RETURN CODE
                 93  FMRCOK   EQU   Ø              NO ERRORS
                 94  FMRCBO   EQU   2              BAD OPCODE
                 95  FMRCBS   EQU   3              BAD SUBCODE
```

```
                  96   FMRCWP   EQU   4              WRITE PROTECTED
                  97   FMRCED   EQU   5              END OF DATA
                  98   FMRCNF   EQU   6              FILE NOT FOUND
                  99   FMRCBV   EQU   7              BAD VOLUME
                  100  FMRCIO   EQU   8              I/O ERROR
                  101  FMRCDF   EQU   9              DISK FULL
                  102  FMRCLK   EQU   10             FILE LOCKED
000B: 00          103           DS    1          NOT USED
000C: 00 00       104  FMFMWA   DS    2          FILE MANAGER WORKAREA PTR
000E: 00 00       105  FMTSL    DS    2          T/S LIST PTR
0010: 00 00       106  FMBUFF   DS    2          DATA BUFFER PTR
                  107           ORG   $800

                  109  *       LOCATE FM PARMLIST

0800: 20 DC 03    111  COPY     JSR   LOCFPL     FIND PARMLIST
0803: 84 00       112           STY   PTR        SET UP POINTER TO IT
0805: 85 01       113           STA   PTR+1

                  115  *       OPEN INPUT FILE

0807: A0 08       117           LDY   #FMNAME    STORE INPUT FILE NAME
0809: A9 AC       118           LDA   #<INAME    PTR IN LIST
080B: 91 00       119           STA   (PTR),Y
080D: C8          120           INY
080E: A9 09       121           LDA   #>INAME
0810: 91 00       122           STA   (PTR),Y
0812: A0 07       123           LDY   #FMTYPE    BINARY FILE AS INPUT
0814: A9 04       124           LDA   #FMTYPB
0816: 91 00       125           STA   (PTR),Y
0818: A2 01       126           LDX   #1         OLD FILE EXPECTED
081A: 20 D3 08    127           JSR   OPEN       AND OPEN THE FILE
081D: 90 03       128           BCC   INOP
081F: 4C BC 08    129           JMP   ERROR      ANY ERROR IS FATAL

0822: A5 02       131  INOP     LDA   BUFP
0824: 8D EA 09    132           STA   IBUFF      SAVE OPEN FILE BUFFER
0827: A5 03       133           LDA   BUFP+1
0829: 8D EB 09    134           STA   IBUFF+1
082C: 20 5A 09    135           JSR   REWIND     POSITION TO START OF FILE

                  137  *       OPEN OUTPUT FILE

082F: A0 08       139           LDY   #FMNAME    STORE OUTPUT FILE NAME
0831: A9 CA       140           LDA   #<ONAME    PTR IN LIST
0833: 91 00       141           STA   (PTR),Y
0835: C8          142           INY
0836: A9 09       143           LDA   #>ONAME
0838: 91 00       144           STA   (PTR),Y
083A: A0 07       145           LDY   #FMTYPE    TEXT FILE AS OUTPUT
083C: A9 00       146           LDA   #FMTYPT
083E: 91 00       147           STA   (PTR),Y
0840: A2 00       148           LDX   #0         NEW FILE IS OK
0842: 20 D3 08    149           JSR   OPEN
0845: 90 0B       150           BCC   OUTOP
0847: A0 0A       151           LDY   #FMRC
0849: B1 00       152           LDA   (PTR),Y
084B: C9 06       153           CMP   #FMRCNF    FILE NOT FOUND?
084D: F0 03       154           BEQ   OUTOP      YES, WAS ALLOCATED THEN
084F: 4C BC 08    155           JMP   ERROR

0852: A5 02       157  OUTOP    LDA   BUFP       SAVE OPEN OUTPUT FILE BUFFER
0854: 8D E8 09    158           STA   OBUFF
0857: A5 03       159           LDA   BUFP+1
0859: 8D E9 09    160           STA   OBUFF+1
```

```
Ø85C: 2Ø 5A Ø9  161           JSR   REWIND     POSITION TO START OF FILE

                163  *       READ ADDRESS/LENGTH FROM BINARY FILE

Ø85F: A9 Ø4     165           LDA   #4         READ 4 BYTES FIRST
Ø861: AØ Ø6     166           LDY   #FMRALN
Ø863: 91 ØØ     167           STA   (PTR),Y
Ø865: A9 ØØ     168           LDA   #Ø
Ø867: C8        169           INY
Ø868: 91 ØØ     17Ø           STA   (PTR),Y
Ø86A: 2Ø 74 Ø9  171           JSR   READ

                173  *       READ ENTIRE BINARY FILE INTO MEMORY AT $1ØØØ

Ø86D: AD Ø2 1Ø  175           LDA   BUFFER+2   COPY DATA LENGTH TO LIST
Ø87Ø: AØ Ø6     176           LDY   #FMRALN
Ø872: 91 ØØ     177           STA   (PTR),Y
Ø874: AD Ø3 1Ø  178           LDA   BUFFER+3
Ø877: C8        179           INY
Ø878: 91 ØØ     18Ø           STA   (PTR),Y
Ø87A: 18        181           CLC
Ø87B: AD Ø2 1Ø  182           LDA   BUFFER+2   COMPUTE ENDING BYTE
Ø87E: 48        183           PHA
Ø87F: 69 ØØ     184           ADC   #<BUFFER
Ø881: 85 Ø4     185           STA   EBYTE
Ø883: AD Ø3 1Ø  186           LDA   BUFFER+3
Ø886: 48        187           PHA
Ø887: 69 1Ø     188           ADC   #>BUFFER
Ø889: 85 Ø5     189           STA   EBYTE+1
Ø88B: 2Ø 74 Ø9  19Ø           JSR   READ       READ BLOB INTO MEMORY

                192  *       WRITE ENTIRE BLOB OUT INTO TEXT FILE

Ø88E: AØ ØØ     194           LDY   #Ø
Ø89Ø: 98        195           TYA
Ø891: 91 Ø4     196           STA   (EBYTE),Y  MARK END OF FILE
Ø893: 68        197           PLA
Ø894: AØ Ø7     198           LDY   #FMRALN+1  SET RANGE LENGTH
Ø896: 91 ØØ     199           STA   (PTR),Y
Ø898: 88        2ØØ           DEY
Ø899: 68        2Ø1           PLA
Ø89A: 91 ØØ     2Ø2           STA   (PTR),Y
Ø89C: 2Ø 82 Ø9  2Ø3           JSR   WRITE      WRITE BLOB FROM MEMORY

                2Ø5  *       WHEN FINISHED, CLOSE FILES

Ø89F: AD E8 Ø9  2Ø7  EXIT     LDA   OBUFF
Ø8A2: 85 Ø2     2Ø8           STA   BUFP
Ø8A4: AD E9 Ø9  2Ø9           LDA   OBUFF+1
Ø8A7: 85 Ø3     21Ø           STA   BUFP+1
Ø8A9: 2Ø 46 Ø9  211           JSR   CLOSE      CLOSE OUTPUT FILE
Ø8AC: AD EA Ø9  212           LDA   IBUFF
Ø8AF: 85 Ø2     213           STA   BUFP
Ø8B1: AD EB Ø9  214           LDA   IBUFF+1
Ø8B4: 85 Ø3     215           STA   BUFP+1
Ø8B6: 2Ø 46 Ø9  216           JSR   CLOSE      CLOSE INPUT FILE
Ø8B9: 4C DØ Ø3  217           JMP   DOSWRM     BACK TO DOS

                219  *       ERROR, PRINT "ERRXX"

Ø8BC: AØ ØA     221  ERROR    LDY   #FMRC      FIND RETURN CODE
Ø8BE: B1 ØØ     222           LDA   (PTR),Y
Ø8CØ: 48        223           PHA
Ø8C1: A9 45     224  ERR      LDA   #'E'       PRINT "ERR"
Ø8C3: 2Ø ED FD  225           JSR   COUT
```

```
Ø8C6: A9 52     226          LDA   #'R'
Ø8C8: 2Ø ED FD  227          JSR   COUT
Ø8CB: 2Ø ED FD  228          JSR   COUT
Ø8CE: 68        229          PLA
Ø8CF: 2Ø DA FD  23Ø          JSR   PRBYTE     PRINT HEX CODE
Ø8D2: ØØ        231          BRK   DIE        HORRIBLY

                233  *      OPEN: COMPLETE PARMLIST AND OPEN FILE

Ø8D3: AD D2 Ø3  235  OPEN    LDA   DOSWRM+2   FIND DOS ENTRY
Ø8D6: 85 Ø3     236          STA   BUFP+1
Ø8D8: AØ ØØ     237          LDY   #Ø
Ø8DA: 84 Ø2     238          STY   BUFP       POINT AT BUFFER CHAIN

                24Ø  *      SCAN DOS BUFFERS FOR A FREE ONE

Ø8DC: B1 Ø2     242  GBUFØ   LDA   (BUFP),Y   LOCATE NEXT DOS BUFFER
Ø8DE: 48        243          PHA
Ø8DF: C8        244          INY
Ø8EØ: B1 Ø2     245          LDA   (BUFP),Y
Ø8E2: 85 Ø3     246          STA   BUFP+1
Ø8E4: 68        247          PLA
Ø8E5: 85 Ø2     248          STA   BUFP
Ø8E7: DØ ØA     249          BNE   GBUF       GOT ONE
Ø8E9: A5 Ø3     25Ø          LDA   BUFP+1
Ø8EB: DØ Ø6     251          BNE   GBUF       GOT ONE

Ø8ED: A9 ØC     253          LDA   #12        NO FILE BUFFERS RETURN CODE
Ø8EF: 48        254          PHA
Ø8FØ: 4C C1 Ø8  255          JMP   ERR        GO PRINT MESSAGE

Ø8F3: AØ ØØ     257  GBUF    LDY   #Ø         LOOK AT FILENAME
Ø8F5: B1 Ø2     258          LDA   (BUFP),Y
Ø8F7: FØ Ø4     259          BEQ   GOTBUF     NONE THERE, FREE BUFFER
Ø8F9: AØ 24     26Ø          LDY   #36        IT'S NOT FREE
Ø8FB: DØ DF     261          BNE   GBUFØ      GO GET NEXT ONE

Ø8FD: A9 Ø1     263  GOTBUF  LDA   #1
Ø8FF: 91 Ø2     264          STA   (BUFP),Y   MARK BUFFER IN USE

                266  *      FINISH COMPLETING OPEN LIST

Ø9Ø1: AØ ØØ     268          LDY   #FMOCOD
Ø9Ø3: A9 Ø1     269          LDA   #FMOCOP
Ø9Ø5: 91 ØØ     27Ø          STA   (PTR),Y    SET OPCODE TO OPEN
Ø9Ø7: A9 ØØ     271          LDA   #Ø
Ø9Ø9: AØ Ø2     272          LDY   #FMRCLN
Ø9ØB: 91 ØØ     273          STA   (PTR),Y    SET RECORD LENGTH TO Ø
Ø9ØD: C8        274          INY
Ø9ØE: 91 ØØ     275          STA   (PTR),Y
Ø91Ø: AØ Ø4     276          LDY   #FMVOL
Ø912: 91 ØØ     277          STA   (PTR),Y    AND VOLUME (ANY VOL)

Ø914: 2Ø E3 Ø3  279          JSR   LOCRPL     FIND RWTS PARMS
Ø917: 84 3C     28Ø          STY   A1L
Ø919: 85 3D     281          STA   A1L+1
Ø91B: AØ Ø1     282          LDY   #1
Ø91D: B1 3C     283          LDA   (A1L),Y    GET SLOT*16
Ø91F: 4A        284          LSR   A
Ø92Ø: 4A        285          LSR   A
Ø921: 4A        286          LSR   A
Ø922: 4A        287          LSR   A          SLOT=SLOT/16
Ø923: AØ Ø6     288          LDY   #FMSLT
Ø925: 91 ØØ     289          STA   (PTR),Y    STORE IN LIST
Ø927: AØ Ø2     29Ø          LDY   #2
```

```
Ø929: B1 3C     291          LDA   (A1L),Y    GET DRIVE
Ø92B: AØ Ø5     292          LDY   #FMDRV
Ø92D: 91 ØØ     293          STA   (PTR),Y    AND SLOT

                295 *       COMMON INTERFACE TO FILE MANAGER

Ø92F: AØ 1E     297 CALLFM   LDY   #3Ø
Ø931: B1 Ø2     298 CFMLP1   LDA   (BUFP),Y   GET THREE BUFFER PTRS
Ø933: 48        299          PHA
Ø934: C8        3ØØ          INY
Ø935: CØ 24     3Ø1          CPY   #36
Ø937: 9Ø F8     3Ø2          BCC   CFMLP1

Ø939: AØ 11     3Ø4          LDY   #FMBUFF+1
Ø93B: 68        3Ø5 CFMLP2   PLA
Ø93C: 91 ØØ     3Ø6          STA   (PTR),Y    COPY THEM TO FM LIST
Ø93E: 88        3Ø7          DEY
Ø93F: CØ ØC     3Ø8          CPY   #FMFMWA
Ø941: BØ F8     3Ø9          BCS   CFMLP2

Ø943: 4C D6 Ø3  311          JMP   FM         EXIT THRU FILE MANAGER

                313 *       CLOSE: CLOSE DOS FILE

Ø946: AØ ØØ     315 CLOSE    LDY   #FMOCOD
Ø948: A9 Ø2     316          LDA   #FMOCCL
Ø94A: 91 ØØ     317          STA   (PTR),Y
Ø94C: 2Ø 2F Ø9  318          JSR   CALLFM     CLOSE FILE
Ø94F: 9Ø Ø3     319          BCC   CLOK
Ø951: 4C BC Ø8  32Ø          JMP   ERROR
Ø954: AØ ØØ     321 CLOK     LDY   #Ø         FREE BUFFER
Ø956: 98        322          TYA
Ø957: 91 Ø2     323          STA   (BUFP),Y
Ø959: 6Ø        324          RTS

                326 *       REWIND: POSITION TO START OF FILE

Ø95A: AØ Ø2     328 REWIND   LDY   #FMRCNM
Ø95C: A9 ØØ     329          LDA   #Ø
Ø95E: 91 ØØ     33Ø REWLP    STA   (PTR),Y    ZERO RECORD NUMBER AND..
Ø96Ø: C8        331          INY
Ø961: CØ Ø6     332          CPY   #FMOFFS+2  BYTE OFFSET.
Ø963: 9Ø F9     333          BCC   REWLP
Ø965: AØ ØØ     334          LDY   #FMOCOD
Ø967: A9 ØA     335          LDA   #FMOCPO    POSITION OPCODE
Ø969: 91 ØØ     336          STA   (PTR),Y
Ø96B: 2Ø 2F Ø9  337          JSR   CALLFM     EXIT VIA FILE MANAGER
Ø96E: 9Ø Ø3     338          BCC   REWRTS     CHECK FOR ERRORS
Ø97Ø: 4C BC Ø8  339          JMP   ERROR
Ø973: 6Ø        34Ø REWRTS   RTS

                342 *       READ: READ A RANGE OF BYTES TO $1ØØØ

Ø974: AD EA Ø9  344 READ     LDA   IBUFF      FIND PROPER BUFFER
Ø977: 85 Ø2     345          STA   BUFP
Ø979: AD EB Ø9  346          LDA   IBUFF+1
Ø97C: 85 Ø3     347          STA   BUFP+1
Ø97E: A9 Ø3     348          LDA   #FMOCRD    READ OPCODE
Ø98Ø: DØ ØC     349          BNE   DOIO       GO DO COMMON CODE

                351 *       WRITE: WRITE A RANGE OF BYTES FROM $1ØØØ

Ø982: AD E8 Ø9  353 WRITE    LDA   OBUFF      FIND PROPER BUFFER
Ø985: 85 Ø2     354          STA   BUFP
Ø987: AD E9 Ø9  355          LDA   OBUFF+1
```

```
Ø98A: 85 Ø3      356            STA   BUFP+1
Ø98C: A9 Ø4      357            LDA   #FMOCWR    WRITE OPCODE
                 358  *        BNE    DOIO

                 36Ø  *        DOIO: READ/WRITE A RANGE OF BYTES

Ø98E: AØ ØØ      362  DOIO      LDY   #FMOCOD
Ø99Ø: 91 ØØ      363            STA   (PTR),Y    SET OPCODE
Ø992: AØ Ø1      364            LDY   #FMSBCD
Ø994: A9 Ø2      365            LDA   #FMSBRA
Ø996: 91 ØØ      366            STA   (PTR),Y    DO RANGE OF BYTES
Ø998: AØ Ø8      367            LDY   #FMRAAD
Ø99A: A9 ØØ      368            LDA   #<BUFFER
Ø99C: 91 ØØ      369            STA   (PTR),Y    RANGE ADDRESS=$1ØØØ
Ø99E: C8         37Ø            INY
Ø99F: A9 1Ø      371            LDA   #>BUFFER
Ø9A1: 91 ØØ      372            STA   (PTR),Y
Ø9A3: 2Ø 2F Ø9   373            JSR   CALLFM     CALL FM TO DO I/O OPERATION
Ø9A6: 9Ø Ø3      374            BCC   DOIORT
Ø9A8: 4C BC Ø8   375            JMP   ERROR
Ø9AB: 6Ø         376  DOIORT    RTS

                 378  *        DATA

Ø9AC: 49 4E 5Ø   38Ø  INAME     ASC   'INPUT                         '
Ø9AF: 55 54 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
Ø9B7: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
Ø9BF: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
Ø9C7: 2Ø 2Ø 2Ø
Ø9CA: 4F 55 54   381  ONAME     ASC   'OUTPUT                        '
Ø9CD: 5Ø 55 54 2Ø 2Ø 2Ø 2Ø 2Ø
Ø9D5: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
Ø9DD: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
Ø9E5: 2Ø 2Ø 2Ø

Ø9E8: ØØ ØØ      383  OBUFF     DS    2
Ø9EA: ØØ ØØ      384  IBUFF     DS    2

--End assembly, 528 bytes, Errors: Ø
```

DOS Ø.Ø1

Appendix B
Disk Protection Schemes

As the quantity and quality of Apple II software has increased, so has the incidence of illegal duplication of copyrighted software. To combat this, software vendors have introduced methods for protecting their software. Since most protection schemes involve a modified or custom Disk Operating System, it seems appropriate to discuss disk protection in general.

Typically, a protection scheme's purpose is to stop unauthorized duplication of the contents of the diskette, although it may also include, or be limited to, preventing the listing of the software (if it is in BASIC). This has been attempted in a variety of ways, all of which necessitate reading and writing non-standard formats on the disk. If the reader is unclear about how a normal diskette is formatted, refer to Chapter 3 for more information.

Early protection methods were primitive in comparison to what is being done now. Just as the methods of protection have improved, so have the techniques people have used to break them. The cycle seems endless. As new and more sophisticated schemes are developed, they are soon broken, prompting the software vendor to try to create even more sophisticated systems.

It seems reasonable at this time to say that it is impossible to protect a disk in such a way that it can't be broken. This is, in large part, due to the fact that the diskette must be "bootable"; i.e. that it must contain at least one sector (Track 0, Sector 0) which can be read by the program in the PROM on the disk controller card. This means that it is possible to trace the boot process by disassembling the normal sector or sectors that must be on the disk. It turns out that it is even possible to protect these sectors. Because of a lack of space on the PROM (256 bytes), the software doesn't fully check either the Address Field or the Data Field. But potential protection schemes which take advantage of this are limited and must involve only certain changes which will be discussed below.

Most protected disks use a modified version of Apple's DOS. This is a much easier task than writing one's own Disk Operating System and will be the primary area covered by this discussion.

Although there are a vast array of different protection schemes, they all consist of having some portion of the disk unreadable by a normal Disk Operating System. The two logical areas to alter are the Address Field and the Data Field. Each include a number of bytes which, if changed, will cause a sector to be unreadable. We will examine how that is done in some detail.

The Address Field normally starts with the bytes `$D5/$AA/$96`. If any one of these bytes were changed, DOS would not be able to locate that particular Address Field, causing an error. While all three bytes can and have been changed by various schemes, it is important to remember that they must be chosen in such a way as to guarantee their uniqueness. Apple's DOS does this by

Disk snooping

reserving the bytes `$D5` and `$AA`; i.e. these bytes are not used in the storage of data. The sequence chosen by the would-be disk protector can not occur anywhere else on the track, other than in another Address Field. Next comes the address information itself (volume, track, sector, and checksum). Some common techniques include changing the order of the information, doubling the sector numbers, or altering the checksum with some constant. Any of the above would cause an I/O error in a normal DOS. Finally, we have the two closing bytes (`$DE/$AA`), which are similar to the starting bytes, but with a difference. Their uniqueness is not critical, since DOS will read whatever

two bytes follow the information field, using them for verification, but not to locate the field itself.

The Data Field is quite similar to the Address Field in that its three parts correspond almost identically, as far as protection schemes are concerned. The Data Field starts with `$D5/$AA/$AD`, only the third byte being different, and all that applies to the Address Field applies here also. Switching the third bytes between the two fields is an example of a protective measure. The data portion consists of 342 bytes of data, followed by a checksum byte. Quite often the data is written so that the checksum computation will be non-zero, causing an error. The closing bytes are identical to those of the Address Field (`$DE/$AA`).

As mentioned earlier, the PROM on the disk controller skips certain parts of both types of fields. In particular, neither trailing byte (`$DE/$AA`) is read or verified nor is the checksum tested, allowing these bytes to be modified even in track 0 sector 0. However, this protection is easily defeated by making slight modifications to DOS's RWTS routines, rendering it unreliable as a protective measure.

In the early days of disk protection, a single alteration was all that was needed to stop all but a few from copying the disk. Now, with more educated users and powerful utilities available, multiple schemes are quite commonly used. The first means of protection was probably that of hidden control characters imbedded in a file name. Now it is common to find a disk using multiple non-standard formats written even between tracks.

A state of the art protection scheme consists of two elements. First, the data is stored on the diskette in some non-standard way in order to make copying very difficult. Secondly, some portion of memory is utilized that will be altered upon a RESET. (For example, the primary text page or certain zero page locations) This is to prevent the software from being removed from memory intact.

Recently, several "nibble" or byte copy programs have become available. Unlike traditional copy programs which require the data to be in a predefined format, these utilities make as few assumptions as possible about the data structure. Ever since protected disks were first introduced, it has been asked, "why can't a track be read into memory and then written back out to another diskette in exactly the same way?". The problem lies with the self-sync or auto-sync bytes. (For a full discussion see Chapter 3) These bytes contain extra zero bits that are lost when read into memory. In memory it is impossible to determine the difference between a hexadecimal `$FF` that was data and a hex `$FF` that was a self-sync byte. Two solutions are currently being implemented in nibble copy programs. One is to analyze the data on a track with the hope

that the sync gaps can be located by deduction. This has a high probability of success if 13 or 16 sectors are present, even if they have been modified, but may not be effective in dealing with non-standard sectoring where sectors are larger than 256 bytes. In short, this method is effective but by no means foolproof. The second method is simple but likewise has a difficulty. It simply writes every hex `$FF` found on the track as if it were a sync byte. This, however, will expand the physical space needed to write the track back out, since sync bytes require 25% more room. If enough hex `$FF`s occur in the data, the track will overwrite itself. This can happen in general if the drive used to write the data is significantly slower than normal. Thus, we are back to having to analyze the data and, in effect, make some assumptions. It appears that, apart from using some hardware device to help find the sync bytes, a software program must make some assumptions about how the data is structured on the diskette.

The result of the introduction of nibble copy programs has been to "force the hand" of the software vendors. The initial response was to develop new protection schemes that defeated the nibble copy programs. More recent protection schemes, however, involve hardware and timing dependencies which require current nibble copy programs to rely heavily upon the user for direction. If the present trend continues, it is very likely that protection schemes will evolve to a point where automated techniques cannot be used to defeat them.

DOS 3.2 and DOS 3.3 meet ProDOS

Beneath Apple ProDOS

Contents

CHAPTER 1: INTRODUCTION 155

CHAPTER 2: TO BUILD A BETTER DOS 157
- THE DEFICIENCIES OF DOS 157
- ENTER PRODOS 158
- MORE PRODOS ADVANTAGES 16Ø
- WHAT YOU GIVE UP WITH PRODOS 161
- OTHER DIFFERENCES BETWEEN PRODOS AND DOS 163

CHAPTER 3: DISK II HARDWARE AND DISKETTE FORMATTING 165
- TRACKS AND SECTORS 166
- TRACK FORMATTING 167
- ADDRESS FIELDS 174
- DATA FIELDS 175
- DISK II BLOCK AND SECTOR INTERLEAVING 176
  - INTRA-BLOCK INTERLEAVING 178
  - INTER-BLOCK INTERLEAVING 178
- READING OR WRITING A BLOCK 179
  - READING OR WRITING CONSECUTIVE BLOCKS 18Ø

CHAPTER 4: VOLUMES, DIRECTORIES, AND FILES 183
- THE DISKETTE VOLUME 184
- VOLUME OVERHEAD 185
- VOLUME SPACE ALLOCATION — THE VOLUME BIT MAP 186
- THE VOLUME DIRECTORY 187
- THE VOLUME DIRECTORY HEADER 188
- FILE DESCRIPTIVE ENTRIES 19Ø
- FILE STRUCTURES 194
- FILE DATA TYPES 199
  - TXT FILES 2ØØ
  - BIN FILES 2Ø3
  - BAS FILES 2Ø4

OTHER FILE TYPES (VAR, REL, SYS)..2Ø4
DIR FILES — PRODOS SUBDIRECTORIES 2Ø6
EMERGENCY REPAIRS...21Ø
FRAGMENTATION 213

CHAPTER 5: THE STRUCTURE OF PRODOS.......................................215
PRODOS MEMORY USE...215
GLOBAL PAGES..218
WHAT HAPPENS DURING BOOTING...22Ø

CHAPTER 6: USING PRODOS FROM ASSEMBLY LANGUAGE...........................225
CAVEAT 225
DIRECT USE OF THE DISKETTE DRIVE 225
STEPPER PHASE OFF OR ON..227
MOTOR OFF OR ON..227
ENGAGE DRIVE 1 OR 2 228
READ A BYTE 228
SENSE WRITE PROTECT..228
WRITE LOAD AND WRITE A BYTE 228
CALLING A STORAGE DEVICE DRIVER (BLOCK ACCESS) 229
CALLING THE DISK II DEVICE DRIVER 231
DEVICE DRIVER PARAMETER LISTS BY COMMAND CODE.............................232
CALLING THE MACHINE LANGUAGE INTERFACE 234
MLI PARAMETER LISTS BY FUNCTION CODE......................................237
MLI ERROR CODES 268
PASSING COMMAND LINES TO THE BASIC INTERPRETER 27Ø
COMMON ALGORITHMS..272
IS PRODOS ACTIVE?..272
WHAT KIND OF MACHINE IS THIS? 272
HOW MUCH MEMORY IS IN THIS MACHINE?.......................................272
GIVEN A PAGE NUMBER, SEE IF IT IS FREE 273
IS A BASIC PROGRAM RUNNING?...273
SETTING UP YOUR OWN RESET VECTOR 273
ACTIVATE A PRINTER OR OTHER PERIPHERAL 274

CHAPTER 7: CUSTOMIZING PRODOS 275
SYSTEM PROGRAMMING WITH PRODOS..275
INSTALLING A PROGRAM BETWEEN THE BI AND ITS BUFFERS.......................277
ADDING COMMANDS TO THE PRODOS BASIC INTERPRETER............................278
DISABLE /RAM VOLUME FOR 128K MACHINES.....................................279
WRITING YOUR OWN INTERPRETER..281
INSTALLING NEW PERIPHERAL DRIVERS...283
INSTALLING AN INTERRUPT HANDLER...285
DIRECT MODIFICATION OF PRODOS — A WORD OF WARNING 287
APPLYING PATCHES TO PRODOS 287

CHANGING THE NAME OF THE STARTUP FILE 29Ø
PUT THE CURSOR ON COMMAND THAT CAUSED PRODOS ERROR 29Ø
HOW TO WRITE TO A DIRECTORY FILE 29Ø
CREATING A NEW FILE TYPE 291
RECOVERING DATA FROM A DAMAGED DISK 292
USING PRODOS WITH 40-TRACK DRIVES 292
FORCING PRODOS TO LOAD IN 48K 294

CHAPTER 8: PRODOS GLOBAL PAGES 295
BASIC INTERPRETER GLOBAL PAGE 296
PRODOS SYSTEM GLOBAL PAGE — MLI GLOBAL PAGE 299

APPENDIX A: EXAMPLE PROGRAMS 3Ø3
STORING THE PROGRAMS ON DISKETTE 3Ø4

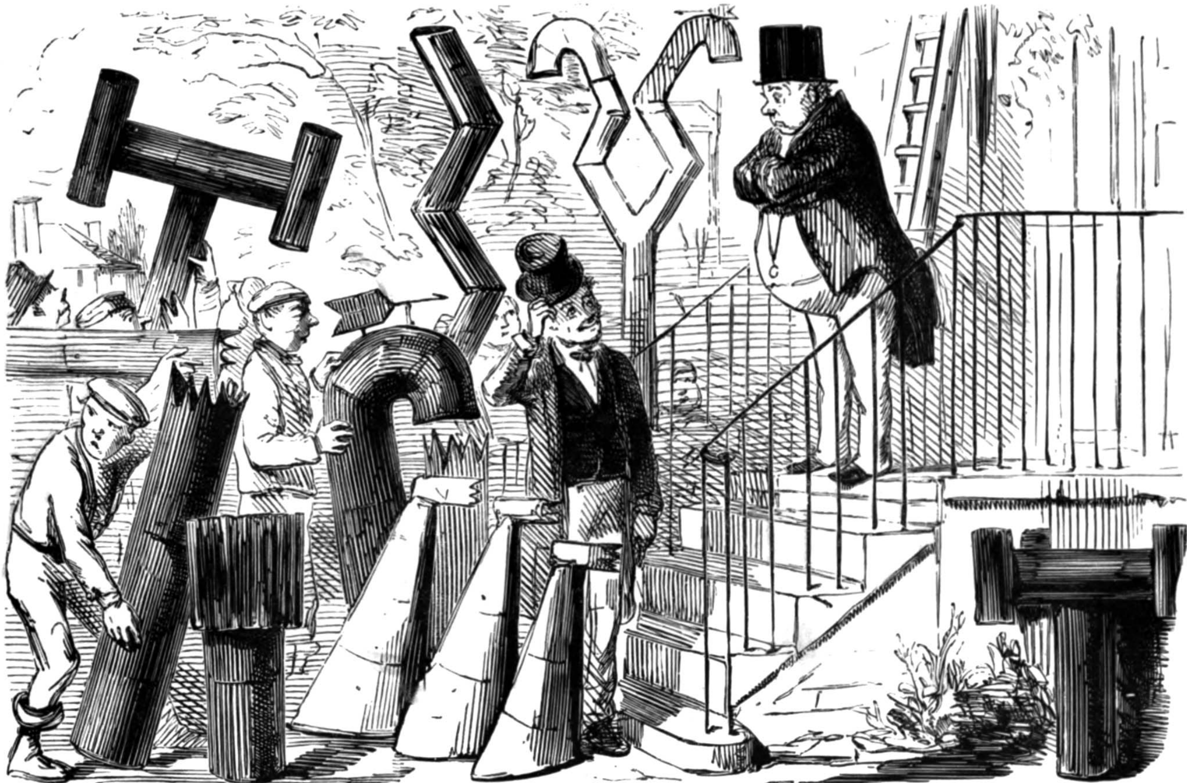
APPENDIX B: DISKETTE PROTECTION SCHEMES 345
A BRIEF HISTORY OF APPLE SOFTWARE PROTECTION 345
PROTECTION METHODS 346
FORMAT ALTERATION 347
SIGNATURE 35Ø
MEMORY PROTECTION 353
CODE PROTECTION 353
THE IDEAL PROTECTION SCHEME 354
NIBBLE COPIERS 354
HARDWARE BOARDS 354
FRONT DOOR METHOD 354

APPENDIX C: NIBBLIZING 357
ENCODING TECHNIQUES 357
“4 AND 4” ENCODING 358
“6 AND 2” ENCODING 359
THE ENCODING PROCESS 36Ø
STAGE 1 367
STAGE 2 368
STAGE 3 369

APPENDIX D: THE LOGIC STATE SEQUENCER 371
LOGIC STATE SEQUENCER ROM 371
SEQUENCER EXAMPLE 377

APPENDIX E: PRODOS, DOS, AND SOS 379
CONVERTING FROM DOS TO PRODOS 379
WRITING PROGRAMS FOR PRODOS AND SOS 38Ø

APPENDIX F: DIFFERENCES BETWEEN THE PRODOS 8 VERSIONS 383
CHANGES INTRODUCED IN THE 1.2 VERSION 383
RELOCATOR 383
MLI AND MI GLOBAL PAGE 384
QUIT CODE 385
CLOCK CODE 385
CHANGES INTRODUCED IN THE 1.3 VERSION 385
RELOCATOR 386
MLI 386
DISK II DEVICE DRIVER 386
BUGS IN VERSIONS 1.2 AND 1.3 386



Pardon me, sir, but do any of these look like your file?

Chapter 1
Introduction

Beneath Apple ProDOS is intended to serve as a companion to the manuals provided by Apple Computer, Inc. for ProDOS, providing additional information for the advanced programmer or for the novice Apple user who wants to know more about the structure of disks. It is not the intent of this manual to replace the documentation provided by Apple. Although, for the sake of continuity, some of the material covered in the Apple manuals is also covered here, it will be assumed that the reader is reasonably familiar with the contents of Apple's *ProDOS User's Manual* and *BASIC Programming With ProDOS*. Since all chapters presented here may not be of use to each Apple owner, each has been written to stand on its own. Readers of our earlier book, *Beneath Apple DOS*, will notice that we have retained the basic organization of that book in an attempt to help them familiarize themselves with *Beneath Apple ProDOS* more quickly.

The information presented here is a result of intensive disassembly and annotation of various versions of ProDOS by the authors. It also uses as a reference various application notes and preliminary documentation from Apple. Although no guarantee can be made concerning the accuracy of the information presented here, all of the material included in *Beneath Apple ProDOS* has been thoroughly researched and tested.

There were several reasons for writing *Beneath Apple ProDOS*:

- To show how to access ProDOS and/or the Disk II drive directly from machine language.
- To help you fix damaged disks.
- To correct errors and omissions in the Apple documentation.
- To allow you to customize ProDOS to fit your needs.
- To provide complete information on the diskette formatting.
- To document the internal logic of ProDOS.
- To present a critical, non-Apple perspective of ProDOS.
- To provide more examples of ProDOS programming.
- To help you learn about how an operating system works.

When Apple introduced ProDOS Version 1.0.1 in January 1984, three manuals were available: the *ProDOS User's Manual* documents the use of ProDOS utilities; the *BASIC Programming With ProDOS* manual describes the command language supported by the BASIC Interpreter and how to write BASIC programs which access the disk; and the *ProDOS Technical Reference Manual (for the Apple II family)* documents the assembly language interfaces to ProDOS. It should be stated that this technical reference manual represents the best internal documentation Apple has ever provided to users of one of their operating systems.

Unfortunately, the *ProDOS Technical Reference Manual* documents a prerelease version of ProDOS, and is not entirely accurate for the current release at the time of this writing. In addition, many sections require further explanation before the interfaces they describe can be used at all. For example, the discussion of how one adds a command to the BASIC interpreter omits several vital pieces of information which are documented fully in *Beneath Apple ProDOS*. In addition, none of the Apple documentation addresses diskette formatting or direct access of the Disk II family of controllers from assembly language. *Beneath Apple ProDOS* was written in an attempt to improve upon the documentation base established by Apple. Most of the topics covered by Apple's technical manual are covered bere also, but they are explained in a different and, we hope, clearer way, based upon a programmer's understanding of the code in the ProDOS Kernel and the BASIC Interpreter.

We have also added substantial information on diskette formatting and repair, the internal logic and structure of ProDOS, and customizing techniques, as well as providing several example programs and quick reference materials.

In addition to the ProDOS specific information provided, many of the discussions also apply to other operating systems in the Apple II and Apple III family of machines. For example, disk formatting at the track and sector level is for the most part the same. Also, the format of a ProDOS volume is nearly identical to that of an Apple III SOS volume.

Chapter 2
To Build a Better DOS

From June 1978 to January 1984, the primary disk operating system for the Apple II family was Apple DOS. Throughout its first six years of existence, DOS has gone through a number of changes, culminating in its final version, DOS 3.3. DOS was originally designed primarily to support the BASIC programmer, but has since been adopted by assembly language programmers and by the majority of Apple users for a variety of applications.

THE DEFICIENCIES OF DOS

Although it is a flexible and easy to use operating system, DOS suffers from many weaknesses. Among these are:

- **DOS is slow.** Since each byte read from the disk is copied between memory buffers up to three times, a large portion of the actual overhead in reading data from the disk is in processor manipulation after the data has been read. To circumvent this, several "fast DOS" packages have been marketed by third parties which heavily modify DOS to prevent multiple buffering under certain circumstances.
- **DOS is device dependent.** When DOS was developed, the only mass storage device for the Apple was the Disk II diskette drive. Now that diskette drives with increased capacity and hard disks are available, a more device independent file organization is needed. DOS is limited in the number of files which can be stored on a diskette as well as their maximum size. These are significant drawbacks when a hard disk with five million bytes or more is used.
- Over the years, new hardware has been introduced by Apple and other manufacturers which DOS does not intrinsically support. The Apple IIe with its 80-column card and the Thunderclock are examples.
- **DOS is difficult to customize.** There are few external "hooks" provided to allow system programmers the opportunity to personalize the operating

system to special applications. For example, a new command cannot be added to DOS without version dependent patches.

- DOS file structures and system calls are **incompatible** with other operating systems. Each operating system Apple has announced in the past has had its on way of organizing data on a diskette. There is no compatibility between DOS, SOS and the Apple Pascal system. This means that special utilities must be written to move data between these systems and that applications developed in one environment will not run without major modifications under any other system.
- DOS does not provide a consistent mechanism for supporting multiple peripherals which can generate hardware **interrupts**. In the past, various manufacturers have implemented interrupt handlers on their own, often resulting in incompatibilities between their devices.
- DOS provides little standardization of **memory use** and of operating system interfaces. Most "interesting" locations within DOS are internalized and therefore not officially available to the programmer. Also, since there is no standard way to set aside portions of memory for specific applications, it is difficult to put a program in a "safe" place so that it may co-reside with another application.
- Although DOS allows most of its commands to be executed from within a BASIC program, additional function is needed. Under DOS there is no way to conveniently read a file directory from a BASIC program, or to save and restore Applesoft's variables, for example. Likewise, the implementation of program `CHAIN`ing is not integrated into DOS.
- Additional functions under DOS which would be desirable (to name only a few) are: a display of the amount of freespace left on a diskette; a way to show the address and length parameters stored with a binary file; and a way to create unbootable data disks to increase storage space for user files.

ENTER PRODOS

In January 1984, Apple introduced a new disk operating system for its Apple II family of computers. ProDOS is intended to replace DOS 3.3 as the standard Apple II operating system, and it is now being shipped with all new Disk II drives instead of DOS. Although, on the surface, ProDOS is very similar in appearance to DOS 3.3, it represents a major redesign and is a new and separate system. From the beginning, ProDOS addresses all of DOS's weaknesses mentioned above:

- ProDOS is up to **eight times faster** than DOS in disk access. A new "direct read" mode has been implemented which allows multisector reads to be performed directly from the disk to the programmer's buffer without multiple buffering within ProDOS itself. When performing direct reads, ProDOS can transfer data from the diskette at a rate of eight kilobytes per second (at best, DOS can read one kilobyte per second). Even when reading small amounts of data from the disk, ProDOS does less multiple buffering than does DOS.
- ProDOS provides a **device independent** interface to "foreign" mass storage devices. The concept of a hierarchically organized disk "volume" was created to allow for large capacity devices, and vectors are provided to allow device drivers for non-standard disks to be integrated into ProDOS. Directories may be dynamically expanded to unlimited size to allow for large numbers of files, and an individual file may now occupy up to 16 million bytes of space on a volume. The largest volume which can be supported is 32 million bytes.
- Device **driver support** has also been provided for calender/clock peripherals, allowing time and date stamping of files, and support for the Apple IIe and IIc 80-column hardware is a part of ProDOS.
- Learning from its mistakes with DOS, Apple has externalized as many ProDOS functions as possible through well defined **system calls**. In addition to standard file management system calls, interfaces are provided to support user written commands to the BASIC interpreter, and to invoke a ProDOS command from within an assembly language program.
- The ProDOS file and volume structure is nearly identical to that of the Apple III SOS operating system. There are even strong similarities between ProDOS system calls and those on Apple's Macintosh! A ProDOS volume may be accessed from SOS directly without the need for a special utility program. ProDOS system calls are a large subset of those offered under SOS and applications may be developed which will easily port between the two operating systems.
- ProDOS defines a protocol which **interrupting** devices may use to coexist harmoniously in the same machine. Up to four interrupt drivers may be installed in ProDOS, and each device need not know that the others exist.
- Most locations of general interest have been placed in externally accessible areas of memory called **global pages**. Through a global page, a user written program can obtain the current ProDOS version number, the most recent values entered on a ProDOS command line, or the configuration of the current hardware including the machine type, memory size, and

the contents of the peripheral card slots. In addition, a voluntary system has been provided to "fence off" portions of memory for special uses by marking a memory bit map in the system global page.

- New support has been provided under ProDOS for BASIC programmers. A BASIC program can now read a directory file, make a "snapshot" of its variables on disk and later restore them, and chain between programs, preserving variables.
- The `CATALOG` command under ProDOS displays the address and length values of binary files as well as the space remaining on a disk volume.

MORE PRODOS ADVANTAGES

In addition to addressing needs which grew out of DOS, Apple has also come up with **other enhancements** with ProDOS:

- A new "smart" `RUN` command ("`-`") has been added which will automatically perform the function of a `RUN`, `EXEC`, or `BRUN` as appropriate depending upon the type of file being `RUN`.
- The assembly language interface has been expanded to include obtaining and updating statistical information about a file, moving the end of file mark in a file, allowing line at a time reads versus byte stream reads, determining the names of diskettes mounted in online drives, and creating new files or directories. In addition, entry points are included to allow applications to pass control from program to program and to allocate memory.
- The language independent, file management portion of ProDOS (the Kernel) is a separate unit from the BASIC support routines. Applications may be written which reclaim the memory normally occupied by BASIC support routines.
- All ProDOS utilities are menu oriented with enhanced user interfaces.
- Owners of the Extended 80-column card in an Apple IIe have access to a 64K "RAM/electronic disk drive" under ProDOS. Data stored there may be accessed almost instantaneously allowing much more efficient loading and storing of programs and data.
- Applesoft string "garbage collection" has been rewritten under ProDOS, and is now many times faster and more efficient.
- Files may be restricted or "locked" by type of access. Read only files may be established, or files which may be written but not destroyed, for example.

- The binary save (`BSAVE`) command has been enhanced under ProDOS. `BSAVE`s into existing binary files whose A and/or L keywords are omitted will use the current values of the target file. Also, other file types besides `BIN` files may be `BLOAD`ed and `BSAVE`d, allowing direct modification at a byte-by-byte level. (For example, one can `BLOAD` a text file and examine it in memory, making modifications to the hex image.)
- The record length of a random access text file is now stored with the file, allowing subsequent BASIC programs to access it without knowing its record length.
- Data disk volumes may now be created which do not contain an image of the operating system. ProDOS makes more efficient use of the disk, resulting in slightly more user storage for files.
- More information about a file is stored in the directory entry under ProDOS than under DOS. The length of a binary or Applesoft file, for example, is stored in the directory, not in the file itself.
- The manner in which the ProDOS BASIC Interpreter intercepts a BASIC program's command lines has been improved and is more reliable. It is now very difficult to "disconnect" ProDOS as could occur under DOS.
- More file types (256) are available under ProDOS. Some are "user definable."

WHAT YOU GIVE UP WITH PRODOS

ProDOS is not for everyone, however. There are a number of **disadvantages** to moving from DOS to ProDOS:

- Most assembly language programs which ran under DOS will have to be rewritten for ProDOS. The file management interfaces are completely different, and the "`PRINT control-D`" mechanism which worked from assembly language under DOS no longer works under ProDOS. This means that most commercial applications, such as word processors, compilers, and spreadsheets, will not be available for ProDOS until they are converted. This state of affairs will change, however, since ProDOS is now the "official" operating system for Apple II computers.
- Apple's older version of BASIC, Integer BASIC, is not supported under ProDOS. Indeed, Applesoft must be in the motherboard ROMs for the ProDOS BASIC Interpreter to work at all. This means that only the ProDOS Kernel, used in a standalone, run-time environment, will run on

an original, Integer Apple II. It is likely that someone (probably not Apple) will soon market an Integer BASIC interpreter for ProDOS, however.

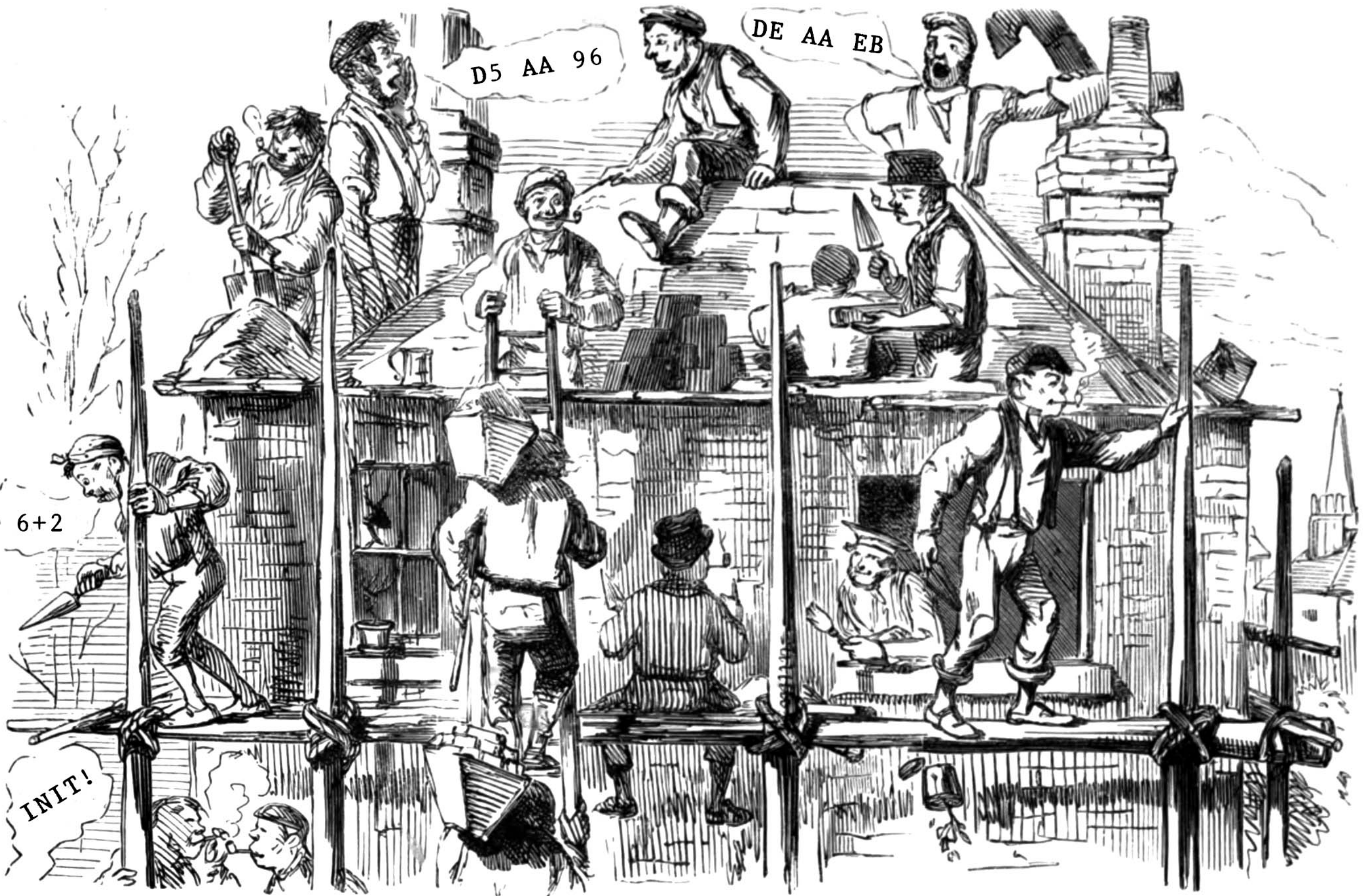
- ProDOS requires 64K to support BASIC programming and commands. It can be made to run in 48K for run-time assembly language applications, but 64K is required to run the BASIC Interpreter which incorporates all of the ProDOS commands (e.g., `CATALOG`, `BLOAD`, etc.).
- Under BASIC, less memory is available to the program. Under DOS, `HIMEM` was set at `$9600` with three file buffers built into DOS. Under ProDOS, `HIMEM` is at `$9600` with no file buffers built in. Thus, as soon as a ProDOS BASIC program opens a file, `HIMEM` is moved down and 1K less memory is available. Likewise. since the Kernel occupies the Language Card (or bank switched memory), this space may not be used for other purposes. (DOS could be relocated into the language card to make more space available to BASIC programs. Also, Applesoft enhancement aid programs typically were loaded into the language card's alternate 4K bank under DOS. This is where ProDOS stores its `QUIT` code now).
- ProDOS only maintains a single directory prefix for all volumes, rather than remembering a default prefix for each volume. Hence, diskette swapping and access to multiple volumes at once can be cumbersome.
- Although the pathname for a file may be 64 characters, the actual name of a file may be only 15 characters, and may not include any special characters or blanks (other than "period"). 30 characters were permitted under DOS.
- Under DOS, up to 16 files may be opened concurrently by a BASIC program. Under ProDOS, only eight files may be opened at once. Also, an open file "cost" 595 bytes under DOS; under ProDOS, a 1024-byte buffer is allocated.
- BASIC programs which are computationally oriented will run about four percent slower on ProDOS than they did under DOS. This is because the ProDOS BASIC Interpreter leaves Applesoft `TRACE` running (invisibly) at all times so that it can monitor the execution of the program and perform garbage collection and disk commands. On the other hand, if strings or disk accesses are used, this degradation of performance will be more than offset by improvements in these areas.
- Several DOS commands have been removed, including `NOMON`, `MON`, and `VERIFY`. There is no way to see the commands in an EXEC file as they are executed.
- If a ProDOS directory is destroyed, it is harder to reconstruct than was the DOS `CATALOG` track. More information is stored in the directory making it harder to identify a file's type by examining its data blocks. Also, since

seedling files do not have index blocks (similar to DOS Track/Sector Lists), they are almost impossible to find once their directory entries are gone.

OTHER DIFFERENCES BETWEEN PRODOS AND DOS

There are a few other minor differences between ProDOS and DOS which are worth noting:

- The `BRUN` command now calls the target program rather than jumping to it as did DOS. The invoked program may return to ProDOS via a return subroutine.
- `CLOSE` will not produce an error message if the file named is not currently open.
- `APPEND` implies `WRITE`. It is not necessary to follow an `APPEND` command with a `WRITE` command in a BASIC program.
- ASCII text in ProDOS directory entries or `TXT` files is stored with the most significant bit off.



The original disk formatters

Chapter 3

Disk II Hardware and Diskette Formatting

This chapter will explain how data is stored on a floppy diskette using a disk drive (Disk II family or equivalent). Much of the information in this chapter is applicable not only to ProDOS, but also to other operating systems on the Apple computer (DOS, Pascal, CP/M). Because ProDOS isolates device specific code, the contents of this chapter should not be considered a prerequisite for understanding succeeding chapters.

For system housekeeping, ProDOS divides external storage devices into blocks. Each **block** contains 512 bytes of information. It is device independent in that each device has its own driver. This driver enables ProDOS to read and write blocks, and additionally to obtain the status of a device. The device itself may actually store information in a number of ways and not necessarily in blocks. Blocks can be thought of as a conceptual unit of data that was created in software, having little or no relation to how data is actually stored on an external storage device. In fact, the standard Disk II stores information in a track and sector format. The device driver provides a mapping between these tracks and sectors, and the blocks. Since a sector contains 256 bytes, two **sectors** are required for each block. There are 560 sectors on a diskette and therefore 280 blocks. Chapter 4 deals with how ProDOS allocates these blocks to create files.

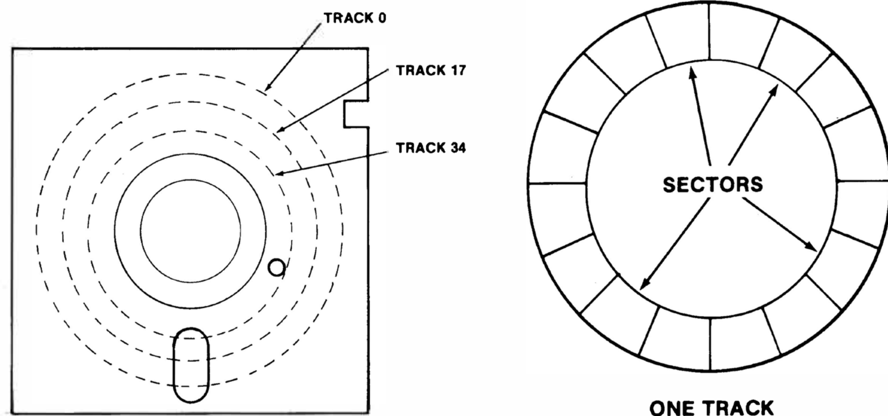


Figure 3.1 Tracks and Sectors

TRACKS AND SECTORS

As stated above, a diskette is divided into tracks and sectors. This is done during the initialization or formatting process. A **track** is a physically defined circular path which is concentric with the hole in the center of the diskette. Each track is identified by its distance from the center of the disk. Similar to a phonograph stylus, the read/write head of the disk drive may be positioned over any given track. The tracks are similar to the grooves in a record, but they are not connected in a spiral. Much like playing a record, the diskette is spun at a constant speed while the data is read from or written to its surface with the read/write head. Apple formats its diskettes into 35 tracks, numbered from 0 to 34, track 0 being the outermost track and track 34 the innermost. Figure 3.1 illustrates the concept of tracks, although they are invisible to the eye on a real diskette.

It should be pointed out, for the sake of accuracy, that the disk arm can position itself over 70 distinct locations or **phases**. To move the arm from one track to the next, two phases of the stepper motor which moves the arm must be cycled. This implies that data might be stored on 70 tracks, rather than 35. Unfortunately, the resolution of the read/write head is such that attempts to use these phantom **half** tracks create so much cross-talk that data is lost or overwritten. Although standard ProDOS uses only full tracks (even phases), some copy protected disks use half tracks (odd phases) or combinations of the two. This will work provided that no data is closer than two phases from other data. See Appendix B for more information on copy protection schemes.

A sector is a subdivision of a track. It is the smallest unit of "updatable" data on the diskette. While ProDOS reads or writes data a block at a time (two sectors), the device driver operates on one sector at a time. This allows the device driver to use only a small portion of memory as a buffer during read or write operations. Apple has used two different track formats to date. The initial operating system divided the track into 13 sectors but all recent operating systems use 16 sectors. The sectoring does not use the index hole, provided on most diskettes, to locate the first sector of the track. The implication is that the software must be able to locate any given track and sector with no help from the hardware. This scheme, known as **soft sectoring**, takes a little more space for storage but allows flexibility, as evidenced by the previous change from 13 sectors to 16 sectors per track. The following table categorizes the amount of data stored on a diskette under ProDOS. Both system and data diskettes are categorized.

DISKETTE ORGANIZATION

Tracks	35
Sectors Per Track	16
Sectors Per Diskette	560
Bytes Per Sector	256
Bytes Per Diskette	143,360
Usable* Blocks For Data Storage	
ProDOS System Diskette	221
ProDOS Data Diskette	273
Usable* Bytes Per Diskette	
ProDOS System Diskette	113,152
ProDOS Data Diskette	139,776

*System Diskette includes ProDOS and BASIC.SYSTEM files only.

TRACK FORMATTING

Up to this point we have have broken down the structure of data to the track and sector level. To better understand how data is stored and retrieved, we will start at the bottom and work up.

As this manual is about software (ProDOS), we will deal primarily with the function of hardware rather than explain how it performs that function. For example, while data is in fact stored as a continuous stream of analog signals, we will deal with discrete **digital** data, i.e., a "`Ø`" or a "`1`". We recognize that the hardware converts analog data to digital data, but how this is accomplished is beyond the scope of this manual. For a full and detailed explanation of the hardware, please refer to Jim Sather's excellent book, *Understanding the Apple II*, published by Quality Software.

Data bits are recorded on the diskette in precise intervals. The hardware recognizes each of these intervals as either a "`Ø`" or a "`1`". We will define these intervals to be **bit cells**. A bit cell can be thought of as the distance the diskette moves in four machine cycles, which is about four microseconds. Using this representation, data written on and read back from the diskette takes the form shown in Figure 3.2. The data pattern shown represents a binary value of `1Ø1`.

A byte as recorded on the disk consists of eight (8) consecutive bit cells. The **most significant bit** cell is usually referred to as bit cell 7, and the **least significant** is **bit** cell 0. When reference is made to a specific data bit (i.e. data bit 5), it is with respect to the corresponding bit cell (bit cell 5). Data is written and read serially, one bit at a time. Thus, during a write operation, bit cell 7 of

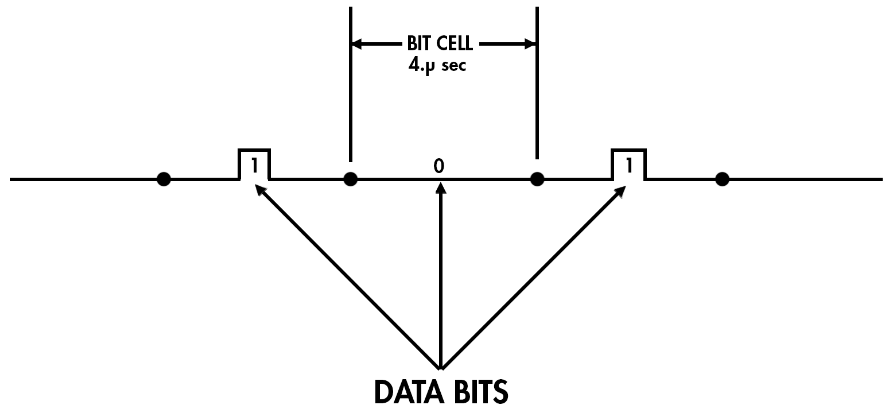


Figure 3.2 Bits on Diskette

each byte is written first, and bit cell 0 is written last. Correspondingly, when data is being read back from the diskette, bit cell 7 is read first and bit cell 0 is read last. Figure 3.3 illustrates the relationship of the bits within a byte.

To graphically show how bits are stored and retrieved, we must take certain liberties. The diagrams are a representation of what functionally occurs within the disk drive. For the purposes of our presentation, the hardware interface to the diskette will be represented as an 8-bit **data register**. Since the hardware involved considerably more complication, from a software standpoint, it is reasonable to use the data register, as it accurately embodies the function of data flow to and from the diskette. For a further discussion of the hardware, please see Appendix D.

Figure 3.4 shows the three bits, `1Ø1`, being read from the diskette data

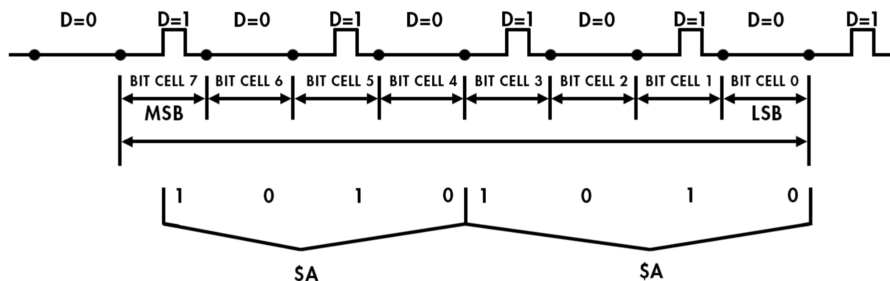


Figure 3.3 One Byte on Diskette

stream into the data register. Of course, another five bits would be read to fill the register.

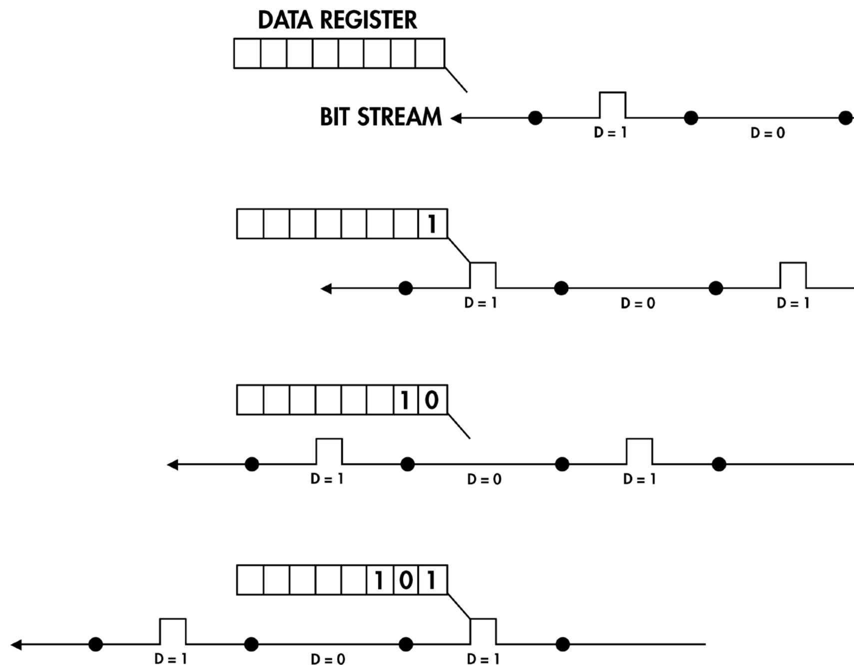


Figure 3.4 Reading Data from a Diskette

Writing data can be depicted much in the same way (See Figure 3.5). It should be noted that, while in write mode, zeroes are being brought into the data register to replace the data being written. It is the task of the software to make sure that the register is loaded and instructed to write on 32-cycle (microsecond) intervals. If not, zero bits will continue to be written every four cycles, which is in fact exactly how self-sync bytes are created. Self sync bytes will be covered in detail shortly.

A **field** is made up of a group of consecutive bytes. The number of bytes varies, depending upon the nature of the field. The two types of fields present on a diskette are the **Address Field** and the **Data Field**. They are similar in that they both contain a prologue, a data area, a checksum, and an epilogue. Each field on track is separated from adjacent fields by a number of bytes. These areas of separation are called **gaps** and are provided for two reasons. First, they allow the updating of one field without affecting adjacent fields (on the Apple, only data fields are updated). Secondly, they allow the computer time to decode the address field before the corresponding data field can pass beneath the read/write head.

All gaps are primarily alike in content, consisting of self-sync hexadecimal `FF`s and vary only in the number of bytes they contain. Figure 3.6 is a diagram of a portion of a typical track, broken into its major components.

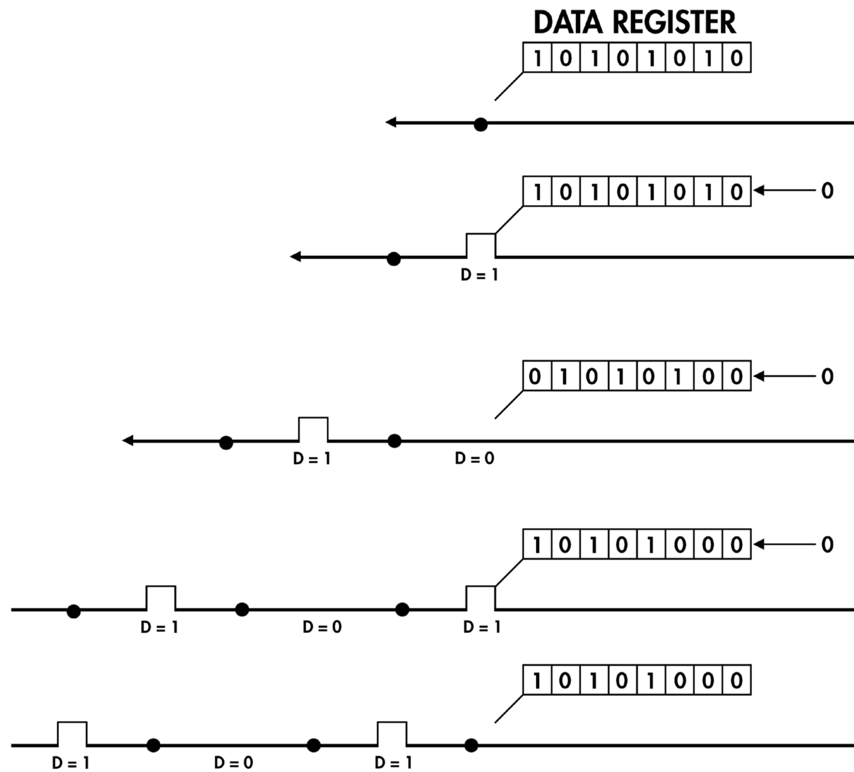


Figure 3.5 Writing Data to a Diskette

Self sync or auto-sync bytes are special bytes that make up the three different types of gaps on a track. They are so named because of their ability to automatically bring the hardware into synchronization with data bytes on the disk. The difficulty in doing this lies in the fact that the hardware reads bits, and the data must be stored as 8-bit bytes. It has been mentioned that a track is literally a continuous stream of data bits. In fact, at the bit level, there is no way to determine where a byte starts or ends, because each bit cell is exactly the same, written in precise intervals with its neighbors. When the drive is instructed to read data, it will start wherever it happens to be on a particular track. That could be anywhere among the 50,000 or so bits on a track. The hardware finds the first bit cell with data in it and proceeds to read the following seven bits of data into the 8-bit register. In effect, it assumes that it has started at the beginning of a data byte. Of course, in reality, it could have started at any of the "1" bits of the byte. Pictured in Figure 3.7 is a small portion of a track.

From looking at the data, there is no way to tell what bytes are represented, because we don't know where to start. This is exactly the problem that self-sync bytes overcome.

TRACK FORMAT

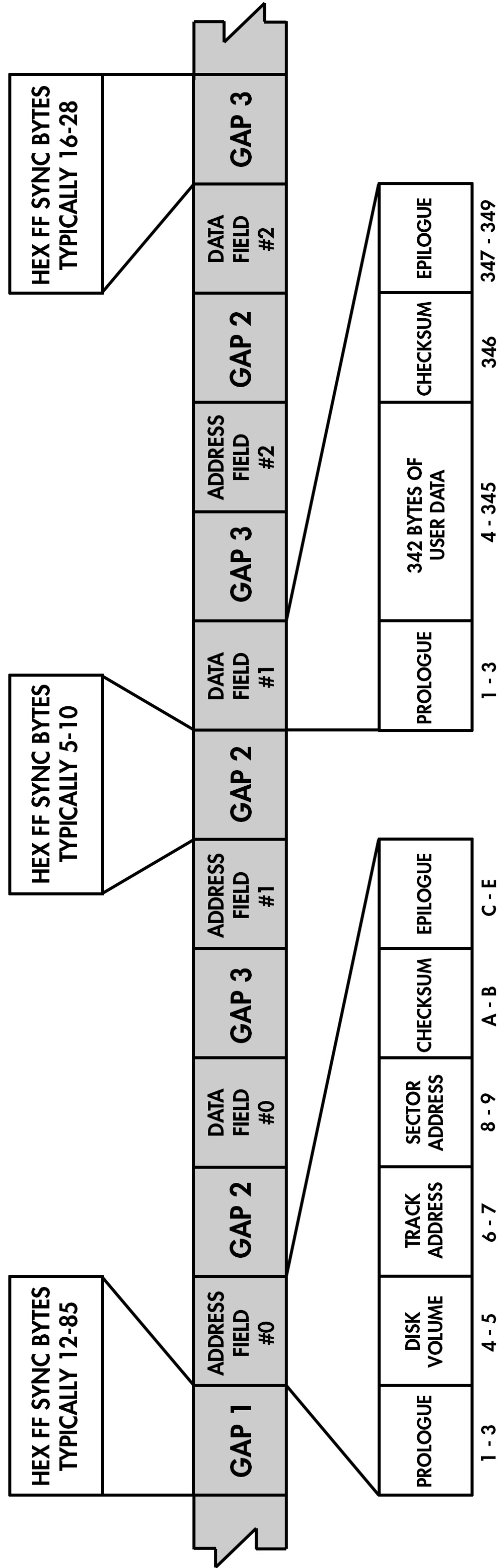


Figure 3.6 Track Format

A **self-sync** byte is defined to be a hexadecimal `FF` with a special difference. It is, in fact, a 10-bit byte rather than an 8-bit byte. Its two extra bits are zeroes. Figure 3.8 shows the difference between a normal data hex `FF` that might be found elsewhere on the disk and a self-sync hex `FF` byte.

A self-sync byte is generated by using a 40-cycle (microsecond) loop while writing an `FF`. A bit is written every four cycles, so two of the zero bits brought into the data register while the `FF` was being written are also written to the disk, making the 10-bit byte. It can be shown, using Figure 3.9, that four self-sync bytes are sufficient to guarantee that the hardware is reading valid data. The reason for this is that the hardware requires the first bit of a byte to be a "`1`". Pictured at the top of the figure is a stream of four self-sync bytes followed by a normal `FF`. Each row below that demonstrates what the hardware will read should it start reading at any given bit in the first byte. In each case, by the time the four sync bytes have passed beneath the read/write head, the hardware will be synced to read the data bytes that follow. As long as the disk is left in read mode, it will continue to correctly interpret the data unless there is an error on the track.

```
Ø 1 1 Ø 1 Ø 1 1 1 Ø 1 Ø 1 1 Ø Ø 1 1 1 1 Ø 1 1 Ø 1 1 1 Ø 1 Ø 1
```

Figure 3.7 An Example Bit Stream on a Diskette

We can now discuss the particular portions of a track in detail. The three gaps will be covered first. Unlike some other disk formats, the size of the three gap types will vary from drive to drive and even from track to track. During the formatting process, ProDOS will start with large gaps and keep making them smaller until an entire track can be written without overlapping itself. A minimum number of self-sync bytes is maintained for each gap type. The result is fairly uniform gap sizes within each particular track.

Gap 1 is the first data written to a track during initialization. Its purpose is twofold. The gap originally consists of 128 self-sync bytes, a large enough area to insure that all portions of a track will contain data. Since the speed of a particular drive may vary, the total length of the track in bytes is uncertain, and the percentage occupied by data is unknown. The initialization process is set up, however, so that even on drives of differing speeds, the last data field written will overlap Gap 1, providing continuity over the entire physical track.

Figure 3.8 Comparison of Normal Byte and Self-Sync Byte

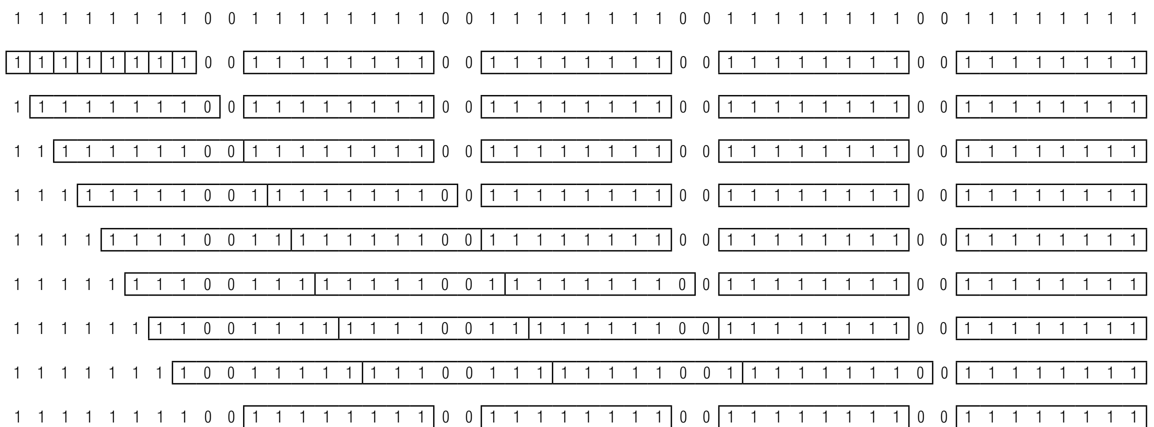


Figure 3.9 Self-Sync Bytes

Unlike earlier operating systems, ProDOS will let you know if your drive is too fast or too slow. The remaining portion of Gap 1 must be approximately 75% as long as a Gap 3 on that track, enabling it to serve as a Gap 3 type for Address Field number 0 (See Figure 3.6 for clarity).

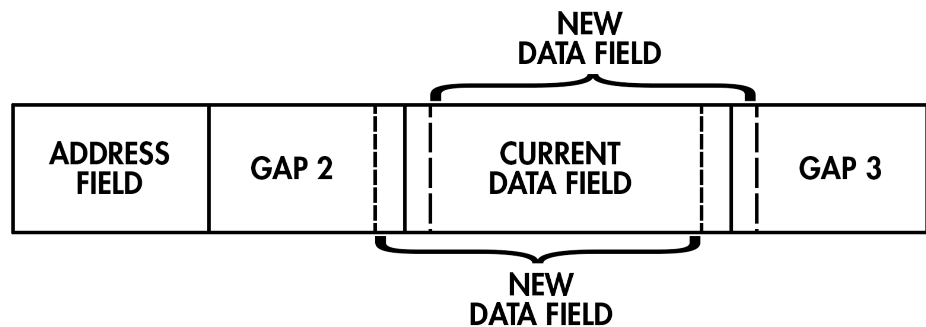


Figure 3.10 ProDOS Doesn't Always Write in the Same Place

Gap 2 appears after each Address Field and before each Data Field. Its primary purpose is to provide time for the information in an Address Field to be decoded by the computer before a read or write takes place. If the gap was too short, the beginning of the Data Field might spin past while ProDOS was still determining if this was the sector to be read. The 200 cycles that five self-sync bytes provide seems ample time to decode an Address Field. When a Data Field is written, there is no guarantee that the write will occur in exactly the same spot each time. This is due to the fact that the drive which is rewriting the Data Field may not be the one which originally formatted or wrote it. Since the speed of the drives can vary, it is possible that the write could start in mid-

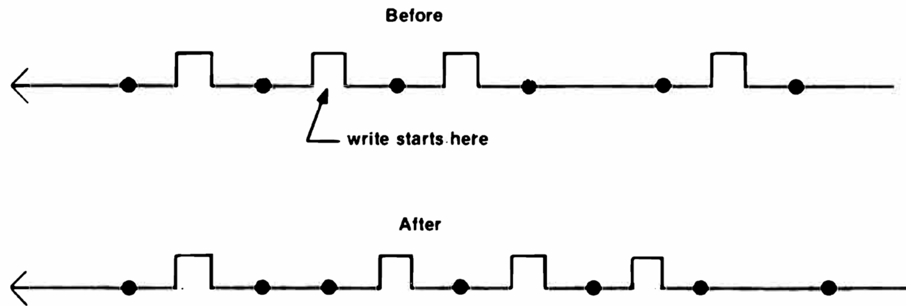


Figure 3.11 Writing Out of Sync

byte (see Figure 3.10). For this reason, the length of Gap 2 varies from five to ten bytes. This is not a problem as long as the difference in positioning is not too great. To insure the integrity of Gap 2 when writing a data field, five self-sync bytes are written prior to writing the Data Field itself. This serves two purposes. Since relatively little time is spent decoding an address field, the five bytes help place the Data Field near its original position. Secondly, and more importantly, the five self-sync bytes are the minimum number required to guarantee read-synchronization. It is probable that, in writing a Data Field, at least one sync byte will be destroyed. This is because, just as in reading bits on the track, the write may not begin on a byte boundary, thus altering an existing byte. Figure 3.11 illustrates this.

Gap 3 appears after each Data Field and before each Address Field. It is longer than Gap 2 and care is taken to make sure it ranges from 16 to 28 bytes in length. It is quite similar in purpose to Gap 2. Gap 3 allows the additional time needed to manipulate the data that has been read before the next sector is to be read. The length of Gap 3 is not as critical as that of Gap 2. If the following Address Field is missed, ProDOS can always wait for the next time it spins around under the read/write head (one revolution of the disk at most). Since Address Fields are never rewritten, there is no problem with Gap 3 providing synchronization, since only the first part of the gap can be overwritten or damaged (see Figure 3.10 for clarity).

ADDRESS FIELDS

An examination of the contents of the two types of field is in order. The Address Field contains the address or identifying information about the Data Field which follows it. The volume, track, and sector number of any given sector can be thought of as its "address," much like a country, city, and street number might identify a house. As shown previously in Figure 3.6, there are

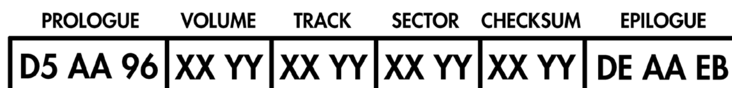


Figure 3.12 Address Field

a number of components which make up the Address Field. A more detailed illustration is given in Figure 3.12.

Each byte of the Address Field is encoded into two bytes when written to the disk. Appendix C describes the "4 and 4" method used for Address Field encoding.

The **prologue** consists of three bytes which form a unique sequence, found in no other component of the track. This fact enables ProDOS to locate an Address Field with almost no possibility of error. The three bytes are `$D5`, `$AA`, and `$96`. The `$D5` and `$AA` are reserved (never written as data), thus insuring the uniqueness of the prologue. The `$96`, following this unique string, indicates that the data following constitutes an Address Field (as opposed to a Data Field). The address information follows next, consisting of the **volume***, **track**, and **sector** number and a checksum. This information is absolutely essential for ProDOS to know where it is positioned on a particular diskette. The **checksum** is computed by exclusive-ORing the first three pieces of information, and is used to verify its integrity. Lastly follows the **epilogue**, which contains the three bytes `$DE`, `$AA`, and `$EB`. The `$EB` is only partly written during initialization, and is therefore never verified when an Address Field is read. The epilogue bytes are sometimes referred to as **bit-slip marks**, which provide added assurance that the drive is still in sync with the bytes on the disk. These bytes are probably unnecessary, but do provide a means of double checking.

DATA FIELDS

The other field type is the Data Field. Much like the Address Field, it consists of a **prologue**, **data**, **checksum**, and an **epilogue** (refer to Figure 3.13). The Prologue differs only in the third byte. The bytes are `$D5`, `$AA`, and `$AD`, which again form a unique sequence, enabling ProDOS to locate the beginning of the sector data. The data consists of 342 bytes of encoded data. (The encoding scheme used is quite complex and is documented in detail in Appendix C.) The data is followed by a checksum byte, used to verify the integrity of the data

* Volume number is a leftover from earlier operating systems and is not used by ProDOS.

just read. The epilogue portion of the Data Field is absolutely identical to the epilogue in the Address Field and serves the same function.

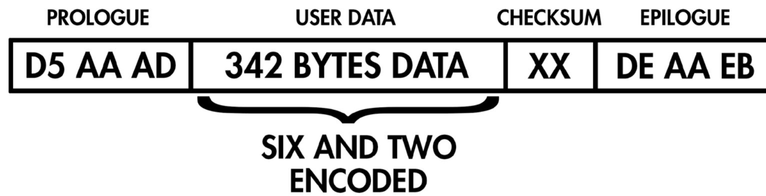


Figure 3.13 Data Field

DISK II BLOCK AND SECTOR INTERLEAVING

Because the Disk II is such an integral part of the Apple II family of machines, it is important that it perform efficiently. One major factor in disk drive performance is how the data is arranged on the diskette. Because the diskette spins and the head that reads and writes the data is stationary, it is necessary to wait for a particular portion of a given track to pass by. This waiting (rotational delay) can add significant time to a disk access if the data is poorly arranged. **Interleaving** (or skewing) is the arranging of data at the block or sector level to maximize access speed. It effectively places a gap between blocks or sectors that will normally be accessed sequentially, allowing sufficient time for internal housekeeping before the next one appears. In general, if blocks or sectors are poorly arranged on a track, it is usually necessary to wait an entire revolution of the diskette before the next desired block or sector can be accessed.

The first versions of Apple's operating system used **physical interleaving** on the disk. (That is, sectors were written in a particular order on the diskette.) A

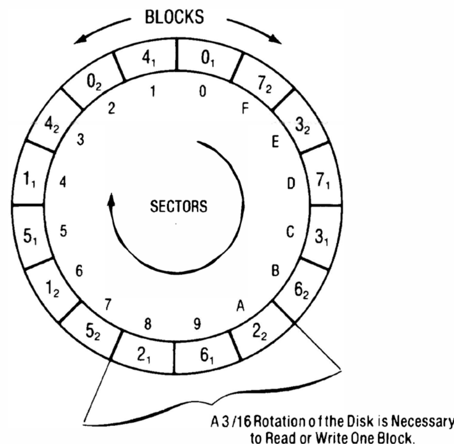


Figure 3.14 Block Interleaving (Track 0)

number of different schemes were used in an attempt to maximize performance. This worked reasonably well, but because different methods were used for different operations, performance suffered. Later versions standardized the physical interleaving (as sequential), and used a software method to try to maximize performance. An attempt was also made to standardize some operations, but performance still was not optimal as evidenced by a proliferation of "fast" DOSs.

ProDOS provides an impressive improvement over Apple's earlier operating systems. Several factors account for the dramatic improvement. The routine to read data is significantly faster, minimizing the delay occurring between read operations. The data is dealt with in larger pieces (512 bytes vs. 256 bytes), lowering the number of requests to the code that actually reads and writes data (Device Driver). And almost all operations involve files stored on sequential blocks. As a disk begins to get full, this will not always be possible and some file will be discontinuous; but for the most part, all operations (loading ProDOS or Applesoft BASIC, reading or writing to files or a directory) involve data in contiguous pieces. This greatly simplifies the problem of finding an optimal interleaving for disk accesses.

In ProDOS, the interleaving is done in software. The 16 sectors are in numerically ascending order on the diskette (0, 1, 2, . . . 15), and are not physically interleaved at all. An algorithm is used to translate block numbers into physical sector numbers used by the ProDOS device driver. For example, if the block number requested were 2, this would be translated to track 0, physical sectors 8 and A*. Figure 3.14 illustrates the concept of **software interleaving** and Table 3.1 shows the mapping of physical sectors to blocks for a Disk II or compatible drive.

There are two kinds of interleaving to consider in the case of ProDOS. First, there is the interleaving of the two sectors that make up a block. This will be referred to as **intra-block** or "within block" interleaving. Second, there is the interleaving between blocks on a given track. This will be referred to as **inter-block** or "between block" interleaving. It should be noted that we are concerned primarily with delays within ProDOS and the Disk II Device Driver, and not with delays that may be present in various application packages.

* Those familiar with DOS 3.3 should note that physical sector numbers and DOS 3.3 sector numbers are not the same. Most "disk utilities" use DOS 3.3 sector numbers and not physical sector numbers. The bottom of Table 3.1 shows how DOS 3.3 sector numbers are related to ProDOS block numbers.

INTRA-BLOCK INTERLEAVING

When ProDOS accesses a block, it must of course access the two sectors that make up that block. There is a small delay after the device driver has accessed the first sector, before it can access the second sector. This delay is different for Read and Write operations. The Read operation is so fast that the disk can read two sectors in a row. However, the Write operation takes longer, so for optimal performance there must be a gap between the two sectors that make up a block. If there wasn't a gap, an entire revolution of the diskette would be required for each block written. A single sector provides a sufficient gap, so intra-block interleaving (within the block) consists of **one sector**. The result is that ProDOS is able to write to a given block as rapidly as is possible. Some time is lost when reading a block, but no other interleaving scheme would provide the same overall efficiency. Intra-block interleaving is illustrated in Figure 3.15.

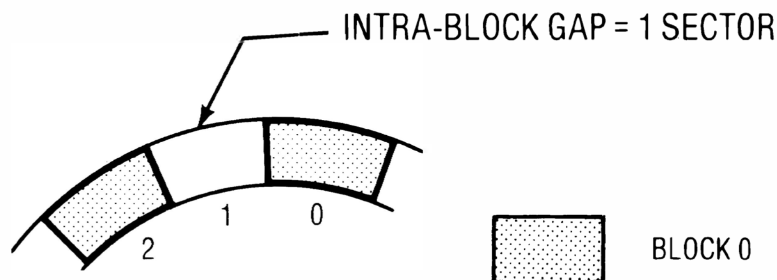


Figure 3.15 Intra-Block Interleaving (Within Block)

INTER-BLOCK INTERLEAVING

When ProDOS accesses a number of blocks as required in most disk operations (i.e. reading or writing a directory or a file), another kind of interleaving is involved. There will be a delay between accesses, but it is now between blocks rather than sectors. There is relatively little difference in delay time in the MLI itself between reading and writing—almost all the difference occurs in the device driver. However, when ProDOS writes a block that is already allocated (i.e. part of an existing directory or file), it always reads that block before writing to it. This requires an entire revolution of the diskette regardless of how the interleaving is done. It turns out that, just as for intra-block operations, a **single sector** is a sufficient gap for reading blocks. Inter-block interleaving is illustrated in Figure 3.16.

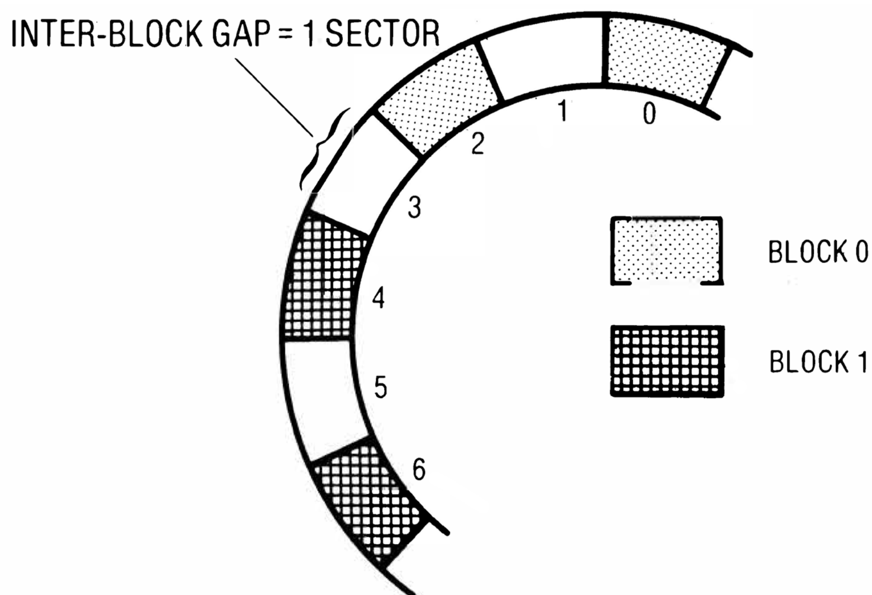


Figure 3.16 Inter-Block Interleaving (Between Block)

READING OR WRITING A BLOCK

Assume that we wish to access block 2. ProDOS passes the request to the device driver which in turn converts the block number into its track and sector representation (see Figure 3.14). The arm is moved to the proper track (0) and then a sector is read. This could be any sector, because the diskette is spinning. Sectors are continually read until sector 8 is found. The following two sectors are then read (9 and A) which completes the read of block 2 (sectors 8 and A). Depending on where we start on the track, we could read between 3 and 18 sectors. The same process occurs when writing a single block, with one small difference. After sector 8 is located and written to, the delay required to ready the data for sector A will cause us to miss reading sector 9. This does not alter the amount of rotation necessary to complete the task. To summarize, the time required to either read or write a single block consists of two factors. (We are assuming the track has already been located). First, there is the time required to **locate** the first sector of the block—this is variable and ranges between 0 and the time of one full rotation of the diskette. Second is the time required to actually **read or write** the two sectors that make up the block—this is fixed and always requires 3/16 rotation of the diskette.

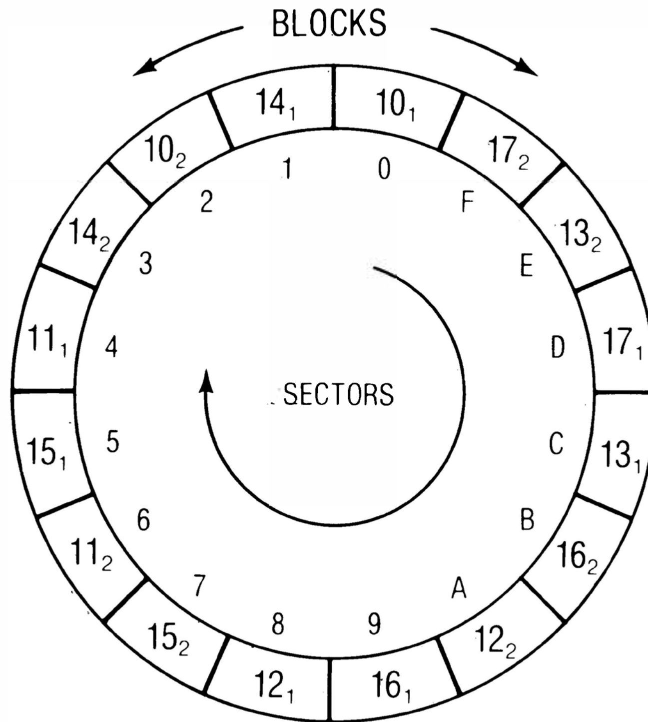


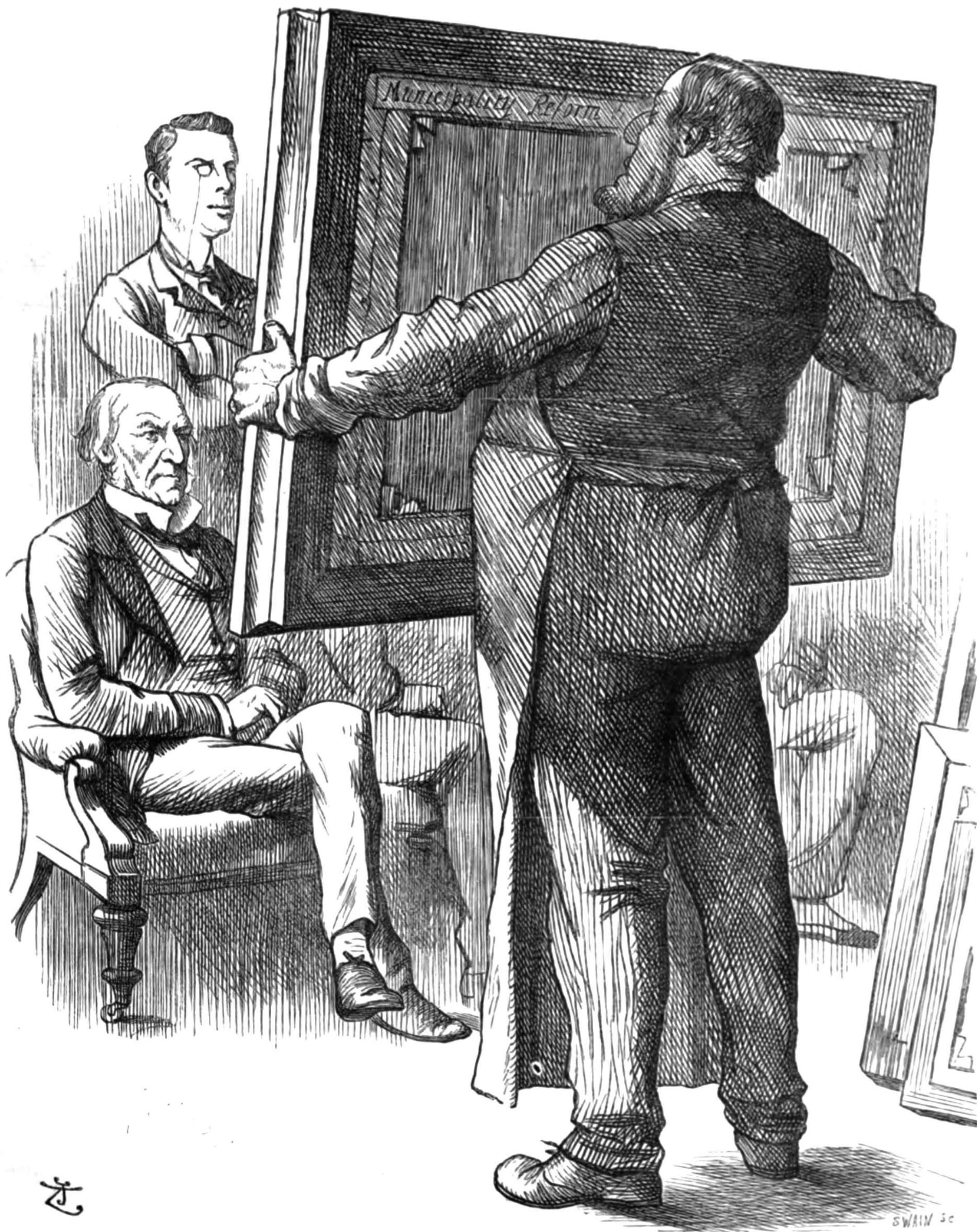
Figure 3.17 Example of Block Interleaving of Track 2

READING OR WRITING CONSECUTIVE BLOCKS

Let's examine what occurs when a number of blocks are accessed during reading or writing of a typical file. We will assume the file is reasonably large and takes up a number of blocks. We will confine our observation to a single track in which eight blocks comprise the file of interest. We will assume track 2, which contains blocks 10 through 17 (as in Figure 3.17), and we will further assume that the blocks will be accessed sequentially. When the read/write head moves to track 2, we will start reading sectors until the appropriate sector is found (0 in this case). Then each sector is read until all eight blocks are found. This will require exactly **two revolutions** of the disk. Writing takes significantly longer because each block is read before being written to. Therefore, once the first sector of the block in question is located, one entire revolution is necessary to write each block. Upon writing a block, ProDOS is able to locate the next block immediately, read it, wait one revolution, and write it. A total of **ten revolutions** is required to write an entire track as opposed to two revolutions to read it.

Table 3.1 ProDOS Block Conversion Table for Diskettes

	PHYSICAL SECTOR							
	Ø&2	4&6	8&A	C&E	1&3	5&7	9&B	D&F
TRACK ØØ	ØØØ	ØØ1	ØØ2	ØØ3	ØØ4	ØØ5	ØØ6	ØØ7
TRACK Ø1	ØØ8	ØØ9	ØØA	ØØB	ØØC	ØØD	ØØE	ØØF
TRACK Ø2	Ø1Ø	Ø11	Ø12	Ø13	Ø14	Ø15	Ø16	Ø17
TRACK Ø3	Ø18	Ø19	Ø1A	Ø1B	Ø1C	Ø1D	Ø1E	Ø1F
TRACK Ø4	Ø2Ø	Ø21	Ø22	Ø23	Ø24	Ø25	Ø26	Ø27
TRACK Ø5	Ø28	Ø29	Ø2A	Ø2B	Ø2C	Ø2D	Ø2E	Ø2F
TRACK Ø6	Ø3Ø	Ø31	Ø32	Ø33	Ø34	Ø35	Ø36	Ø37
TRACK Ø7	Ø38	Ø39	Ø3A	Ø3B	Ø3C	Ø3D	Ø3E	Ø3F
TRACK Ø8	Ø4Ø	Ø41	Ø42	Ø43	Ø44	Ø45	Ø46	Ø47
TRACK Ø9	Ø48	Ø49	Ø4A	Ø4B	Ø4C	Ø4D	Ø4E	Ø4F
TRACK ØA	Ø5Ø	Ø51	Ø52	Ø53	Ø54	Ø55	Ø56	Ø57
TRACK ØB	Ø58	Ø59	Ø5A	Ø5B	Ø5C	Ø5D	Ø5E	Ø5F
TRACK ØC	Ø6Ø	Ø61	Ø62	Ø63	Ø64	Ø65	Ø66	Ø67
TRACK ØD	Ø68	Ø69	Ø6A	Ø6B	Ø6C	Ø6D	Ø6E	Ø6F
TRACK ØE	Ø7Ø	Ø71	Ø72	Ø73	Ø74	Ø75	Ø76	Ø77
TRACK ØF	Ø78	Ø79	Ø7A	Ø7B	Ø7C	Ø7D	Ø7E	Ø7F
TRACK 1Ø	Ø8Ø	Ø81	Ø82	Ø83	Ø84	Ø85	Ø86	Ø87
TRACK 11	Ø88	Ø89	Ø8A	Ø8B	Ø8C	Ø8D	Ø8E	Ø8F
TRACK 12	Ø9Ø	Ø91	Ø92	Ø93	Ø94	Ø95	Ø96	Ø97
TRACK 13	Ø98	Ø99	Ø9A	Ø9B	Ø9C	Ø9D	Ø9E	Ø9F
TRACK 14	ØAØ	ØA1	ØA2	ØA3	ØA4	ØA5	ØA6	ØA7
TRACK 15	ØA8	ØA9	ØAA	ØAB	ØAC	ØAD	ØAE	ØAF
TRACK 16	ØBØ	ØB1	ØB2	ØB3	ØB4	ØB5	ØB6	ØB7
TRACK 17	ØB8	ØB9	ØBA	ØBB	ØBC	ØBD	ØBE	ØBF
TRACK 18	ØCØ	ØC1	ØC2	ØC3	ØC4	ØC5	ØC6	ØC7
TRACK 19	ØC8	ØC9	ØCA	ØCB	ØCC	ØCD	ØCE	ØCF
TRACK 1A	ØDØ	ØD1	ØD2	ØD3	ØD4	ØD5	ØD6	ØD7
TRACK 1B	ØD8	ØD9	ØDA	ØDB	ØDC	ØDD	ØDE	ØDF
TRACK 1C	ØEØ	ØE1	ØE2	ØE3	ØE4	ØE5	ØE6	ØE7
TRACK 1D	ØE8	ØE9	ØEA	ØEB	ØEC	ØED	ØEE	ØEF
TRACK 1E	ØFØ	ØF1	ØF2	ØF3	ØF4	ØF5	ØF6	ØF7
TRACK 1F	ØF8	ØF9	ØFA	ØFB	ØFC	ØFD	ØFE	ØFF
TRACK 2Ø	1ØØ	1Ø1	1Ø2	1Ø3	1Ø4	1Ø5	1Ø6	1Ø7
TRACK 21	1Ø8	1Ø9	1ØA	1ØB	1ØC	1ØD	1ØE	1ØF
TRACK 22	11Ø	111	112	113	114	115	116	117



HGR to the MAX

Chapter **4**

Volumes, Directories, and Files

As was described in Chapter 3, a 16-sector diskette consists of 560 data areas of 256 bytes each, called sectors. These sectors are arranged on the diskette in 35 concentric rings, called tracks, of 16 sectors each. The way ProDOS allocates these tracks of sectors is the subject of this chapter.

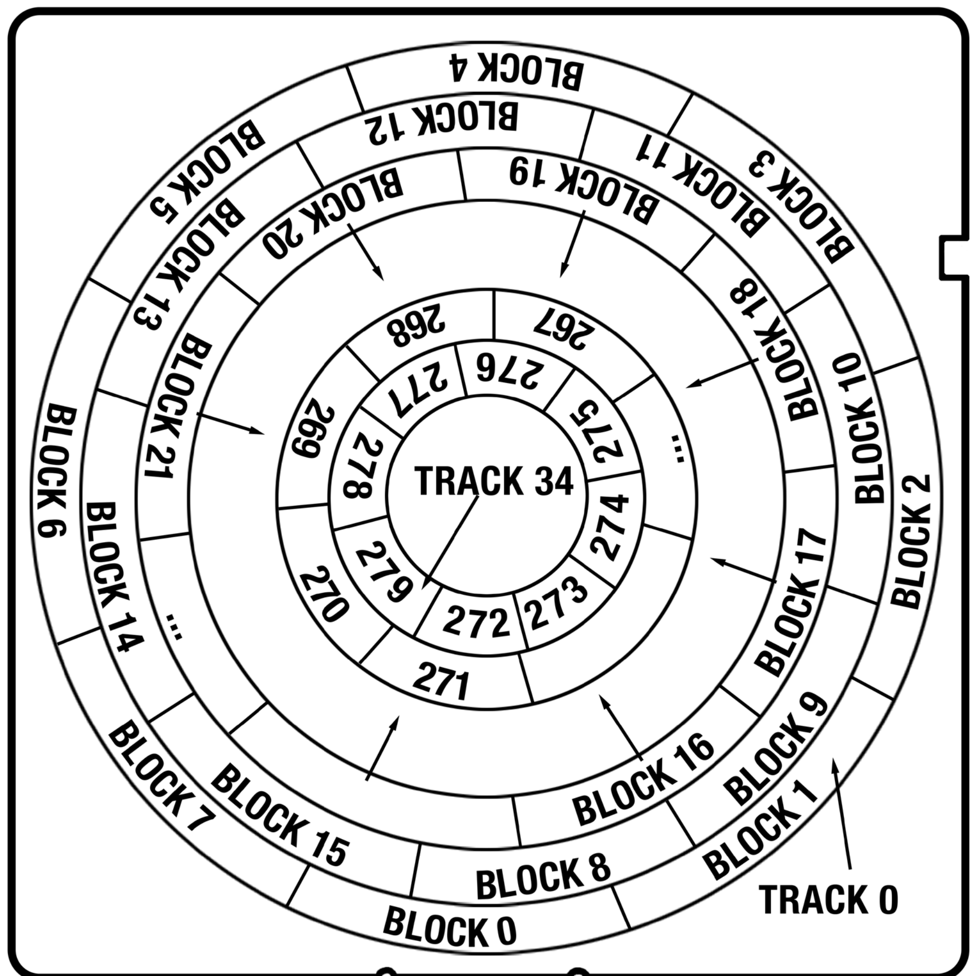


Figure 4.1 Blocks on a Diskette

THE DISKETTE VOLUME

ProDOS defines a **volume** to be any (usually direct access) individual mass storage media. The discussion which follows assumes this media to be a single 35-track diskette, but all of the structures presented here are identical for other diskette sizes and even for a hard disk such as the Apple ProFile. Another interesting point is that the structure of a ProDOS volume is almost identical to that of an Apple III SOS volume. This fact allows greater data compatibility between the two operating systems.

To make the allocation of sectors more manageable, ProDOS pairs them up to form 512-byte **blocks**. Since there are 16 sectors per track and 560 sectors per diskette volume, there are eight blocks per track and 280 blocks per volume. These blocks are numbered from 0 to 279 (decimal) or `$ØØØØ` to `$Ø117` (hexadecimal). The arrangement of blocks on a diskette is shown in Figure 4.1. Of course, on a real diskette, skewing (discussed in Chapter 3) would reorder the blocks on any given track, but, for the purposes of this discussion, the blocks can be assumed to be stored sequentially.

A file, be it `BAS`, `BIN`, `TXT`, or `SYS` type, consists of one or more blocks containing data. Since a block is the smallest unit of allocatable space on a ProDOS volume, a file will use up at least one block even if it is less than 512 bytes long; the remainder of the block is wasted. Thus, a file containing 600 characters (or bytes) of data will occupy one entire block and 88 bytes of another with 424 bytes wasted. Knowing that there are 280 blocks on a diskette, one might expect to be able to use up to 280 times 512 or 143,360 bytes of space on a diskette for files. Actually, the largest file that can be stored in 271

```
/EXAMPLE

 NAME           TYPE  BLOCKS  MODIFIED          CREATED          ENDFILE SUBTYPE

 BASFILE         BAS       1  <NO DATE>         <NO DATE>            109
 TXTFILE         TXT       1  <NO DATE>         <NO DATE>              9 R=   64
 BINFILE         BIN       1  <NO DATE>         <NO DATE>             48 A=$03D0

BLOCKS FREE:  270     BLOCKS USED:   10     TOTAL BLOCKS:  280
```

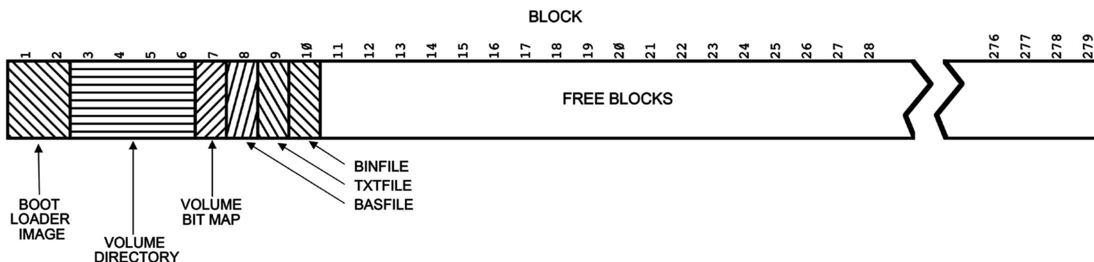


Figure 4.2 Block Usage on an Example Diskette

blocks long (or 138,752 bytes). The reason for this is that some of the blocks on the diskette volume must be used for what is called **overhead**.

VOLUME OVERHEAD

Overhead blocks contain the image of the ProDOS bootstrap loader (which is loaded by the ROM on your diskette controller card and, in turn, loads the ProDOS system files into memory), a list of file names and locations of the files on the diskette, and an accounting of the blocks which are free for use by new files or for expansions of existing ones. An example of the way ProDOS uses blocks is given in Figure 4.2.

Notice that in the case of this diskette volume, system overhead (that part of the diskette which does not actually contain files) falls entirely on track 0 of the diskette (blocks 0 through 7). In fact, there is room for one block's worth of file data on track 0 (block 7). The first block (block 0) is always devoted to the image of the bootstrap loader. (Block 1 is the SOS bootstrap loader.) Following these, and always starting at block 2, is the **Volume Directory**. The Volume Directory is the "anchor" of the entire volume. On any diskette (or hard disk for that matter) for any version of ProDOS, the first or "key" block of the Volume Directory is always in the same place—block 2. Since files can end up anywhere on the diskette, it is through the Volume Directory key block that ProDOS is able to find them. Thus, just as the card catalog is used to locate a book in a library, the Volume Directory is the master index to all of the files on a volume. In addition to describing the name, attributes, and placement of

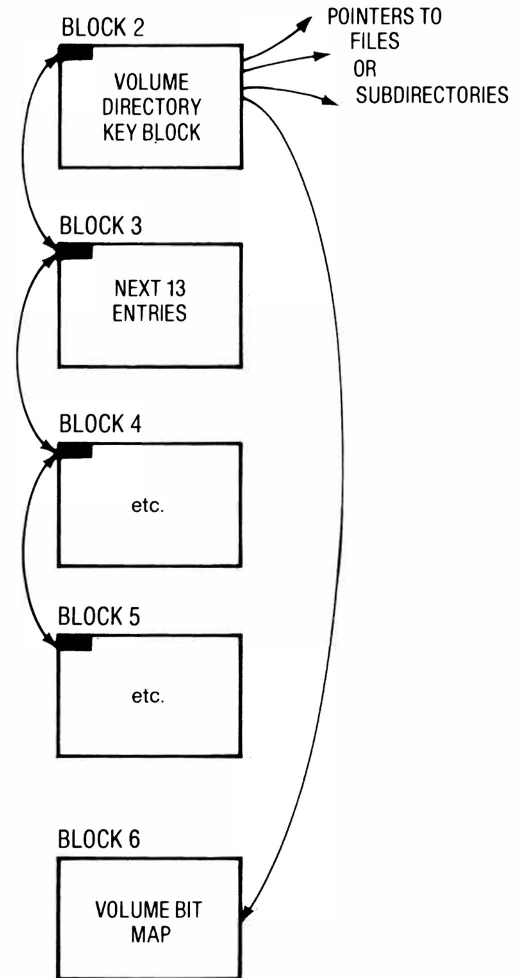


Figure 4.3 Linking of Volume Directory and Volume Bit Map

each file, it also contains the block number of the **Volume Bit Map** which will be described next. The first four bytes of every Volume Directory block are reserved for "pointers" to (the block numbers of) the previous Volume Directory block and the next Volume Directory block. This structure is called a doubly-linked list and is handy in that, from any block, it is easy to move forward or backward thought the directory entries. The Volume Directory and Volume Bit Map are diagrammed in Figure 4.3.

VOLUME SPACE ALLOCATION — THE VOLUME BIT MAP

When a diskette volume is first formatted, only the first seven blocks described above are marked in use. All of the remainder of the diskette blocks are considered "free" for use with files yet to be created. Each time a new block is required for a file, the free block with the lowest number is used. To keep track of which blocks have been used and which are free, ProDOS maintains one block as the Volume Bit Map. The Volume Bit Map is located by following a pointer in the Volume Directory, however, it is almost always in block 6. It consists of 512 bytes, each byte representing eight blocks on the volume. If the bytes are examined in binary form, each consists of eight bits having a value of one or zero. Thus, if block zero is in use as it always is, then the first byte's first bit is set to zero. If the ninth block (block 8) is free, then the first bit of the second byte is set to one. Since there are many more bits in the Volume Bit Map (4096 in all) than there could ever be blocks on a diskette, only the first 280 (or 35 bytes) are used. For a 5 megabyte hard disk, like the Apple ProFile, 1241 bytes are needed; in this case, since the number of blocks on the volume

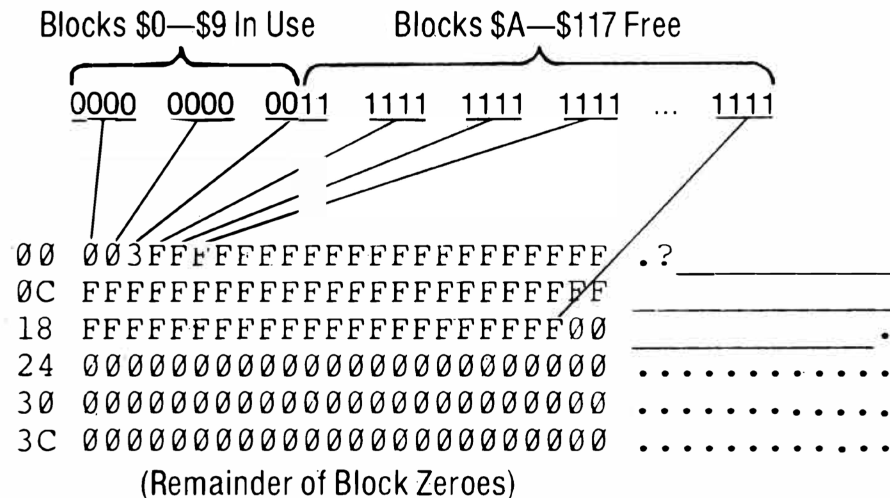


Figure 4.4 Example Volume Bit Map

is stored in the Volume Directory, ProDOS automatically knows to expect a bigger Volume Bit Map—one which is three blocks long. Bits which do not correspond to a real block (because it would be past the end of the volume) are set to zero. An example of a Volume Bit Map for the volume mapped in Figure 4.2 is given in Figure 4.4. Notice that, since three 1-block files have been allocated, a total of ten blocks are marked "in use."

THE VOLUME DIRECTORY

When ProDOS must find a specified file by name, it first reads block 2 of the diskette, the **key block** of the Volume Directory. If the filename is not found in this block, the next directory block is read, following the pointer in the third and fourth bytes of the current block. Typically, the Volume Directory blocks occupy blocks 2 through 5 of a volume. Of course, as long as a block pointer exists, linking one block to the next, and the first Volume Directory block is block 2, ProDOS does not really care where the rest of the directory blocks are located. Figure 4.5 diagrams the Volume Directory for the example given in Figure 4.2. The figure shows the "next block" pointer (bytes +2 and +3 in the block) of block

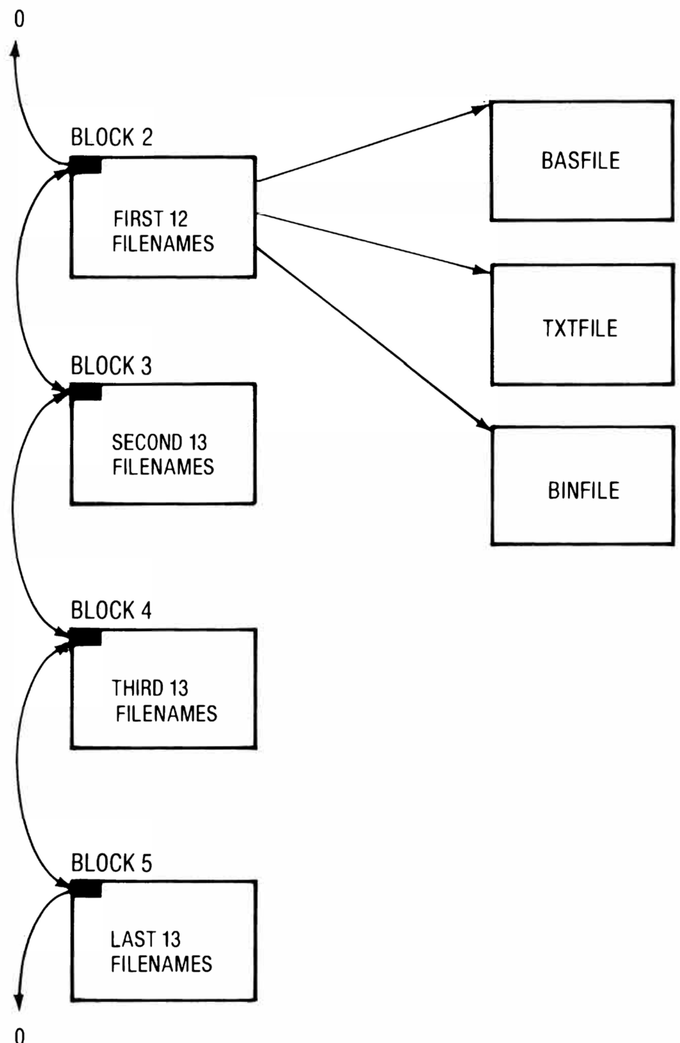


Figure 4.5 The Volume Directory

2 in the Volume Directory, as an arrow pointing to block 3. Each block, in turn, has block numbers in the same relative location (+0, +1 and +2, +3) which point backward to the previous block and forward to the next block respectively. If no previous or next block exists, a block number of zero is used to indicate this (block 0, being part of the boot image, would never be a valid block number for a directory or file block, so this is a safe convention). The first block in the Volume Directory (the key block) contains a special entry called the **header** which describes the directory itself and the characteristics of the volume, etc. This is followed by 12 file descriptive entries. All Volume Directory blocks other than the key block contain descriptions of up to 13 files each. (In practice, these entries can also be used to describe subdirectories, but this will be covered in detail later in the chapter.) Thus, with four Volume Directory blocks, a total of 4 times 13 less 1 (for the Volume Directory Header entry) or 51 files may be described.

THE VOLUME DIRECTORY HEADER

The Volume Directory Header is the first entry in the first block of the Volume Directory. As such, its first byte follows the four bytes of next/previous block pointers, so it's first byte is at +$04. A description of its format follows:*

BLOCK BYTE	DESCRIPTION
$Ø4	STORAGE_TYPE/NAME_LENGTH: The first nibble (top four bits) of this byte describes the type of entry. In this case, this is a Volume Directory Header so this nibble is $F. The low four bits are the length of the name in the next field (the volume name).
$Ø5-$13	VOLUME_NAME: A 15-byte field containing the name of this column. The actual length is defined by NAME_LENGTH above; the remainder of the field is ignored. No "/" is present as the first character since this is only used to delimit different level names but is not part of the names themselves.
$14-$1B	Reserved for future use. Usually zeroes.
$1C-$1F	CREATION: The date and time of creation (formatting) of this volume. This field is zero if no date was assigned. The format of the field is as follows: BYTE 0 and 1—yyyyyyym mmmddddd year/month/day BYTE 2 and 3—ØØØhhhhh ØØmmmmmm hours/minutes where each letter above represents one binary bit. This is the standard form for all create and modify date/time stamps in directories.

* Unless otherwise indicated, all multiple byte numeric values, such as block numbers, EOF marks, etc., are stored least significant byte first, most significant byte last (LO/HI).

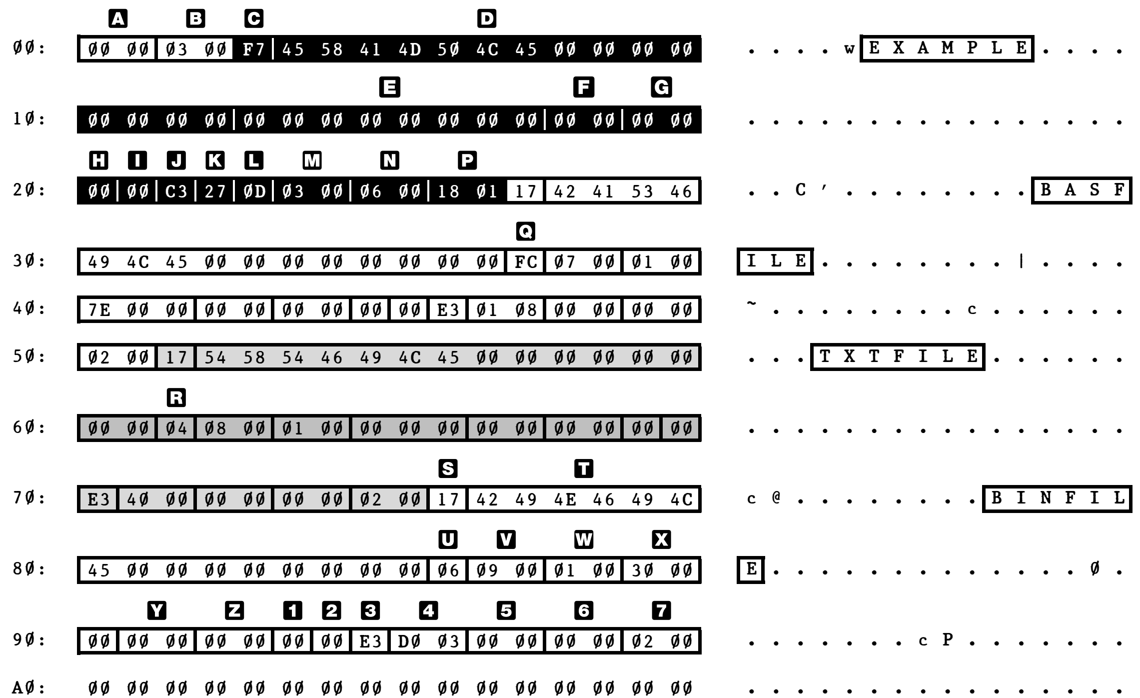
BLOCK BYTE	DESCRIPTION
$2Ø	VERSION: The ProDOS version number under which this volume was formatted. This field tells later versions of ProDOS not to expect to find any fields which were defined by Apple after this version of ProDOS was released. This field indicates the level of upward compatibility between versions. Under ProDOS 1.0, its value is zero.
$21	MIN_VERSION: Minimum version of ProDOS which can access this volume. A value in this field implies that significant changes were made to the field definitions since prior versions of ProDOS were in use and these older versions would not be able to successfully interpret the file structure of this volume. This field indicates the level of downward compatibility between versions. Under ProDOS 1.0, its value is zero.
$22	ACCESS: The bits in the flag byte define how the directory may be accessed. The bit assignments are as follows: $8Ø—Volume may be destroyed (reformatted) $4Ø—Volume may be renamed $2Ø—Volume directory has changed since last backup $Ø2—Volume directory may be written to $Ø1—Volume directory may be read All other bits are reserved for future use.
$23	ENTRY_LENGTH: Length of each entry in the Volume Directory in bytes (usually $27).
$24	ENTRIES_PER_BLOCK: Number of entries in each block of the Volume Directory (usually $ØD). Note that the Volume Directory Header is considered to be an entry.
$25-$26	FILE_COUNT: Number of active entries in the Volume Directory. An active entry is one which describes a file or subdirectory which has not been deleted. This count does not include the Volume Directory Header. Note that this field's name is a bit misleading since the count also includes subdirectory entries.
$27-$28	BIT_MAP_POINTER: The block number of the first block of the Volume Bit Map described earlier. This value is usually 6.
$29-$2A	TOTAL_BLOCKS: The total number of blocks on this volume. $Ø118 is for a 35-track diskette (280 decimal). This number may be used to compute the number of blocks in the Volume Bit Map as described earlier.

FILE DESCRIPTIVE ENTRIES

Each file (or subdirectory) on a volume has a File Descriptive Entry in the Volume Directory or another directory. These entries all have the same format:

BYTE OFFSET	DESCRIPTION
$ØØ	STORAGE_TYPE/NAME_LENGTH: The first nibble (top four bits) of this byte describes the type of entry. Currently assigned values are: $Ø - Deleted entry. Available for reuse $1 - File is a seedling (only one data block) $2 - File is a sapling (2 to 256 data blocks) $3 - File is a tree (257 to 32768 data blocks) $D - File is subdirectory $E - Reserved for Subdirectory Header entry $F - Reserved for Volume Directory Header entry The low four bits are the length of the file or subdirectory name in the next field. When a file is deleted, a $00 is stored in this byte.
$Ø1-$ØF	FILE_NAME: a 15-byte field containing the name of this file. The actual length is defined by NAME_LENGTH above; the remainder of the field is ignored.
$1Ø	FILE_TYPE: Primary file type. The hexadecimal value of this byte gives the file type as show in the following table:

TYPE	NAME	DESCRIPTION
$ØØ		Typeless file
$Ø1	BAD	Bad block(s) file
$Ø4	TXT	Text file (ASCII text, msb off)
$Ø6	BIN	Binary file (8-bit binary image)
$ØF	DIR	Directory file
$19	ADB	AppleWorks data base file
$1A	AWP	AppleWorks word processing file
$1B	ASP	AppleWorks spreadsheet file
$EF	PAS	ProDOS PASCAL file
$FØ	CMD	ProDOS added command file
$F1-$F8		User defined file types 1 through 8
$FC	BAS	Applesoft BASIC program file
$FD	VAR	Applesoft stored variables file



POINTER FIELDS

- **A** ($ØØ) no previous DIR block
- **B** ($ØØ) next DIR block is 3

VOLUME DIRECTORY HEADER

- **C** $F indicates Volume Directory Header
  volume name is 7 characters long
- **D** volume name is "Example"
- **E** reserved
- **F** creation date ($ØØØØ = Not Set)
- **G** creation time ($ØØØØ = Not Set)
- **H** ($ØØ = 1.0) version of ProDOS
- **I** ($ØØ) minimum version of ProDOS
- **J** ($C3) access byte
- **K** ($27) entry length
- **L** ($ØD) entries per block
- **M** ($ØØØ3) file count
- **N** bit map pointer (Block $0006)
- **P** total blocks

OTHER FILES IN DIRECTORY

- **Q** BAS file type
- **R** TXT file type
- **S** $1 indicates seedling file
  $7 characters in name
- **T** filename is "BINFILE"
- **U** filetype is $Ø6 (BIN)
- **V** pointer to key block of file
- **W** total blocks used by file
- **X** length of file (3 byte value)
- **Y** creation date
- **Z** creation time
- **1** version of ProDOS
- **2** minimum version of ProDOS
- **3** access byte
- **4** auxiliary type
- **5** last modification date
- **6** last modification time
- **7** pointer to directory header block

Figure 4.6 Example Volume Directory Block

TYPE	NAME	DESCRIPTION
$FE	REL	Relocatable object module file (EDASM)
$FF	SYS	ProDOS system file

All other types are either SOS file types or are reserved by Apple for future use. See Appendix E for a complete list.

$11-$12 KEY_POINTER: The block number of the key block of the file. In the case of a seedling file, this is the block number of the index block. For tree files, this is the block number of the master index block. (More on these file structures later.) If the file is a subdirectory, this is the block number of its first block.

$13-$14 BLOCKS_USED: The total number of blocks used by this file including index blocks and data blocks. If the file is a subdirectory, this is the number of directory blocks.

$15-$17 EOF: The location of the end of the file (EOF) as a 3-byte offset from the first byte. This can also be thought of as the length in bytes of a sequential file.

$18-$1B CREATION: The date and time of the creation of this file. This field is zero if no date was assigned. The format of the field is as follows:

Byte 0 and 1 - yyyyyyym mmmddddd year/month/day

Byte 2 and 3 - ØØØhhhhh ØØmmmmmm hours/minutes

where each letter above represents one binary bit. This is the standard form for all create and modify date/time stamps in directories.

$1C VERSION: The ProDOS version number under which this file was created. This field tells later versions of ProDOS not to expect to find any fields which were defined by Apple after this version of ProDOS was released. This field indicates the level of upward compatibility between versions. Under ProDOS 1.0, its value is zero.

$1D MIN_VERSION: Minimum version of ProDOS which can access this file. A value in this field implies that significant changes were made to the file structure definition since prior versions of ProDOS were in use, and these older versions would not be able to successfully interpret the file structure of this file. This field indicates the level of downward compatibility between versions. Under ProDOS 1.0, its value is zero.

$1E ACCESS: The bits in this flag byte define how the file may be accessed. The bit assignments are as follows:

$8Ø - File may be destroyed

$4Ø - File may be renamed

$2Ø - File has changed since last backup

$Ø2 - File may be written to

$Ø1 - File may be read

All other bits are reserved for future use. An unlocked file's ACCESS is usually $E3. If a file is locked, ACCESS will be set to $21. Subdirectory files which have a non-zero FILE_COUNT field will be locked until all files described by them are deleted.

Value	Read?	Write?	Rename?	Delete?	ProDOS Use
$20	-	-	-	-	
$21	Y	-	-	-	Locked
$22	-	Y	-	-	
$23	Y	Y	-	-	
$60	-	-	Y	-	
$61	Y	-	Y	-	
$62	-	Y	Y	-	
$63	Y	Y	Y	-	
$A0	-	-	-	Y	
$A1	Y	-	-	Y	
$A2	-	Y	-	Y	
$A3	Y	Y	-	Y	
$E0	-	-	Y	Y	
$E1	Y	-	Y	Y	
$E2	-	Y	Y	Y	
$E3	Y	Y	Y	Y	Unlocked

$1F-$2Ø AUX_TYPE: Auxiliary type field whose contents depend upon FILE_TYPE. Common uses are as follows:

Type	Use
TXT	Random access record length (L from OPEN)
BIN	Load address for binary image (A from BSAVE)
BAS	Load address for program image (when SAVEd)
VAR	Address of compressed variables images (when STOREd)
SYS	Load address for system program (usually $2ØØØ)

$21-$24 LAST_MOD: Date and time at which file was last modified. This field is zero if no date was assigned. Format is identical to CREATION above.

$25-$26 HEADER_POINTER: Block number of the key block for the directory which describes this file.

Figure 4.6 is an example of a typical Volume Directory block for the example introduced with Figure 4.2. In this case, there are only three files on the diskette so only the first three directory entries are filled in. The remaining directory entries have never been used and contain zeroes.

FILE STRUCTURES

One of ProDOS's major jobs is to keep track of the blocks which make up a file. When programming, the user need never know that a file is actually made up of one or more blocks scattered far and wide all over the diskette volume. ProDOS must make the file appear to the programmer to be a continuous stream of sequential data.

So far the files shown in the examples here have only had one block. This was done to avoid complicating the discussion of the Volume Directory. In practice, however, very few files are 512 bytes or less in length. ProDOS defines three file structures to handle files of different sizes:

The Seedling	— for files of 512 bytes or less
The Sapling	— for files with more than 512 bytes but less than 128K bytes of data
The Tree	— for files with more than 128K bytes of data up to 16 megabytes (16,777,216 bytes)

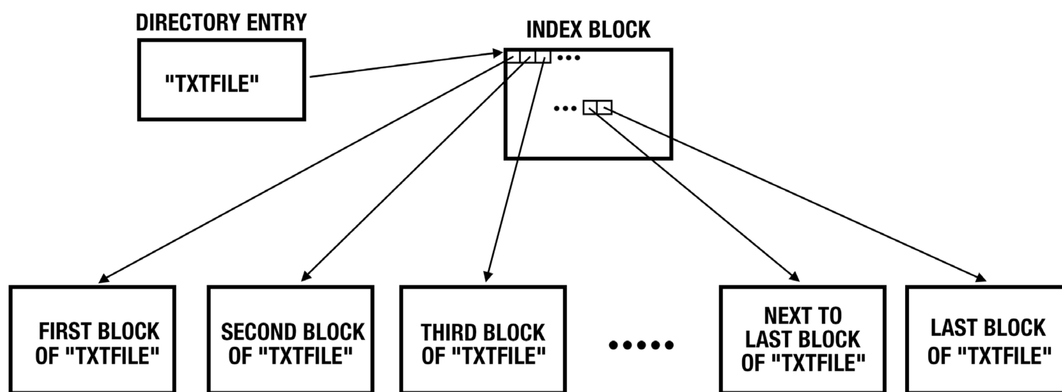


Figure 4.7 Sapling File Organization

Examples of **seedling** files have already been shown. A seedling file consists of a single **data block** whose number is stored in the `KEY_POINTER` field in the file entry of the directory. Thus, a seedling file, by definition, costs only one block of storage (and a file descriptive entry).

FIRST HALF OF INDEX BLOCK (LSBs)

	A												B			
ØØ	08	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	00	00	00
1Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
AØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
BØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
CØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
DØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
EØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
FØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

SECOND HALF OF INDEX BLOCK (MSBs)

	A												B			
ØØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
1Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
4Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
5Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
6Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9Ø	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
AØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
BØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
CØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
DØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
EØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
FØ	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

A $0008 First data block of file is block $8

B $0016 Last data block of file is $16

Figure 4.8 Example Sapling Index Block

For the purposes of this discussion, let us assume that we had run the following Applesoft BASIC program against our example disk volume from Figure 4-2.

```
1Ø PRINT CHR$(4);"OPEN TXTFILE,L64"
2Ø FOR I=Ø TO 2
```

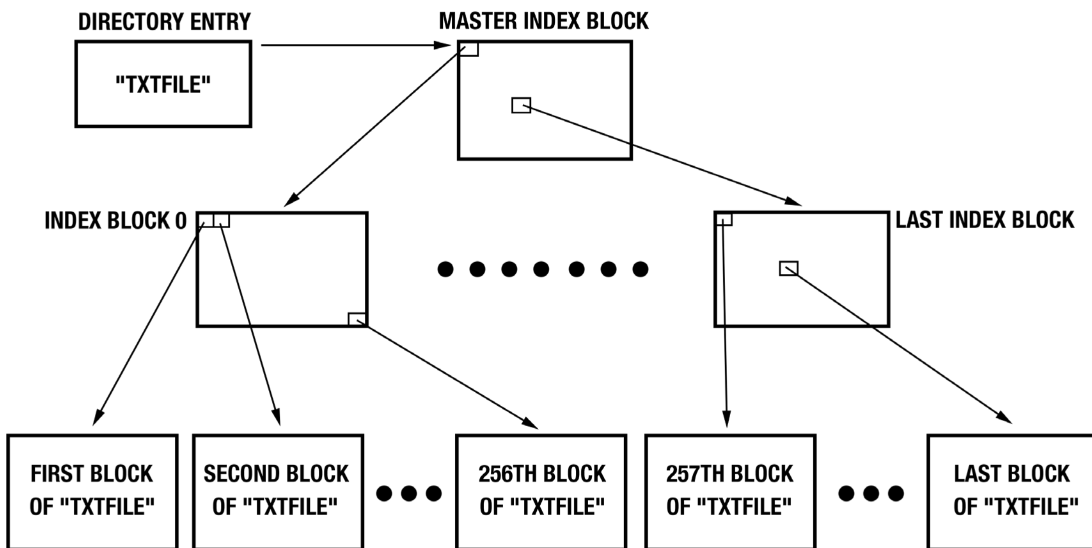


Figure 4.9 Tree File Organization

```
3Ø PRINT CHR$(4);"WRITE TXTFILE,R";I
4Ø PRINT "RECORD";I
5Ø NEXT I
6Ø PRINT CHR$(4);"CLOSE TXTFILE"
7Ø END
```

This program creates the `TXT` file, "TXTFILE", with a record length of 64 bytes. It then writes three records containing the strings "RECORD0", "RECORD1", and "RECORD2". The total size of this file is then 3 times 64 or 192 bytes. Since this is less than 512 bytes, the file is stored as a seedling.

Now, assume that statement 20 is changed to read:

```
2Ø FOR I=Ø TO 1ØØ
```

and the program is rerun. The file it creates will now contain 101 records of 64 bytes each, so the total size is 6464 bytes. As the ninth record is written (RECORD8), ProDOS discovers that the original seedling block is full. There is no room in the directory to store another block number, so ProDOS creates what is called an **index block**. This block contains the block numbers of each data block in the file in the order that they should be accessed. Using an index block, ProDOS can describe the file in a sequential and orderly way, even though its data blocks may not be physically contiguous (next to one another on the diskette). For example, if the previous data block in a file was 47, it is not necessary to store the data which follows it in block 48. Instead, any free

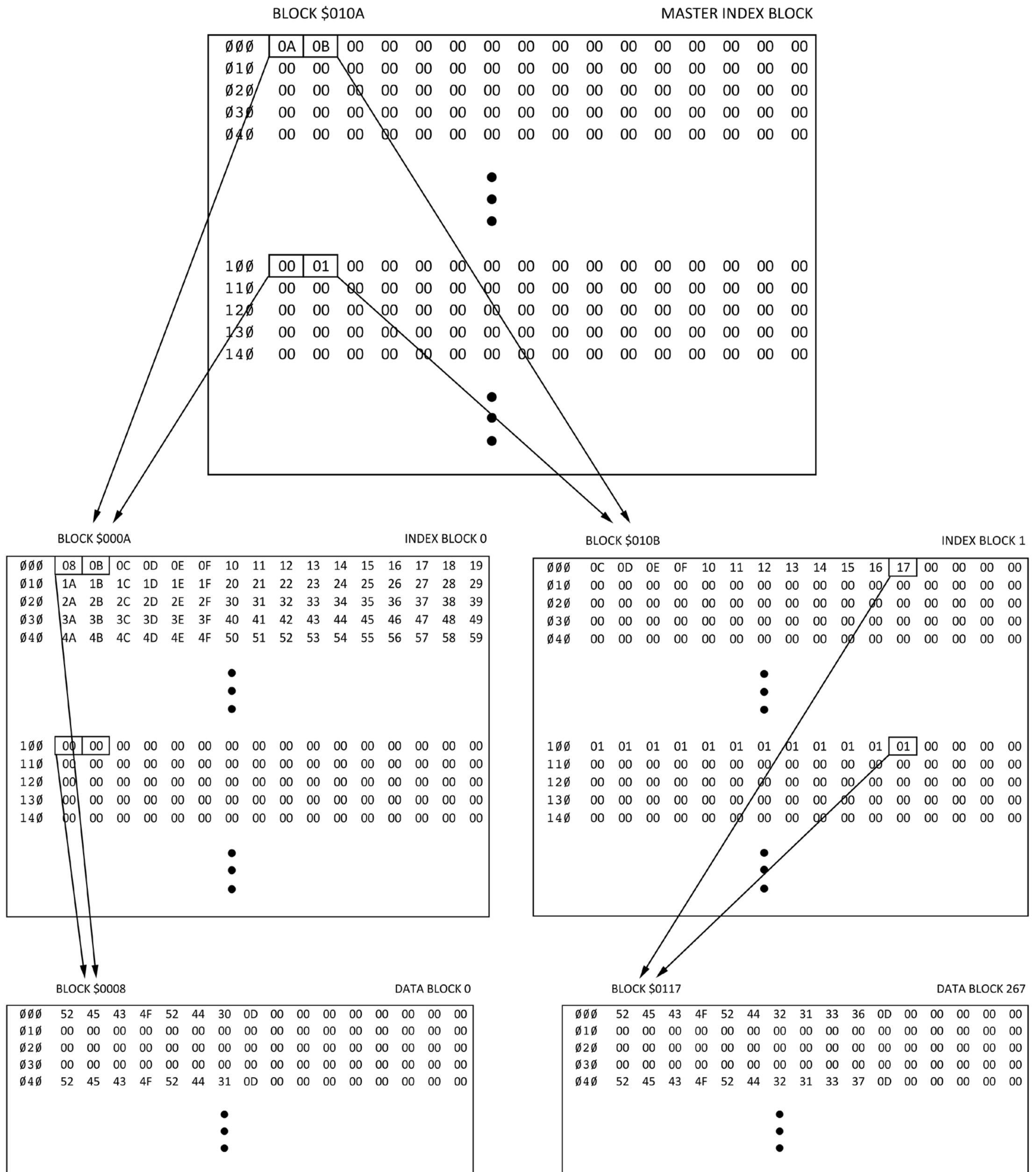


Figure 4.10 Example Tree File

block located anywhere on the diskette may be used simply by placing its block number next to 47's in the index block.

Thus, in our example, a new block is allocated to be the index block (`$A`), another new block is allocated to be the second data block (`$B`), both the original data block's number and the new data block's number are placed in the new index block, and, finally, the directory entry for the file is updated so that it now points to the index block instead of the seedling data block. Of course, the `STORAGE_TYPE` field in the directory entry must also be changed to indicate that this is now a **sapling** file and is no longer a seedling. Index block entries which are not associated with any data block yet (such as those beyond the end of file position) are set to zeroes. Since a block is 512 bytes long and the block numbers require a 2-byte field, this index block can store pointers to up to 256 data blocks representing up to 131072 bytes of data (128K). Obviously, most files will fall within this class of file structure. A diagram of the general form of a sapling file is given in Figure 4.7.

The index block for TXTFILE is given in Figure 4.8. Notice that the first block of the file is still block 8, the original data block of the old seedling version of TXTFILE. Notice also that in an index block, the least significant byte of the block numbers are stored in the first half of the block, and the most significant (in this example all MSB's are `$ØØ`) in the last half. This was done to simplify indexing into the block (the 6502 index registers can only index up to 256 bytes at a time). Thus, to find any given block, one must assemble a block number by picking the Nth byte and the N + 256th byte in the index block where N is the relative block desired.

Suppose that we now modify our program again so that 2144 records will be written. This pushes the total file size up to 137,216, more than can be described by a single index block. ProDOS must "promote" the file to the next level of the hierarchy, a **tree** file. A tree file consists of a single **master index block**, pointed to by the directory entry, which, in turn, contains the block numbers of two or more other index blocks. These lower level index blocks contain the actual data block numbers. This structure is diagrammed in Figure 4.9. Thus, since the master index block can describe 256 "subindex" blocks, and each subindex block can describe 256 data blocks, in principle this structure would support files of up to 32 megabytes! In order to limit block numbers to a 2-byte signed value of 32767, however, an arbitrary upper limit of 16 megabytes was imposed. In other words, a master index block can never be more than half full.

The entire structure for TXTFILE is depicted in Figure 4.10. Note that the original index block of the sapling file (block `$A`) became the first subindex

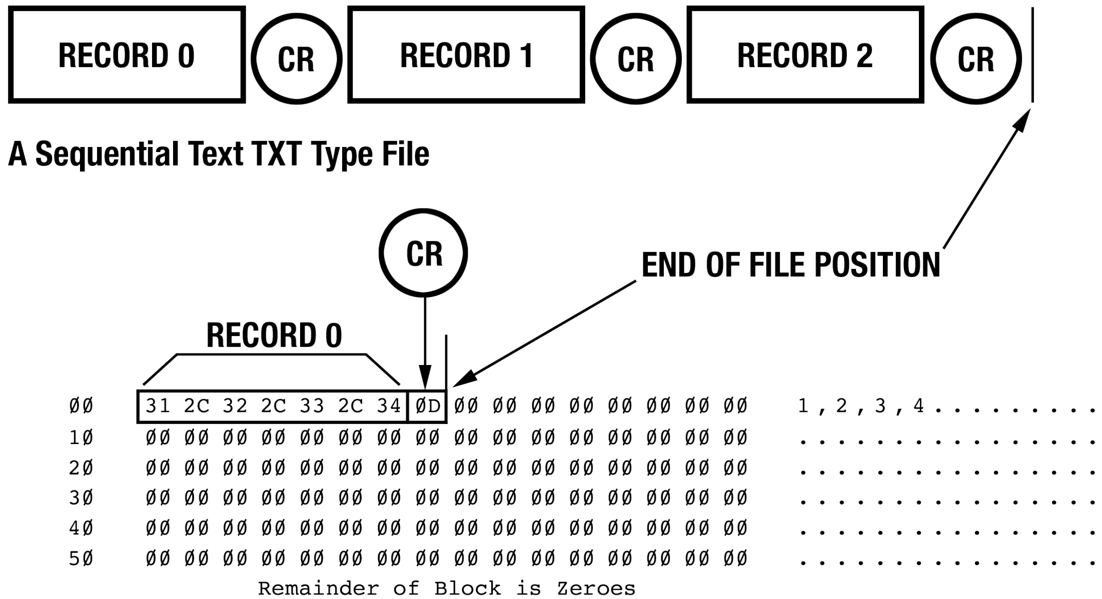


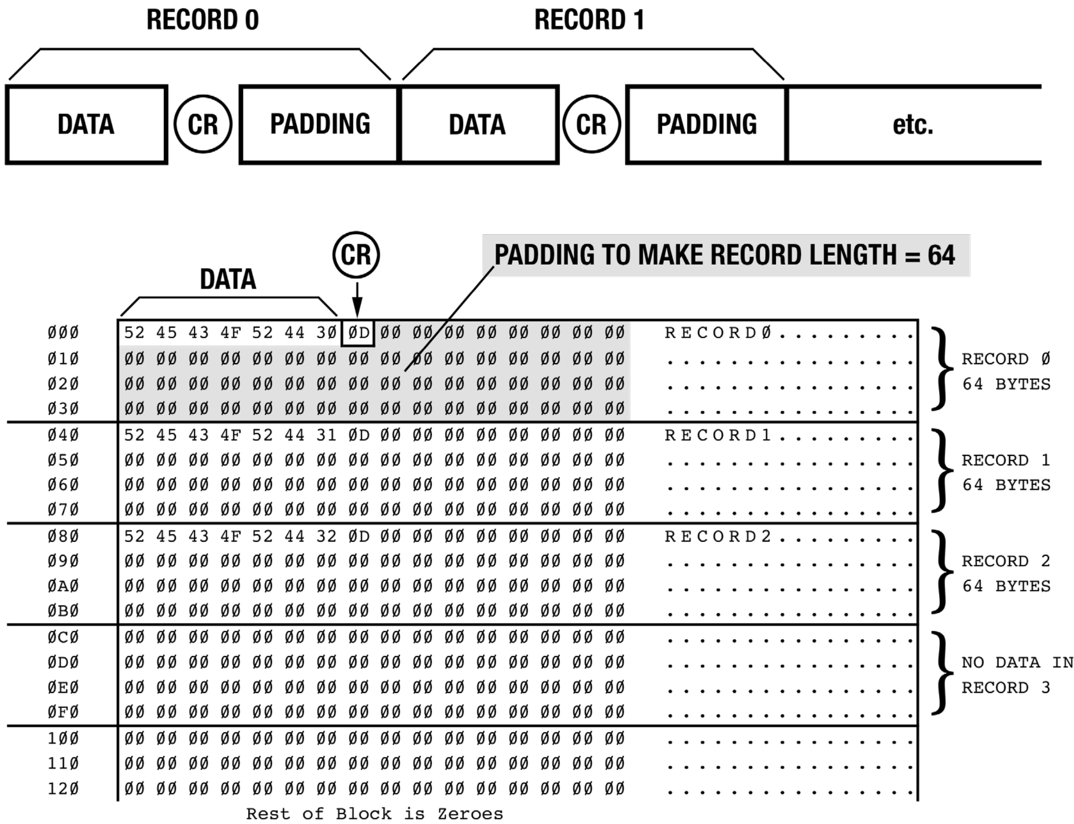
Figure 4.11 Example Sequential Text File Block

block of the tree file. Also, when the changeover was made, the master index block was allocated first (`$1ØA`), then the second subindex block (`$1ØB`), and finally the data block whose allocation made the file into a tree (`$1ØC`). The last block allocated is for RECORD2136 through RECORD2143 (for a total of 2144 records). This is the last block on the diskette (`$117`), and, since no blocks were ever freed, the diskette is now full. Although TXTFILE has only two subindex blocks and it is nearly as large as diskette, this does not imply that all tree files will have two subindex blocks, as will become apparent when sparse files are discussed.

FILE DATA TYPES

Unless they are directories (`DIR` type files), all files conform to one of three file structures described above even though the data in files may have different intended uses. A file might contain an Applesoft BASIC program which was `SAVE`d in addition to being a sapling file. It might be a binary memory image which was `BSAVE`d and conforms to the seedling structure. Or it might be data for a BASIC program in a `TXT` file and have the tree characteristic. File types, such as `BAS`, `TXT`, or `SYS` are less important to ProDOS than they are to the programs which use the files. This means that the basic structure of a `BAS` file is identical to that of a `BIN` file—only the interpretation of the data

A Random Text TXT Type File



AUX_TYPE in directory contains record length = 64 ($40)

Figure 4.12 Example Random Text File Block

differs. ProDOS maintains a consistent set of file types by convention, and to a limited extent, the BASIC command interpreter enforces these conventions (e.g., "`FILE TYPE MISMATCH`"). You are not prevented, however, from storing an Applesoft BASIC program image in a `TXT` file if you really work at it!

TXT FILES

The `TXT` or text file in its **sequential** form is the least complicated file data type (in its random form, it is, perhaps, the most complex). A sequential `TXT` file consists of one or more records, separated from each other by carriage return characters (hex `$ØD`s). This structure is shown and an example file is given in Figure 4.11. Usually, the end of a `TXT` file is signaled by the End Of

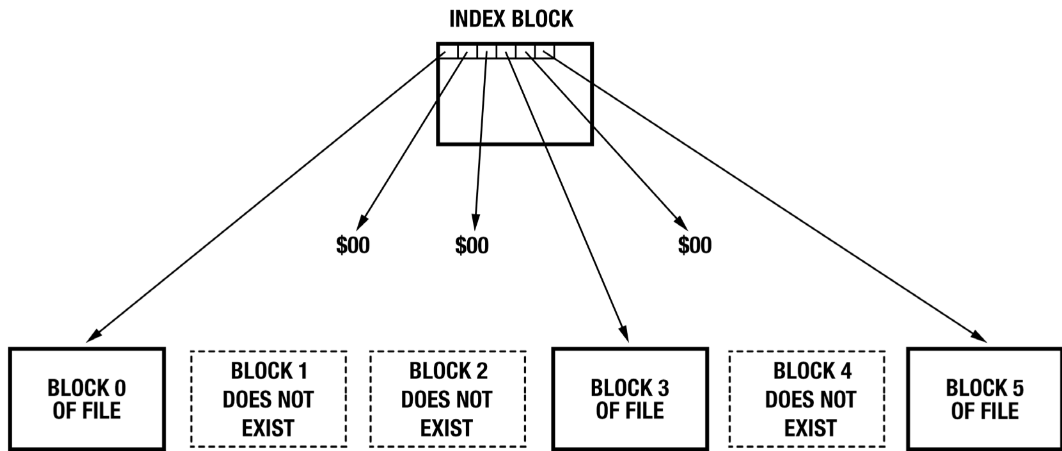


Figure 4.13 A Sparse File

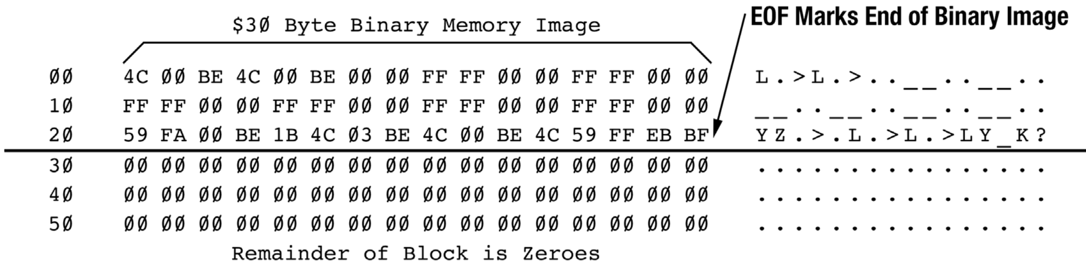
File (`EOF`) position stored in the directory entry for the file. Since `$ØD` is used to delimit records, carriage returns should not appear within a record. Usually, only valid ASCII characters are allowed in a `TXT` file to make them accessible to BASIC programs (i.e. printable text, numerics or special characters; refer to p. 8 of the *Apple II Reference Manual* or p. 16 of the *Apple II Reference Manual for IIe Only*). This restriction makes processing of a `TXT` file slower and less efficient in the use of disk space than with a `BIN` or `VAR` type file, since each digit must occupy a full byte in the file.

When `TXT` files are accessed **randomly**, or by record number, "holes" can appear between records. In the example given earlier and in Figure 4.12, each record is allocated 64 bytes of space in the file. By doing this, it is easy to find any record by multiplying its number by 64 and using this as a byte offset into the file. The record length is chosen as the maximum amount of space any record might occupy. Thus, records with less than 64 bytes of data, such as the ones in the example, will have wasted space at their end (filled in this case with `$ØØ`s). This wasted space is called padding. The actual data in each record is terminated with a `$ØD` (carriage return) just as in the sequential text file record (allowing BASIC to read it as a single `INPUT` line). In this way, data within a single record can be accessed as if it was a miniature sequential `TXT` file. If an attempt is made to sequentially read beyond into the padding, a null string is returned.

When the randomly organized file is `OPEN`ed, the record length given with the "L" keyword is stored in the `AUX_TYPE` field in the directory entry for the file. Then, if later `OPEN`s omit this keyword, the original value can be supplied by ProDOS.

MEMORY IMAGE...

A Binary BIN Type File




```
AUX_TYPE in Directory Contains Address = $Ø3DØ
EOF in Directory Contains Length = $3Ø
```

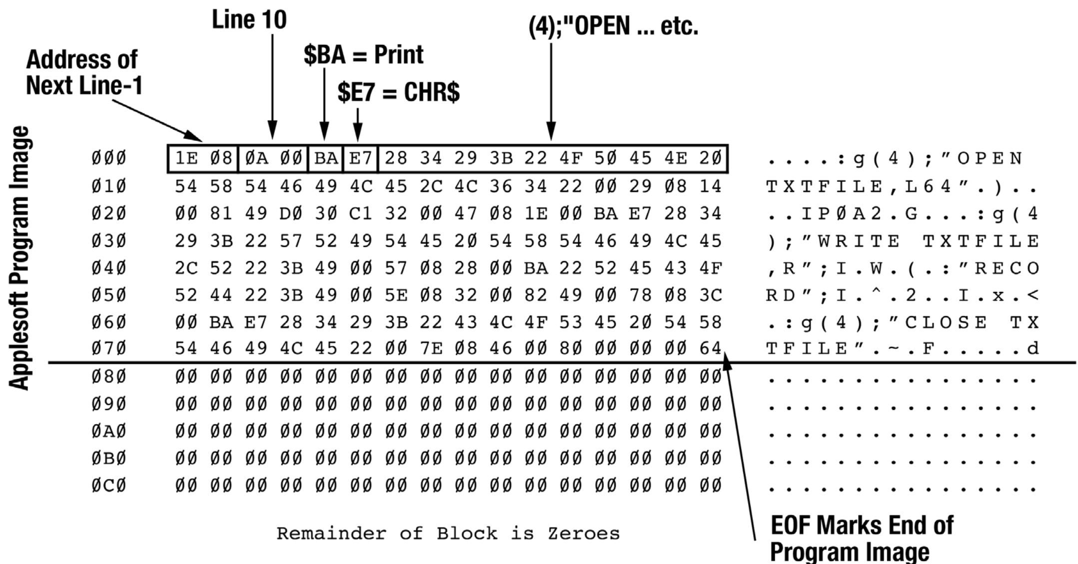
Figure 4.14 Example BIN File Block

Notice that in the example in Figure 4.12, record 3 has not been initialized. Indeed, none of the other records following RECORD2 have anything but `$ØØ`s in them. By `WRITE`ing to specific records in a nonsequential order, it is possible to leave very large holes between records which contain data. Such files are called "sparse." If a hole falls within a block which has other records which contain data, it is represented by binary zeroes. But if the hole covers entire blocks, ProDOS does not bother to allocate them at all. There is no point in wasting disk space on holes! Thus, if the next record containing data in our example was RECORD25, for instance, the rest of block 0 would contain zeroes (as it does now), no block would be allocated for block 1 or block 2, and block 3 would contain zeroes until the position of RECORD25 was reached. This is diagrammed in Figure 4.13. Notice that the positions of the "phantom" blocks are marked in the file's index block with zeros. Thus, although the files covers a "data space" of six blocks, only three data blocks are actually allocated. It is possible to create a file with only two data blocks which covers the entire 16-megabyte data space. Such a file would incorporate one master index block with an entry at `+Ø` and at `+7F`. All the subindex blocks in between would be "phantom," or not allocated and marked with zero pointers. The first index block would contain a single entry at `+Ø` for the first data block. And the last index would contain a single entry at `+FF` for the last data block. A 16-megabyte file using only five blocks of disk space!

PROGRAM MEMORY IMAGE ...

An Applesoft BAS Type File

```
10  PRINT  CHR$ (4);"OPEN TXTFILE,L64"
20  FOR I = 0 TO 2
30  PRINT  CHR$ (4);"WRITE TXTFILE,R";I
40  PRINT "RECORD";I
50  NEXT I
60  PRINT  CHR$ (4);"CLOSE TXTFILE"
70  END
```



AUX_TYPE in Directory Contains Program Start Address = $0801
EOF in Directory Contains Program Length = $80

Figure 4.15 Example BAS File Block

BIN FILES

The structure of a `BIN` type file is shown in figure 4.14. An exact copy of the memory selected is written to the disk block(s). The original address from which the memory was copied is stored in the `AUX_TYPE` field of the directory entry for the file. The `EOF` position in the directory records the length of the binary image. These values are those given in the A and L (or E) keywords of the `BSAVE` command which created the file. ProDOS can be made to `BLOAD` or `BRUN` the image at a different address by specifying the A (address) keyword

when the command is entered, or by changing the address in the directory entry (this is sometimes necessary if the file cannot be BSAVEd from the location where it will run, such as from the screen buffer).

BAS FILES

A BASIC program is saved to the diskette in a way that is nearly identical to BSAVE. The format of a BAS file is given in Figure 4.15. When the SAVE command is typed, the ProDOS BASIC command interpreter determines the location of the BASIC program in memory and its length by examining Applesoft's zero page address. An image of the program is written to the file, and, again, the AUX_TYPE and EOF fields of the directory entry represent the address and length. Notice that the character representation of the program is somewhat garbled. This is because, in the interest of saving memory, BASIC "tokenizes" the program. Reserved BASIC words, such as PRINT, IF, END, or CHR$, are replaced with a single hexadecimal code value (set off from other characters by its most significant bit being forced on). A complete treatment of the appearance of a BASIC program in memory is outside the scope of this manual, but a partial breakdown of the program in Figure 4.15 is given.

OTHER FILE TYPES (VAR, REL, SYS)

Several other file types have been set aside by ProDOS. Many are those found in the SOS operating system (e.g. PCD, PTX, PDA for Pascal, etc.). These are listed in Appendix E and will not be covered here since they are not indigenous to ProDOS. Other ProDOS file types include BAD and CMD. BAD files are obviously intended to mark permanent I/O errors on a disk's surface from accidental use, but there seem to be no utilities within ProDOS 1.0 which create them. The CMD and PAS file types are not currently supported by the ProDOS BASIC command interpreter, so their planned structures are anyone's guess. AppleWorks file types are designed for the AppleWorks package, and their structures are specific to that package. The formats of the VAR, REL, and SYS files are defined, however.

The VAR filetype is used to store the contents of a BASIC program's variables using the STORE command. The ProDOS BASIC command interpreter compresses all of the strings together with the numeric variables and saves the resulting chunk of memory as a VAR file. The first five bytes of the file constitute a header which defines the memory image that follows:

VAR FILE HEADER		
Byte	Offset	Length Description
+0	(2 bytes)	Combined length of simple and array variables
+2	(2 bytes)	Length of simple variables only
+4	(1 byte)	MSB of HIMEM when these variables were STOREd
+5	(n bytes)	Start of memory image...

The AUX_TYPE field of the directory entry for the file contains the starting address from which the compressed variables were copied. EOF is an indication of the end of the image. When a RESTORE is later issued, the memory image is reloaded, the strings are separated from the rest of the variables, and, if necessary, string pointers are adjusted based on the new HIMEM value.

The REL file type is used with a special form of binary file, containing the memory image of a machine language program which may be relocated anywhere in memory based upon additional information stored with the image

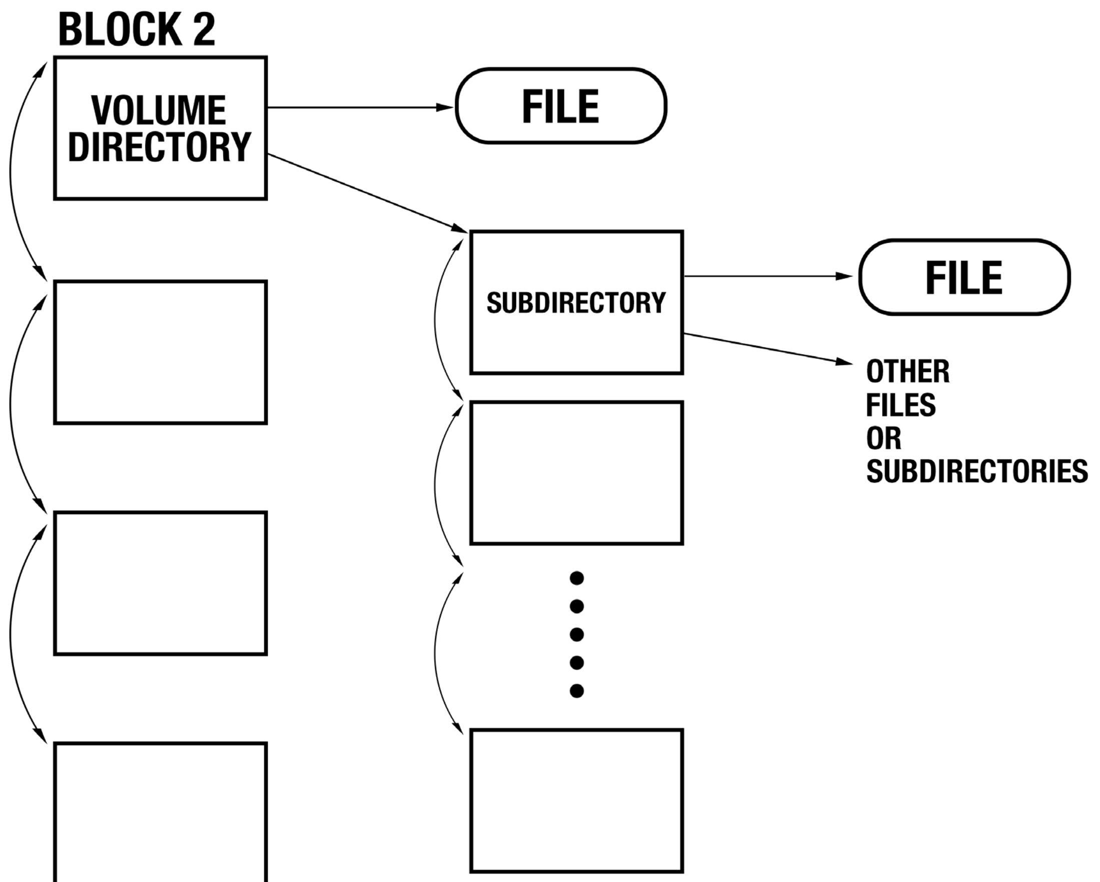


Figure 4.16 A ProDOS Subdirectory

itself. Such a file is called a Relocatable Object Module File and is produced as output from the Apple Toolkit Assembler (EDASM). The format for this type of file is given in the documentation accompanying the assembler.

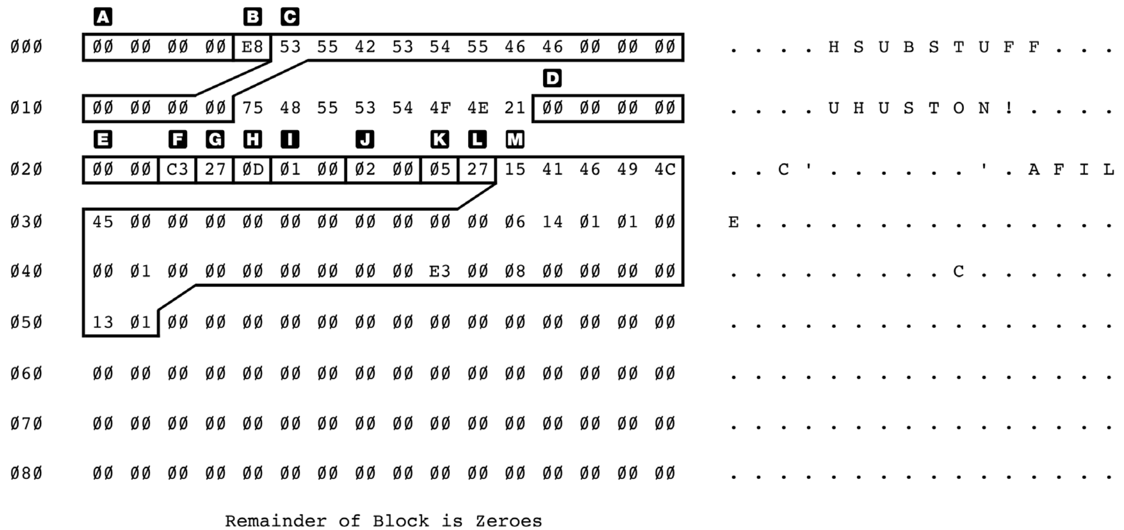
A SYS, or system file, is just like a BIN file except that it nearly always loads at $2000 and implies a reload of the command interpreter after it exits. SYS files are invoked with the "–", or smart RUN command, from the BASIC command interpreter. The interpreter closes all open files, frees all of the memory occupied by itself, and does a standard BRUN at $2000.

DIR FILES — PRODOS SUBDIRECTORIES

Since the Volume Directory has room for just 51 entries, without subdirectories, you would be limited to 51 files per volume. This may not seem to be much of a hardship on a diskette (although it might, since DOS 3.3 allows 105), but on a hard disk with 5 million bytes or more this limit is unthinkable. In order to create a more dynamic and flexible structure, the user is permitted to create subdirectories. A subdirectory can be thought of as an extension to the Volume Directory, but there is more to it than that. In the simplest case, a subdirectory is created and an entry which describes it is placed in the Volume Directory. The subdirectory has a structure very similar to the Volume Directory: it has a header entry located at its beginning; its blocks are doubly linked by pointers in the first four bytes of each block; and it can contain file descriptive entries (including entries for "sub-subdirectories"). Unlike the Volume Directory, however, it can be of any length (it starts out with only a single block and more are added as required), its header has a slightly different format, it can be located anywhere on the diskette, and its blocks are not necessarily contiguous. A diagram of a typical subdirectory is shown in Figure 4.16. Thus, within a single subdirectory, you can create as many file entries as you have disk blocks! In practice, however, it is usually more convenient to create multiple subdirectories "dangling" from the Volume Directory, each for a specific purpose (e.g. one for word processing, one for program development, one for spreadsheets, and so on). These subdirectories might even be thought of as miniature "diskettes" within the larger volume. Although it is possible to set up very complex structures using subdirectories (multiple level tree-like networks), usually this is not very efficient or convenient and a single level (all subdirectories linked directly to the Volume Directory) works best.

One of the major concepts around which ProDOS was designed is the notion of a **path** to a file. Ordinarily, if a file is described by the Volume

SUBDIRECTORY HEADER



A $ØØØØ (previous block) $ØØØØ (next block) Pointer Fields (no other blocks)

B $E indicates subdirectory

$8 characters in DIR name "SUBSTUFF"

C Directory Name padded with $ØØs

D Creation date and time ($ØØØØ ØØØØ = not set)

E $ØØ ProDOS version that created this subdirectory

$ØØ Minimum version of ProDOS to access this subdirectory

F Access $C3 = DESTROY / RENAME / READ / WRITE Enabled

G Entry Length is $27 (39) Bytes

H $ØD (13) Entries per Block

I File Count (1 File)

J Parent Directory Starts in Block 2 (Volume Directory)

K Parent Entry Number (5th entry describes this subdirectory)

L Parent Entry Length is $27 (39) Bytes

M File Entry for "AFILE", Seedling, Type = BIN, Data Block = $114,

EOF = $Ø1ØØ, Full Access, AUX_TYPE = A$Ø8ØØ, DIR Header in Block $Ø113

Figure 4.17 Example Subdirectory Block

Directory, this path is very simple. ProDOS merely looks up the file in the Volume Directory and that is that. If the file is described by a subdirectory, however, ProDOS insists upon knowing how to find the subdirectory. Of course, ProDOS could systematically search all subdirectories for the file and all subdirectories of the subdirectories, and so on, but this would be very time consuming (especially if you had mistyped the file name and it didn't really exist!). Since the user usually knows which subdirectory contains the file (and, perhaps, which subdirectory describes that subdirectory, etc.) the practice is to tell ProDOS what path to follow to find a file. This is done by first specifying the volume to be searched, thereby naming the Volume Directory, followed by a list of all subdirectories which must be traversed to eventually find the file, and finally by the file name itself. For example, if in Figure 4.16 the volume name is "`VOLUME`" and the subdirectory is "`SUB`" and the file described by the subdirectory is "`FILE`," the path to find that file would be:

`/VOLUME/SUB/FILE`

If the file described by the Volume Directory in Figure 4.16 was also called "`FILE`" there would be no confusion at all, because its pathname would be unique:

`/VOLUME/FILE`

This points out an additional advantage of subdirectories. It was mentioned earlier that they were like miniature "diskettes," and, just like diskettes, there is no problem in using identical file names within different directories.

To make specifying pathnames easier, the user can specify a default prefix to ProDOS. When a file name is given (without a leading "`/`" in its name) it is assumed to be an incomplete pathname. To complete it, ProDOS merely attaches the prefix to the beginning. Thus, if the current prefix is:

`/VOLUME/SUB/`

And a reference was made to "`FILE`," ProDOS would create the following fully qualified pathname:

`/VOLUME/SUB/FILE`

Therefore, by specifying a prefix you are, in a sense, stating that you wish to work within a specific "miniature diskette," although you can still access any other file on the volume by giving its complete pathname explicitly.

An examples of a typical subdirectory block is given in Figure 4.17. The format of the Subdirectory Header is given below (remember that the first

four bytes of each subdirectory block contain the previous and next block numbers respectively):

Block Byte	Description
`$Ø4`	`STORAGE_TYPE/NAME_LENGTH`: The first nibble (top 4 bits) of this byte describes the type of entry. In this case, this is a Subdirectory Header so this nibble is `$E`. The low 4 bits are the length of the name in the next field (the subdirectory name).
`$Ø5-$13`	`SUBDIR_NAME`: A 15-byte field containing the name of this subdirectory. The actual length is defined by `NAME_LENGTH` above; the remainder of the field is ignored.
`$14`	`$14` must contain `$75`.
`$15-$1B`	Reserved for future use.
`$1C-$1F`	`CREATION`: The date and time of the creation of this file. This field is zero if no date was assigned. The format of the field is as follows: BYTE 0 and 1 - `yyyyyyymmmmddddd` year/month/day BYTE 2 and 3 - `ØØØhhhhhØØmmmmmm` hours/minutes where each letter above represents one binary bit. This is the standard form for all create and modify date/time stamps in directories.
`$2Ø`	`VERSION`: the ProDOS version number under which this subdirectory was created. This field tells later version of ProDOS not to expect to find any fields which were defined by Apple after this version of ProDOS was released. This field indicates the level of upward compatability between versions. Under ProDOS 1.0, it's value is zero.
`$21`	`MIN_VERSION`: Minimum version of ProDOS which can access this subdirectory. A value in this field implies that significant changes were made to the field definitions since prior version of ProDOS were in use and these older versions would not be able to successfully interpret the structure of this subdirectory. This field indicates the level of downward compatability between versions. Under ProDOS 1.0, its value is zero.
`$22`	`ACCESS`: The bits in this flag byte define how the directory may be accessed. The bit assignments are as follows: `$8Ø` - Subdirectory may be destroyed (deleted) `$4Ø` - Subdirectory may be renamed `$2Ø` - Subdirectory has changed since last backup `$Ø2` - Subdirectory may be written to `$Ø1` - Subdirectory may be read

Block Byte	Description
	All other bits are reserved for future use.
$23	ENTRY_LENGTH: Length of each entry in the Subdirectory in bytes (usually $27)
$24	ENTRIES_PER_BLOCK: Number of entries in each block of the Subdirectory (usually $ØD). Note that the Subdirectory Header is considered to be an entry.
$25-$26	FILE_COUNT: Number of active entries in the Subdirectory. An actice entry is one which describes a file or subdirectory which has not been deleted. This count does not include the Subdirectory Header. Note that this field's name is a bit misleading since the count also includes other subdirectory entries.
$27-$28	PARENT_POINTER: The block number (within the volume directory or a subdirectory) which contains the file entry for this subdirectory.
$29	PARENT_ENTRY: The number of the file entry within the block number pointed to by the PARENT_POINTER. Given that "ENTRIES_PER_BLOCK" is $ØD, then the PARENT_ENTRY number ranges from $Ø1 to $ØD.
$2A	PARENT_ENTRY_LENGTH: The length of entries in the parent directory in bytes (usually $27).

EMERGENCY REPAIRS

From time to time the information on a diskette can become damaged or lost. This can create various symptoms, ranging from mild side effects, such as the diskette not booting, to major problems, such as an input/output (I/O) error in the Volume Directory. A good understanding of the format of a diskette, as described previously, and a few program tools can allow any reasonably sharp Apple II user to patch up most errors on their diskettes.

A first question would be, "how do errors occur?" The most common cause of an error is a worn or physically damaged diskette. Usually a diskette will warn you that it is wearing out by producing "soft errors." Soft errors are I/O errors which occur randomly. You may get an I/O error message when you CATALOG a disk one time and have it CATALOG correctly if you try again. When this happens, a smart programmer immediately copies the files on the aged diskette to a brand new one and discards the old one or keeps it as a backup.

Another cause of damaged diskettes is the practice of hitting the RESET key to abort the execution of a program which is accessing the diskette. Damage will usually occur when the RESET signal comes just as data is being written onto the disk. Powering the machine off just as data is being written to the disk

is also a sure way to clobber a diskette. Of course, real hardware problems in the disk drive, cable, or controller card can cause damage as well.

If the damaged disk can be `CATALOG`ed, recovery is much easier. A damaged ProDOS bootstrap loader on track 0 can usually be corrected by formatting a fresh diskette and copying the files from the old one to the new one. If only one files produces an `I/O ERROR` when it is used, it may be possible to copy most of the sectors of the file to another diskette by skipping over the bad sector with an assembler language program which calls the MLI (Machine Language Interface) in the ProDOS Kernel, or with a BASIC program (if the file is a `TXT` file). Indeed, if the problem is a bad checksum (see Chapter 3), it may be possible to read the bad sector and ignore the error and get most of the data.

An I/O error usually means that one of two conditions has occurred. Either a bad checksum was detected on the data in a sector, meaning that all bytes in the sector which follow the point of damage may be lost; or the sectoring is clobbered such that the sector no longer even exist on the diskette. If the latter is the case, the diskette (or at the very least, the track) must be reformatted, resulting in a massive loss of data. Although a program can be written to format a single track (see Appendix A), it is usually easier to copy all readable sectors from the damaged diskette to another formatted diskette and then reconstruct the lost data there.

`I can't seem to find the PRODOS file around here anywhere!`

Disk utilities, such as Quality Software's *Bag of Tricks*, allow the user to read and display the contents of sectors or blocks. *Bag of Tricks* will also allow you to modify the sector data and rewrite it to the same or another diskette. If you do not have *Bag of Tricks* or another commercial disk utility, you can use the `ZAP` program in Appendix A of this book. The `ZAP` program will read any block of an unprotected disk into memory, allowing the user to examine it or modify the data and then, optionally, rewrite it to a disk. Using such a program is very important when learning about diskette formats and when fixing clobbered data.

Using `ZAP`, a bad sector within a file can be localized by reading each block listed in the index blocks for that file. If the bad block is in a directory, the pointers of up to 13 files may be lost. When this occurs, a search of the diskette can be made to find "homeless" index blocks (ones which are not otherwise connected to the remaining good directory blocks in that and other directories). As these index blocks are found, new file descriptive entries can be made in the damaged sector which point to these blocks. Of course, it helps to know whether the lost files are seedlings, saplings, or trees! When the entire Volume Directory is lost, this process can take hours, even with a good understanding of the format of ProDOS volumes. Such an endeavor should only be undertaken if there is no other way to recover the data. Of course, the best policy is to create backup copies of important files periodically to simplify recovery. More information on the above procedures is given in Appendix A.

A less significant but very annoying form of diskette clobber is the loss of free blocks. It is possible, by powering off or hitting RESET at the wrong time, to leave blocks marked in use in the Volume Bit Map which were about to be marked free. These lost blocks can never be recovered by normal means, even when files are deleted, since they do not belong to anyone. The result is a `DISK FULL` message before the volume is actually full. to reclaim the lost block, it is necessary to compare every block listed in every index block or directory against the Volume Bit Map to see if there are any discrepancies. There are utility programs which will do this automatically, but the best way to solve this problem is to copy all the files on the diskette to another diskette (note that the diskette must be copied on a file by file basis, not as a volume, since a volume copy would copy an image of the diskette, bad Volume Bit Map and all).

If a file is deleted, it can usually be recovered, providing that additional block allocations have not occurred since it was deleted. If another file was created after the `DELETE` command, ProDOS probably has reused some or all of the blocks of the old file. The appropriate directory can be quickly

ZAPped to reactivate the file (you will have to guess at the STORAGE_TYPE and NAME_LENGTH values) at +Ø in the deleted entry. The file should then be copied to another diskette and then the original deleted so that the Volume Bit Map will be correct.

FRAGMENTATION

ProDOS overhead in reading or writing blocks to a volume consists of three main parts:

1. ProDOS computational overhead time (the time to get ready to access the disk).
2. Seek time (moving the disk arm to the proper track).
3. Rotational delay (waiting for the proper sector to appear under the disk head).

In the first respect, ProDOS is an enormous improvement over Apple's earlier operating system, DOS, being up to eight times faster in its operation. This fact only increases the significance of the other two delay areas. Skewing can have an effect on rotational delay to some extent (see Chapter 3), but is much more difficult to control. Seek time, however, can vary greatly depending upon use patterns and the arrangement of files on a volume.

Imagine, for example, a volume on which a great deal of activity has occurred. Many files have been created and deleted over a period of time, leaving "holes" here and there as files are deleted, which are reallocated to existing or new files as necessary. Eventually, a map of the volume looks like a plate of spaghetti! There is nothing really wrong with this — files can be accessed normally — but if parts of an otherwise short file are spread all over the disk volume, ProDOS must spend a lot of time moving the disk read/write head from track to track to pick up all the pieces in the proper order. This costs time. A disk volume in this state of affairs is said to be badly "fragmented." Fragmentation can be even more important on a hard disk since the ratio of seek delay to rotational delay is much greater. Likewise, the best skewing setup in the world can be completely gutted by a fragmented disk, since few sequential files sectors are found together on the same track, and as the arm is moved to the new track, there is no telling how long the rotational delay will be.

When disk access time becomes a concern, it is sometimes useful to intelligently move files to specific spots on the disk. To accomplish this, the user must format a new, blank volume and copy the files from the old disk,

one by one, to the new disk in an appropriate order. Remember that ProDOS allocates blocks for files in numerically increasing order (from the outside track of the disk to the inside track). Thus, the first file you copy will be placed near the Volume Directory (a good place if you want to find that file fast). The last file you copy will go closest to the center hub of the diskette. If your program accesses two files at once, try to place them near one another on the disk. Do not separate them by many other files or you will hear the disk arm "thrashing" back and forth as it first accesses a block in file A and then must access one in file B. While you hear that noise, your program is not doing anything useful! Another thing to remember if your program opens and closes files frequently is that, when it does so, it may access several directories. It is usually a good idea in any case to keep all of your directories squashed down against the Volume Directory (i.e. `CREATE` all directories before you copy any files onto the new diskette) so that pathname searches will go faster.

Chapter 5

The Structure Of ProDOS

PRODOS MEMORY USE

ProDOS is an assembly language program which is loaded into RAM memory when the user boots the disk. Although the ProDOS machine language support routines can run by themselves in a machine smaller than 64K (or 48K plus a language card), ProDOS is primarily intended to run only on a full sized 64K or larger Apple II Plus or an Apple IIe or IIc. In a 64K Apple II, ProDOS normally occupies the 16K of bank switched memory (or the Language Card for older Apples) and about 10.5K at the top of main memory (`$96ØØ` through `$BFFF`). The part of ProDOS which occupies the bank switched memory is called the **Kernel**. The part occupying the top of main memory is called the **BASIC Interpreter** (BI). The Kernel consists of support subroutines which may be called by any assembly language program (such as the BASIC Interpreter) to access the disk, either block by block or file by file. The BASIC Interpreter accepts ProDOS commands entered by the user or inside programs, and translates them into calls to the Kernel subroutines.* When the BASIC Interpreter is loaded, ProDOS must fool Applesoft BASIC into believing that there is actually less RAM in the machine than there is. With ProDOS loaded, Applesoft believes that there is only about 38K of RAM. ProDOS does this by adjusting `HIMEM` after it has loaded the BASIC Interpreter to prevent Applesoft from using the memory ProDOS is occupying. In order to keep track of the memory it is using, ProDOS maintains a "bit map" table which describes every page (256 bytes) in memory and marks it either free or in use. By examining this table, user written programs can avoid using previously assigned memory, even if later versions of ProDOS are loaded elsewhere.

A diagram of ProDOS's memory is given in Figure 5.1. As can be seen, there are numerous subdivisions of the two basic components mentioned above.

---

* It is possible, if the BASIC Interpreter's functions are not required by an application (such as a stand alone arcade-type game), to separate the Kernel from the BASIC Interpreter and not even load the BASIC Interpreter. For the purposes of this discussion, however, we will assume that ProDOS consists of both the Kernel and the BASIC Interpreter. In addition, the ProDOS Kernel may be loaded into the main part of memory if the Apple does not have a language card (48K Apple II), but the BASIC Interpreter may not be used under these circumstances because it cannot be relocated.

In addition, there are two special **global pages** containing the addresses and data pertaining to the ProDOS Kernel (SYSTEM GLOBAL PAGE ar `$BFØØ`) and the BASIC Interpreter (BI GLOBAL PAGE at `$BEØØ`) which may be of interest to external user written programs. These global pages will be discussed in more detail later in this chapter.

As discussed earlier, ProDOS can be divided into two major components: the Kernel, containing the **Machine Language Interface** (MLI); and the BASIC Interpreter (BI). In theory, other interpreters could be written and substituted for the BI (to support Pascal or C language development, for example) but at present the only interpreter provided by Apple is the BASIC Interpreter, supporting Applesoft BASIC. There is currently no support for the older Integer BASIC language. In fact, because of the memory utilization of ProDOS, Applesoft must be resident in ROM (since the Kernel resides in the language card). Hence, ProDOS is only supported for Apple II Pluses, IIc's, and IIe's. Use of the term "BASIC Interpreter" should not be confused with the Applesoft BASIC interpreter in ROM.* Here, "interpreter" means "interpreter of disk access commands," and not "interpreter of BASIC language statements." Although the BI is closely "married" to the Applesoft interpreter in ROM, its primary responsibility is to interpret ProDOS commands which load and save files, display directories, and support file operations in BASIC programs.

The BI normally occupies memory from `$96ØØ` to `$BEFF`. The first 1K (`$96ØØ-$9AØØ`) is a general purpose buffer, used during Applesoft string garbage collection and for other purposes. Following this, at `$9AØØ`, are the actual machine language instructions and work areas of the BI. Any data which is considered to be of interest to external programs is placed in the BI Global Page at `$BEØØ`. As files are opened by BASIC programs, 1024-byte file buffers are allocated and inserted between the general purpose buffer and the BI itself. To do this, the BI must relocate the general purpose buffer and any strings which were allocated by the running BASIC program lower in memory to make room for the file buffers. `HIMEM` must be lowered accordingly. Thus, the memory available to the BASIC program fluctuates according to the number of open files.

The ProDOS Kernel occupies 12K of the 16K bank switched memory (language card). Most of the remaining 4K bank is not currently used, but is reserved by Apple for future use (the `QUIT` code occupies three pages currently). The main part of the ProDOS Kernel begins at `$DØØØ`, and contains the Machine Language Interface (MLI) subroutines which allow access to

* Apple's documentation also refers to the BASIC Interpreter as the "BASIC System Program." "BASIC Interpreter" is used here because of frequent references to the "BI," an earlier designation.

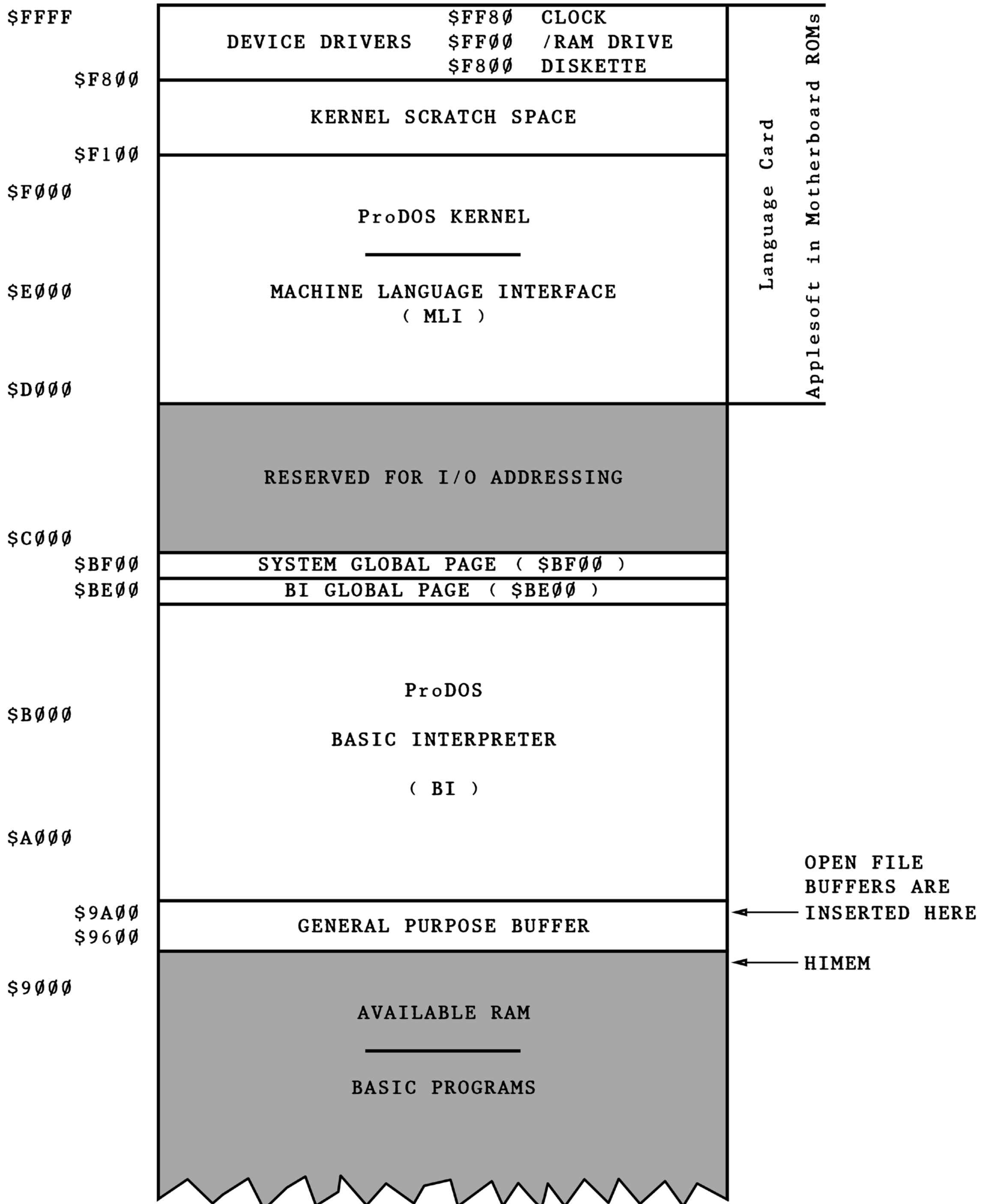


Figure 5.1 ProDOS Memory Usage (64K)

the disk by other programs (such as the BI or user written machine language programs). MLI functions provided include: open a file, create a new file, delete a file, rename a file, determine online volumes, read/write to a file, etc. The Kernel also handles interrupts for devices which can generate them. Access to these subroutines and their data is strictly controlled by the System Global Page which will be described next. Following the Kernel and its scratch space (work areas), is a 2K area devoted to device drivers. In order to provide a device independent interface to peripherals, subroutines are loaded here which can perform block oriented I/O to the Apple diskette drive, the /RAM "electronic" 64K memory diskette drive implemented in the Extended 80-Column Text card, and the Thunderclock. Additional device drivers (Hard disk, printer, etc.) must be placed in interface card ROM or in main RAM memory. The entry point addresses of each device driver in use are kept in the System Global Page.

GLOBAL PAGES

The System and BI Global Pages are maintained by ProDOS at fixed locations in main memory (`$BFØØ` and `$BEØØ` respectively). This practice allows important ProDOS data and subroutines to be accessed by external programs via a fixed location. Each time Apple makes a change in ProDOS and reassembles its source code, the addresses of all of the subroutines and variables may change. By putting the addresses of these routines and the variables themselves in fixed locations in memory, dependencies by a user written program on a particular version of ProDOS can be eliminated. Hopefully, all subroutines or data of general interest have been "vectored" through these global pages. If not, the programmer cannot be sure that a subroutine called directly will not move to a different memory location in a later version.

The exact format of the System Global Page is given in Chapter 8 but it contains the following information:

1. `JMP` (Goto) instructions to the main entry of the MLI, a quit vector, a clock/calendar subroutine, etc.
2. Addresses of the device drivers for each slot and drive.
3. A list of all disk drives online, and the slot and drive each occupies.
4. A "bit map" showing which pages of memory are in use and which are free.
5. Addresses of the buffers being used by MLI opened files.
6. Addresses of up to four interrupt handling routines and associated register save areas.
7. Current date, time and file level.

8. A machine ID flag byte giving the model (e.g., Apple IIe) and memory in the machine on which ProDOS is currently running.
9. Various flags indicating MLI status and whether a card occupies any slot.
10. Language card bank switching routines.
11. Interrupt entry and exit routines.
12. ProDOS version number.

The BI global page contains:

1. Addresses of routines in the BI which allow warmstart, command scanning, and error message printing.
2. I/O vectors for `PR#` and `IN#` for each slot, and the currently active input and output streams.
3. Default slot and drive.
4. BI status flags indicating whether an `EXEC` file is active. a BASIC program is running, a file is being read or written, etc.
5. Parameters that allow a user to pass an external command line to the BI.
6. A table indicating which commands allow which keyword parameters (e.g., `OPEN` does not allow the AD keyword but does allow the L keyword).
7. The current value for all keywords (A, B, E, L, S, D, etc.).
8. The address of the pathname buffers within the BI.
9. A subroutine used by the BI to access the MLI.
10. Parameter lists used by the BI to access the MLI.
11. Vectors to the BI's buffer allocate and free subroutines.
12. The current `HIMEM` MSB.

In addition to the ProDOS vectors in the global pages, the Monitor ROM and Applesoft maintain additional vectors of general interest from `$3FØ` through `$3FF`. They are:

Location	Vector
`$3FØ`	LO/HI address of the routine which handles a `BRK` machine language instruction. Supported by the Autostart and Apple IIe and Ilc ROMs. Normally contains the address of a Monitor ROM routine which prints the contents of the registers.
`$3F2`	LO/HI address of routine which will handle RESET for Autostart and Apple IIe ROM. Normally the BI restart address (`$BEØØ`) is stored here, but the user may change it you need to handle RESETs yourelf.

Location	Vector
`$3F4`	Power-up byte. Contains a "funny complement" of the RESET address with an `$A5`. This scheme is used to determine if the machine was just powered up or if RESET was pressed. If a power-up occurred, the Autostart ROM or Apple IIe ROM ignores the address at `$3F2` (since it has never been initialized), and attempts to boot a diskette. To prevent this from happening when you change `$3F2` to handle your own RESETs, `EOR` (exclusive OR) the new value at `$3F3` with an `$A5` and store the result in the power-up byte.
`$3F5`	A `JMP` to a machine language routine which is to be called when the "`&`" feature is used in Applesoft. Initialized by ProDOS to point to the BI command scanner vector.
`$3F8`	A `JMP` to a machine language routine which is to be called when a control-Y is entered from the monitor.
`$3FB`	A `JMP` to a machine language routine which is to be called when a non-maskable interrupt (NMI) occurs.
`$3FE`	LO/HI address of ProDOS's IRQ maskable interrupt handler dispatcher. If you wish to handle an IRQ interrupt, install an interrupt handler into ProDOS—do not replace this vector.

WHAT HAPPENS DURING BOOTING

When an Apple is powered on, its memory is essentially devoid of any programs. In order to get ProDOS running, a diskette is "booted." The term "boot" refers to the process of bootstrap loading ProDOS into RAM. Bootstrap loading involves a series of steps which load successively bigger pieces of a program until all of the program is in memory and running. In the case of ProDOS, bootstrapping occurs in two major stages, corresponding to the loading of the ProDOS Kernel and the BASIC Interpreter. Within these major stages, there are minor stages which must be performed to complete the loading process. Figures 5.2 and 5.3 diagram the processes involved in loading the Kernel and the BI respectively from the diskette. A description of this process follows.

The first boot stage is the execution of the ROM on the disk controller card. This is called the **Boot ROM**, and it exists on either the diskette controller card or a hard disk controller card at `$Cs00` (where "`s`" is the slot number). Thus, when the Apple is first powered on, the Monitor ROM searches the slots for a disk controller card (starting with slot 7 and moving down in slot number) and, upon finding one, it branches to `$Cs00` (usually `$C600` if the

controller is in slot 6). Control is also passed to this address should the user type `PR#6` in BASIC or `C6ØØG` or `6 Control-P` in the monitor. The diskette controller Boot ROM is a machine language program of about 256 bytes in length. When executed, it "recalibrates" the diskette arm by pulling it back to track 0 (the "clacketty-clack" noise that is heard), and then reads sector 0 from track 0 into RAM memory at location `$8ØØ`. Once this sector has been read, the Boot ROM goes to (`JMP`s) to `$8Ø1` which is the second stage boot, the ProDOS Loader.

The **ProDOS Loader** occupies the first block on a ProDOS diskette (physical sectors 0 and 2). Since the Boot ROM has only loaded sector 0, the

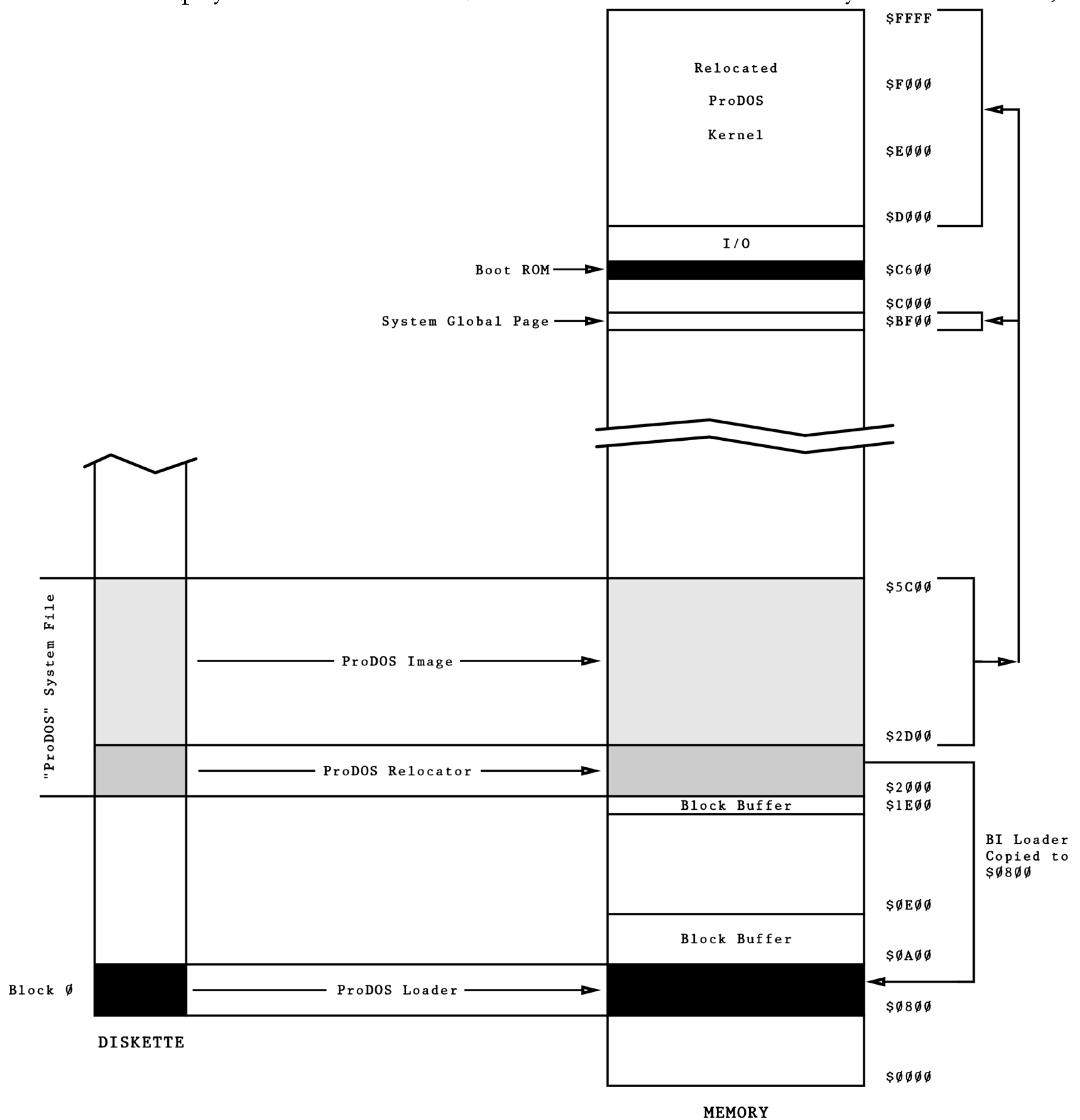


Figure 5.2 ProDOS Kernel Bootstrap Process

first task the ProDOS Loader must perform is to load the remaining sector of itself. It does this by calling the Boot ROM as a subroutine, loading it at $9ØØ. Having completed this, a portion of the Boot ROM is copied into a subroutine in the ProDOS Loader itself (this variable code is different for a diskette or a hard disk), and uses this to search the diskette's Volume Directory for a system file with the name "PRODOS". This file contains an image of the ProDOS Relocator, the BI Loader, and the ProDOS Kernel itself. If the file can be found, its contents are read into memory at $2ØØØ, and the ProDOS Loader jumps to the ProDOS Relocator at $2ØØØ.

The **ProDOS Relocator** prints a copyright and version number on the screen, and then begins to examine the machine in use to find out its model. This is done by testing the Monitor ROM for special model-dependent indicators and by checking for language card memory. The ProDOS Relocator assembles the data it has collected into a byte of flags indicating whether the machine is an Apple II, Apple II Plus, Apple IIe, Apple IIc, or an Apple III in Apple II emulation mode. It also indicates the amount of memory available. Once this has been established, the Kernel image is copied either to the bank switched memory (language card) if the machine has 64K or more, or to $9ØØØ for a 48K Apple. If the machine has 128K, a /RAM drive is set up in the alternate 64K memory. The peripheral card configuration is also checked, and a table of occupied slots and interface card identifications is made. The initialization of the Kernel is completed by moving an image of the System Global Page to $BFØØ and initializing it as necessary. The BI Loader image is then copied to $8ØØ and control transfers there to begin booting the BI.

The **BI Loader** searches the Volume Directory for the first system file it can find whose name ends with ".SYSTEM". The file which is found will normally be BASIC.SYSTEM, but any other interpreter could be loaded in this way. If a file is found, its contents are loaded into memory at $2ØØØ and control passes to the BI Relocator at $2ØØØ.

The **BI Relocator** copies the BI image to high memory ($9AØØ), sets up the BI Global Page at $BEØØ, and marks the pages occupied by these as "in-use" in the System Global Page's memory bitmap. The screen and keyboard vectors in zero page (CSWL/H and KSWL/H) are modified to cause immediate transfer of control to the relocator, and a jump to BASIC's coldstart entry is executed. As soon as Applesoft has completed initialization, it prints a prompt character "] ". This causes control to transfer back into the BI Relocator. CSWL/H and KSWL/H are restored to their normal settings, and initialization of the BI Global Page is completed. If a "STARTUP" file can be found in the Volume Directory, an initial command line of "-STARTUP" is dummied up

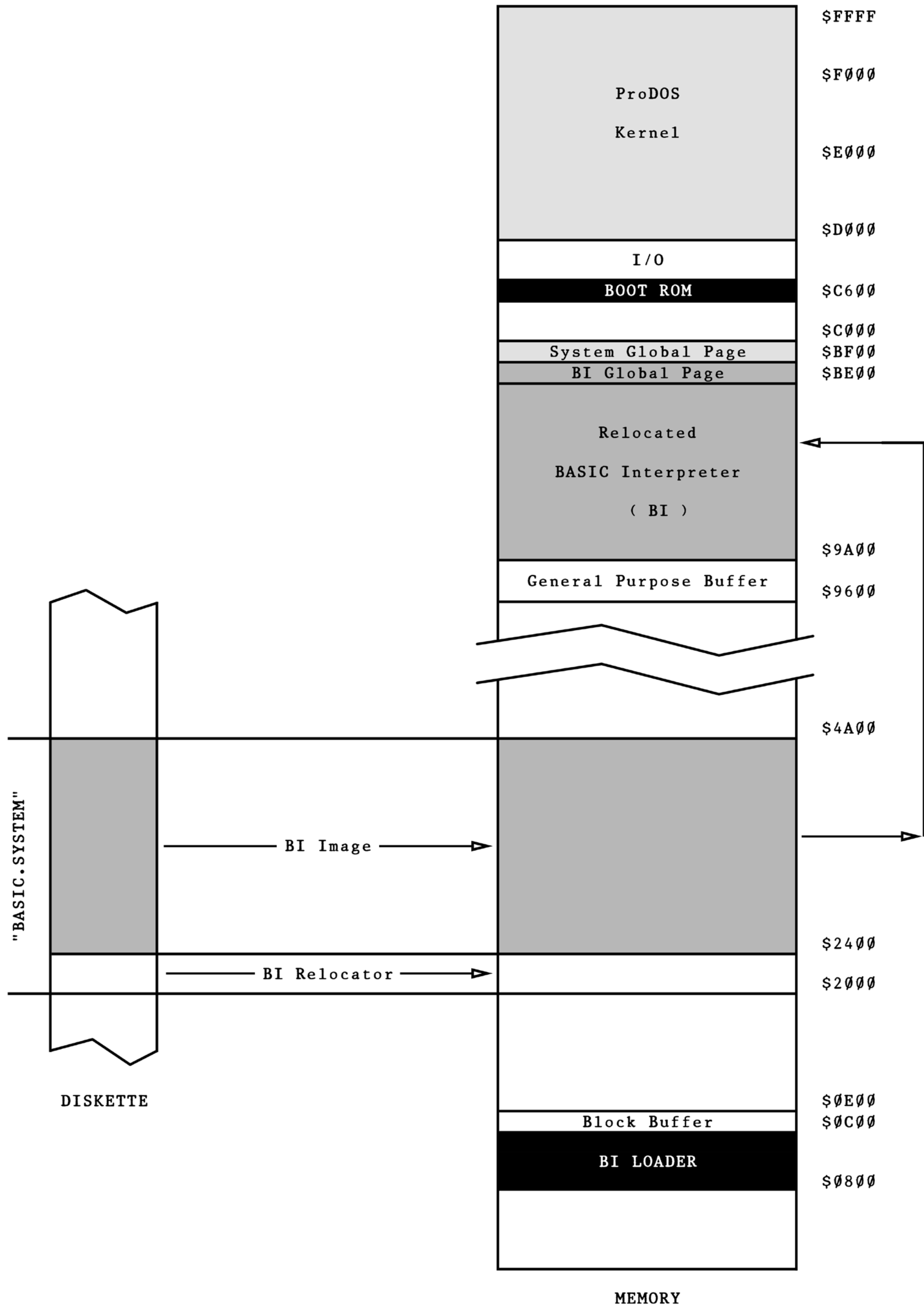


Figure 5.3 Basic Interpreter (BI) Bootstrap Process

and, after completing the vectors in page 3 (`$3FØ` etc.), control transfers to the BI through its vector at `$BEØØ`.

And then he said "HGR2:GR" and everything went haywire

Chapter 6

Using ProDOS From Assembly Language

CAVEAT

This chapter is aimed at the advanced assembly language programmer who wishes to access the disk at any level. Access to the disk by BASIC programs is well documented in the ProDOS manual, *BASIC Programming With ProDOS*. The material presented in this chapter may be beyond the comprehension (at least for the present) of a programmer who has never used assembly language.

Access to a diskette from assembly language may be accomplished at four different levels:

Level 0	Direct access of the diskette controller
Level 1	Block access
Level 2	Machine Language Interface (MLI) access
Level 3	BI command access

At the lowest level is direct access of the diskette controller. Here, data is accessed byte by byte. This may be required to implement diskette protection schemes or to perform low level diagnostics or correction of I/O errors. The next level of access is by ProDOS blocks (two sectors per block). This is done using the appropriate ProDOS device driver; in this case, the diskette device driver. At a higher level still is the ProDOS Machine Language Interface (MLI). Here, data may be accessed on a file basis. Finally, the highest level of access is through the ProDOS BASIC Interpreter. Here, entire ProDOS command lines may be executed to produce formatted directory listings and the like. A detailed description of the programming considerations at each of these levels follows.

DIRECT USE OF THE DISKETTE DRIVE

It is often desirable or necessary to access the Apple's disk drives directly from assembly language, without the use of ProDOS. Applications which might use direct disk access range from a user written operating system to

ProDOS-independent utility programs. Direct access is accomplished using 16 addresses that provide eight on/off switches which directly control the hardware. For information on the disk hardware, please refer to Appendix D. The device address assignments are given in Table 6.1.

	"OFF" Switches		"ON" Switches	
Switch	**Base Address**	**Function**	**Base Address**	**Function**
Q0	`$C080`	Phase 0 off	`$C081`	Phase 0 on
Q1	`$C082`	Phase 1 off	`$C083`	Phase 1 on
Q2	`$C084`	Phase 2 off	`$C085`	Phase 2 on
Q3	`$C086`	Phase 3 off	`$C087`	Phase 3 on
Q4	`$C088`	Drive off	`$C089`	Drive on
Q5	`$C08A`	Select drive 1	`$C08B`	Select drive 2
Q6	`$C08C`	Shift data register	`$C08D`	Load data register
Q7	`$C08E`	Read	`$C08F`	Write

Table 6.1 ProDOS Hardware Addresses

The last two switches are difficult to explain in single phrase definitions because they interact with each other forming a 4-way switch. The four possible settings are given in Table 6.2.

Q6	Q7	Function
Off	Off	Enable read sequencing.
Off	On	Shift data register every four cycles while writing.
On	Off	Check write protect and initialize sequencer for writing.
On	On	Load data register every four cycles while writing.

Table 6.2 Four Way Q6/Q7 Switches

The addresses are slot dependent and the offsets are computed by multiplying the slot number by 16. In hexadecimal this works out nicely. Simply add the value $s0 (where s is the slot number) to the base address. To engage disk drive number 1 in slot number 6, for example, we would add `$60` to `$C08A` (device address for engaging drive 1) for a result ot `$C0EA`. However, since it is generally desirable to write code that is not slot dependent, one would normally use `$C08A,X` (where the X-register contains the value `$s0`). Table 6.3 shows the range of addresses for each slot number.

Slot Number	Address Range
0	$C080-$C08F
1	$C090-$C09F
2	$C0A0-$C0AF
3	$C0B0-$C0BF
4	$C0C0-$C0CF
5	$C0D0-$C0DF
6	$C0E0-$C0EF

Table 6.3 Address Ranges For Slots

In general, the above addresses need only be accessed with any valid 6502 instruction. However, in the case of reading and writing bytes (last four addresses), care must be taken to insure that the data will be in an appropriate register. All of the following would engage drive number 1. (Assume slot number 6.)

```
BIT $C0EA
LDA $C08A,X      (where X-register contains $60)
CMP $C08A,X      (where X-register contains $60)
```

Below are typical examples demonstrating the use of the device address assignments. For more examples, see Appendix A. All examples assume that the label SLOT is set to 16 times the desired slot number (e.g. `$60` for slot 6).

STEPPER PHASE OFF OR ON

Basically, each of the four phases (0-3) must be turned on and then off again. Done in ascending order moves the arm inward. In descending order, the arm moves outward. For optimum performance, the timing between accesses to these locations is critical, making this a nontrivial exercise. An example is provided in Appendix A demonstrating how to move the arm to a given location.

MOTOR OFF OR ON

```
LDX #SLOT        Put slot number times 16 in X-register.
LDA $C088,X      Turn motor off.
LDX #SLOT        Put slot number times 16 in X-register.
LDA $C089,X      Turn motor on (selected drive).
```

Note: A sufficient delay should be provided to allow the motor time to come up to speed before reading or writing to the disk. Either a specific delay or a routine that watches the data register can be used. See Appendix A for an example.

ENGAGE DRIVE 1 OR 2

```
        LDX #SLOT       Put slot number times 16 in X-register.
        LDA $C08A,X     Engage drive 1.
        LDX #SLOT       Put slot number times 16 in X-register.
        LDA $C08B,X     Engage drive 2.
```

READ A BYTE

```
        LDX #SLOT       Put slot number times 16 in X-register.
        LDA $C08E,X     Insure read mode.
READ    LDA $C08C,X     Put contents of data register in Accumulator.
        BPL READ        Loop until the high bit is set.
```

Note: `$C08E,X` must be accessed to assure Read mode. The loop is necessary to assure that the accumulator will contain valid data. If the data register does not yet contain valid data, the high bit will be zero.

SENSE WRITE PROTECT

```
        LDX #SLOT       Put slot number times 16 in X-register.
        LDA $C08D,X
        LDA $C08E,X     Sense write protect.
        BMI ERROR       If high bit set, protected.
```

WRITE LOAD AND WRITE A BYTE

```
        LDX #SLOT       Put slot number times 16 in X-register.
        LDA DATA        Load accumulator with byte to write.
        STA $C08D,X     Write load.
        ORA $C08C,X     Write byte.
```

Note: `$C08F,X` must already have been accessed to insure Write mode and a 100-microsecond delay should be invoked before writing.

Due to hardware constraints, normal data bytes must be written in 32-cycle loops. The example below writes the two bytes `$D5` and `$AA` to the disk. It does this by an immediate load of the accumulator, followed by a subroutine call (`WRITE9`) that writes the bytes in the accumulator. Timing is so critical that different routines may be necessary, depending on how the data is to

be accessed, and code cannot cross memory page boundaries without an adjustment.

```
           LDA #$D5        Load byte to write. (2 cycles)
           JSR WRITE9      Go write it. (6)
           LDA #$AA        Load byte to write. (2)
           JSR WRITE9      Go write it. (6)
           ...
WRITE9     CLC             Provide different (2)
WRITE7     PHA             delays to produce (3)
           PLA             correct timing (4)
WRITE      STA $CØ8D,X     Store byte in register. (5)
           ORA $CØ8C,X     Write byte. (4)
           RTS             Return to caller. (6)
```

CALLING A STORAGE DEVICE DRIVER (BLOCK ACCESS)

ProDOS is device independent in that it requires a device driver for all storage devices. ProDOS comes with two device drivers built in. One supports the standard Apple floppy disk drive (Disk II or equivalent). The other supports a RAM drive on the Apple IIc or an Apple IIe that has 128K of memory. ProDOS can also support the ProFile hard disk which has its device driver on ROM. It seems clear that there will be many kinds of storage devices available in the future, each with its own driver.

These device drivers are used as subroutines by the MLI and provide the means of accessing the appropriate device. Four basic functions are currently defined for a device driver. They are **STATUS**, **READ**, **WRITE**, and **FORMAT**. However, not all device drivers will provide all four functions. The Disk II Device Driver, for example, does not support `FORMAT`: because of space constraints, this function is provided in the program named `FILER`.

The `READ_BLOCK` and `WRITE_BLOCK` calls in the MLI provide the only means of using a device driver from ProDOS and is the preferred method. While it is not generally recommended, any device driver can be called directly. This could prove useful in particular applications that don't require the MLI. Great care should be taken when calling the device driver directly because doing so can easily destroy data on the particular storage device.

While the parameters to call a device driver are quite straightforward, there are several potential difficulties to consider. First, RAM based device drivers normally reside in bank-switched memory, and therefore must be carefully selected and deselected. Second, a request for an unsupported device function may produce undesirable results.

There are four inputs stored in six zero page locations that must contain the appropriate information when a call is made to a device driver. The first input is the **Command Code**, which indicates which operation is requested.

As mentioned earlier, four operations are currently defined. The first of these is `STATUS`, which is used to determine if the device is ready to be accessed (either Read or Write). Although not all device drivers do so, it is suggested that the number of blocks the device supports be returned, in addition to the status. This should be done using the X (low bytes) and Y (high byte) registers. The remaining operations are quite straightforward—`READ` for reading a block, `WRITE` for writing a block, and `FORMAT` to format or initialize the media.

The second input value is the **Unit Number**, indicating in which slot and drive the desired device resides. Only two drives per slot are supported directly, but it is possible to interface a controller card that supports additional drives or volumes.

The third input is a 2-byte **Buffer Pointer** that indicates the location of a 512-byte area for data transfer. The MLI verifies that no memory conflicts exist, but most device drivers will not do so; therefore, some degree of care should be exercised in determining this input.

The fourth input is a 2-byte **Block Number** indicating which block is to be used for data transfer. The value should be in keeping with the number of blocks available on the desired device.

The four inputs necessary are listed in Table 6.4.

Although Apple has defined the manner in which device drivers are to be called, some variations will occur. Even the drivers provided by Apple vary slightly from one another. For this reason, it is advisable to make calls to any device driver with great caution. The parameter list descriptions that follow detail the four kinds of calls that are available. Not all device drivers will support all four call types and a request to an unsupported call type could prove dangerous.

Location	Description	Options
$42	Command Code	$ØØ = STATUS $Ø1 = READ $Ø2 = WRITE $Ø3 = FORMAT
$43	Unit Number	DSSS0000 D=Drive number (0 = drive 1, 1 = drive 2); SSS = Slot number (0 to 7)
$44-$45	I/O Buffer	Can be $ØØØØ to $FFFF*
$46-$47	Block Number	Can be $ØØØØ to $FFFF
	Return Code	The processor CARRY flag is set upon return from the device driver if an error occurred. The ACCUMULATOR contains the return code. $ØØ = No errors $27 = I/O error $28 = No device connected $2B = Write protect error

Table 6.4 Device Driver Parameters — General Format

CALLING THE DISK II DEVICE DRIVER

Access to standard Apple floppy disk drives (Disk II or equivalent) is performed using the Disk II Device Driver provided with ProDOS. As mentioned above, the Disk II Device Driver does not support the `FORMAT` call. If such a request is made, it will be interpreted as a `WRITE` call, and serious problems may result. Formatting floppy disks is performed by a separate utility program called `FILER`.

The I/O buffer location is not checked for validity by the Disk II Device Driver. The block number must be in the range `$Ø-$117` or an error type `$27` (I/O error) will result.

The Disk II Device Driver performs the same `READ` and `WRITE` functions as the RWTS routines of DOS 3.3, but these routines have been substantially modified to decrease disk access time. A comparison of RWTS to the Disk II Device Driver is contained in *Understanding the Apple IIe* by Jim Sather (1985, Quality Software).

* Unlike the MLI, the device driver will not check the location's validity.

DEVICE DRIVER PARAMETER LISTS BY COMMAND CODE

$00 STATUS REQUEST

FUNCTION This call returns the status of a particular device and is generally used to determine if a device is present, and if so, whether it is write protected. Additionally, some drivers will return the number of blocks supported by that device.

REQUIRED INPUTS

`$42`	Must be `$ØØ`.
`$43`	Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000 where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1-7).
`$44-$45`	Unused.
`$46-$47`	Unused but sometimes checked for validity (use `$ØØØØ`).

RETURNED VALUES

Carry Flag	Clear—No error occurred
	Set—Error occurred (see Accumulator for type)
Accumulator	`$ØØ`—No error occurred
	`$27`—I/O error or bad block number
	`$28`—No device connected to unit
	`$2B`—Disk is write protected
X-register	Blocks available (low byte).
Y-register	Blocks available (high byte).

$01 READ REQUEST

FUNCTION This call will read a 512-byte block and store it at the specified memory location. Most drivers will not check the memory location, so some care is suggested.

REQUIRED INPUTS

`$42`	Must be `$Ø1`.
`$43`	Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000 where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1-7).
`$44-$45`	Address (LO/HI) of the caller's 512-byte buffer into which the block will be read. The buffer need not be page aligned.
`$46-$47`	Block number (LO/HI) to read. Must be valid for the device being called.

RETURNED VALUES

Carry Flag — Clear—No error occurred
Set—Error occurred (see Accumulator for type)

Accumulator — $ØØ—No error occurred
$27—I/O error or bad block number
$28—No device connected to unit

$02 WRITE REQUEST

FUNCTION This call will write a 512-byte block from the specified memory location. Since all write operations could potentially destroy data, care is suggested.

REQUIRED INPUTS

$42 — Must be $Ø2.

$43 — Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000 where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1-7).

$44-$45 — Address (LO/HI) of the caller's 512-byte buffer which the block will be written from. The buffer need not be page aligned.

$46-$47 — Block number (LO/HI) to read. Must be valid for the device being called.

RETURNED VALUES

Carry Flag — Clear—No error occurred
Set—Error occurred (see Accumulator for type)

Accumulator — $ØØ—No error occurred
$27—I/O error or bad block number
$28—No device connected to unit
$2B—Disk is write protected.

$03 FORMAT REQUEST

FUNCTION This call will format the media present in the specified device. Since all data will be destroyed, extreme care is suggested.

REQUIRED INPUTS

$42 — Must be $Ø3.

$43 — Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000 where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1-7).

RETURNED VALUES

Carry Flag — Clear—No error occurred

	Set—Error occurred (see Accumulator for type)
Accumulator	`$ØØ`—No error occurred
	`$27`—I/O error or bad block number
	`$28`—No device connected to unit
	`$2B`—Disk is write protected

CALLING THE MACHINE LANGUAGE INTERFACE

The **Machine Language Interface** (MLI) consists of a set of externally callable subroutines in the ProDOS Kernel. Over 20 different functions may be performed to access and manipulate files in a device independent manner (i.e. the programmer need not be concerned with whether the device is a diskette drive or a hard disk). To avoid duplication of code and to eliminate direct calls to unpublished entry points within ProDOS, it is recommended that all file access be performed using the standardized ProDOS Machine Language Interface.

All calls to the MLI are made through the System Global Page at `$BFØØ`. The first item in this page is a `JMP` (GOTO) to the MLI. Thus, to call the MLI, code the following:

```
JSR $BFØØ
DFB function_code
DW  addr_of_parms
```

where "`function_code`" should be replaced with a 1-byte hexadecimal code representing the function you want to perform, and "`addr_of_parms`" is the 2-byte address of a parameter list you have created in your program's memory which indicates such things as the file name being accessed, the record number to access, etc. Note that programming reentrant or "ROMable" code or routines that cannot have instructions mixed with data will be made more difficult by this convention. In these cases, it may be advisable to move the `JSR $BFØØ`, the three bytes following, and a `RTS` instruction to a RAM data area and call them there.

Upon return, the processor Carry flag will be set if an error has occurred, and the return code will be placed in the A register. All other registers are saved and restored by the MLI. The valid function_codes are summarized in Table 6.5. It is interesting to note that most of the function calls are identical between ProDOS and the Apple III SOS operating system. The names used

are the standardized labels for these functions established by Apple for SOS and ProDOS.

Table 6.5 MLI Functions

Code	Name and Description
$4Ø	ALLOC_INTERRUPT Install interrupt handler
$41	DEALLOC_INTERRUPT Remove interrupt handler
$65	QUIT Exit from one Interpreter and dispatch another
$8Ø	READ_BLOCK Read disk block by unit number
$81	WRITE_BLOCK Write disk block by unit number
$82	GET_TIME Read calendar/clock peripheral card and set system date/time
$CØ	CREATE Create a new file or directory
$C1	DESTROY Delete a file or directory
$C2	RENAME Rename a file or directory
$C3	SET_FILE_INFO Change a file's attributes
$C4	GET_FILE_INFO Return a file's attributes
$C5	ONLINE Return names of one or all online volumes
$C6	SET_PREFIX Change default pathname prefix
$C7	GET_PREFIX Return default pathname prefix
$C8	OPEN Open a file
$C9	NEWLINE

Code	Name and Description
	Set end-of-line character for line-by-line reads
`$CA`	`READ` Read one or more bytes from an open file
`$CB`	`WRITE` Write one or more bytes to an open file
`$CC`	`CLOSE` Close one or more open files, flushing buffers
`$CD`	`FLUSH` Flush all write buffers for one or more files
`$CE`	`SET_MARK` Change File Position within an open file
`$CF`	`GET_MARK` Return File Position within an open file
`$DØ`	`SET_EOF` Change end-of-file position of an open file
`$D1`	`GET_EOF` Return end-of-file position of an open file
`$D2`	`SET_BUF` Change File Buffer's address for an open file
`$D3`	`GET_BUF` Return File Buffers address for an open file

The general form for a parameter list is as follows:

```
+Ø  PARAMETER_COUNT
+1  one or more parameters
```

The `PARAMETER_COUNT` is a 1-byte count of the number of parameters which follow. It is used by the MLI to validity check the parameter list to make sure that the address following the caller's `JSR` to the MLI really points to a valid parameter list.

MLI PARAMETER LISTS BY FUNCTION CODE

$40 ALLOC_INTERRUPT
INSTALL INTERRUPT HANDLER

FUNCTION

This function allows the user to install a custom interrupt handling routine into the ProDOS table. The user's handler resides in memory outside ProDOS, and only its entry point address is stored in the System Global Page table by this MLI call. Up to four such routines may be installed at any time. When a maskable interrupt (IRQ) occurs, ProDOS calls each handler in the order in which they were installed to allow the interrupt to be serviced. (See Chapter 7 for more information about writing interrupt handlers.)

PARAMETER LIST FORMAT

+0	`$Ø2`
+1	Priority
+2/+3	Address of Handler

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+2/+3	Address (LO/HI format) of user-written interrupt handling routine.

RETURNED VALUES

+1	Priority assigned to this handler by ProDOS: 1, 2, 3 or 4. This is the handler's position in the calling sequence. It is assigned the highest priority (earliest position) available.
Return Code	`$ØØ`—No errors
	`$Ø4`—Parameter count is not `$Ø2`
	`$25`—Interrupt handler table full (4 are installed)
	`$53`—Invalid parameter in list (address is zero)

$41 DEALLOC_INTERRUPT
REMOVE INTERRUPT HANDLER

FUNCTION

This function removes a previously installed interrupt handling routine's address from the ProDOS table.

PARAMETER LIST FORMAT

+0	`$Ø1`
+1	Priority

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).

+1	Priority of handler to be removed (1, 2, 3, or 4) as returned by MLI call `$4Ø` when it was installed.

RETURNED VALUES

Return Code	`$ØØ` — No errors
	`$Ø4` — Parameter count is not `$Ø1`
	`$53` — Invalid parameter in list (PRIORITY is not 1, 2, 3, or 4)

$65 QUIT
EXIT FROM ONE INTERPRETER, DISPATCH ANOTHER

FUNCTION This function causes the MLI to move three pages of code from `$D1ØØ` in the alternate 4K of the Language card to `$1ØØØ` and branch to it. This code frees any memory allocated by the interpreter in the System Memory Bit Map in the System Global Page, and then prompts the user for the name of a new Interpreter (System Program) to be executed. It then loads the new Interpreter and executes it. For more information on this call and on writing an Interpreter, see Chapter 7.

PARAMETER LIST FORMAT

+0	`$Ø4`
+1	Reserved
+2/+3	Reserved
+4	Reserved
+5/+6	Reserved

REQUIRED INPUTS

+0	Parameter-count (4 parameters in list).
+1 — +6	All other fields in the parameter list are reserved for future use. They must be present and they must be initialized to zeroes.

RETURNED VALUES

Return Code	`$Ø4` — Parameter count is not `$Ø4`

$80 READ_BLOCK
READ DISK BLOCK BY UNIT NUMBER

FUNCTION

This function calls the device handler for a given unit to read a 512-byte disk block. Calling this function is essentially the same as calling the device driver directly with the following additional actions: the buffer memory is validity checked for prior use; interrupts are disabled prior to the call to the driver; the unit number is validity checked and mapped into the appropriate device driver's address; the bank switched memory (language card) is enabled prior to the call and restored to its previous condition when the call completes. For these reasons, it is recommended that all block I/O be performed through the `READ_BLOCK` and `WRITE_BLOCK` MLI calls rather than calling the drivers directly. Direct calls are only recommended when the application will not be using the ProDOS Kernel and only the driver itself is available in memory.

PARAMETER LIST FORMAT

+0	`$Ø3`
+1	Unit Number
+2/+3	Address of Data Buffer
+4/+5	Block Number

REQUIRED INPUTS

+0	Parameter count (3 parameters in list).
+1	Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000, where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1 through 7).
+2/+3	Address (LO/HI) of the caller's 512-byte buffer into which the block will be read. The buffer need not be page aligned.
+4/+5	Block number (LO/HI) to read. This may range from `$ØØØØ` to `$Ø117` for a diskette. The validity of this number is checked by the driver itself.

RETURNED VALUES

Return Code

`$ØØ`—No errors

`$Ø4`—Parameter count is not `$Ø3`

`$27`—I/O error or bad block number

`$28`—No device connected to unit

`$56`—Bad buffer (already in use by ProDOS)

$81 WRITE_BLOCK
WRITE DISK BLOCK BY UNIT NUMBER

FUNCTION

This function calls the device handler for a given unit to write a 512-byte disk block. Calling this function is essentially the same as calling the device driver directly with the following additional actions: the buffer memory is validity checked for prior use: interrupts are disabled prior to the call to the driver; the unit number is validity checked and mapped into the appropriate device driver's address; the bank switched memory (language card) is enabled prior to the call and restored to its previous condition when the call completes. For these reasons, it is recommended that all block I/O be performed through the `READ_BLOCK` and `WRITE_BLOCK` MLI calls rather than calling the drivers directly. Direct calls are only recommended when the application will not be using the ProDOS Kernel and only the driver itself is available in memory.

PARAMETER LIST FORMAT

+0	`$Ø3`
+1	Unit Number
+2/+3	Address of Data Buffer
+4/+5	Block Number

REQUIRED INPUTS

+0	Parameter count (3 parameters in list).
+1	Unit number of disk to be accessed. The bit assignment of a ProDOS unit number is as follows: DSSS0000, where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1 through 7).
+2/+3	Address (LO/HI) of the caller's 512-byte buffer from which the block will be written. The buffer need not be page aligned.
+4/+5	Block number (LO/HI) to write. This may range from `$ØØØØ` to `$Ø117` for a diskette. The validity of this number is checked by the driver itself.

RETURNED VALUES

Return Code

`$ØØ`—No errors

`$Ø4`—Parameter count is not `$Ø3`

`$27`—I/O error or bad block number

`$28`—No device connected to unit

`$2B`—Disk is write protected

`$56`—Bad buffer (already in use by ProDOS)

$82 GET_TIME
READ CALENDAR/CLOCK PERIPHERAL CARD

FUNCTION

This function accesses any calendar/clock card which might be in the system and sets the system date and time in the System Global Page. If no calendar/clock handler has been installed (DATETIME vector in the System Global Page), the call is ignored.

PARAMETER LIST

None — (parameter list address following `JSR` is `$ØØØØ`)

REQUIRED INPUTS

None

RETURNED VALUES

`$BF9Ø/$BF91`	System Global Page date field is filled in. Its format is (LO/HI): `YYYYYYYM MMMDDDDD` where `YYYYYYY` is the year (offset from 1900), `MMMM` is the month (1 through 12), and `DDDDD` is the day.
`$BF92/$BF93`	System Global Page time field is filled in. Its format is (LO/HI): `HHHHHHHH MMMMMMMM` where `HHHHHHHH` is the hour since midnight and `MMMMMMMM` is the minute (0 through 59).
Return Code	`$ØØ`—No errors

$C0 CREATE
CREATE A NEW FILE OR DIRECTORY

FUNCTION

This function creates a new file (either a data file or a directory file). One 512-byte block of disk space is allocated to the new file. The file may not already exist. If it is desirable to recreate an old file, issue the `DESTROY` call first. If the pathname given indicates that the file's directory entry will be in a subdirectory and there are no free directory entries there, the subdirectory will be extended by one block. The Volume Directory may not be extended. If the new file is a directory file, a directory header is created and written to the key block.

PARAMETER LIST FORMAT

+0	`$Ø7`
+1/+2	Address of Pathname
+3	Access Bits
+4	File Type
+5/+6	Auxiliary File Type
+7	Storage Type

+8/+9 Creation Date

+A/+B Creation Time

REQUIRED INPUTS

+0 Parameter count (7 parameters in list).

+1/+2 Address (LO/HI) of pathname buffer for file to be created. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a "/", the name is considered to be fully qualified. If not. the current default prefix is added to the name by ProDOS when the file is created.

+3 Access privileges associated with this file. The access bits are: DNBXXXWR (high bit to low bit) where...

D	(bit 7) if 1 allows the file to be `DESTROY`ed.
N	(bit 6) if 1 allows the file to be `RENAME`d.
B	(bit 5) if 1 indicates file needs backing up.
X	(bits 4, 3, and 2) are reserved for future use.
W	(bit 1) if 1 allows the file to be written.
R	(bit 0) if 1 allows the file to be read.

Full access is `$C3`. A file is "locked" in the BASIC interpreter sense if the D, N, W and R bits are all zeroes. It is unlocked if they are all ones. The B bit is forced to one when the file is created. WARNING: It is possible to set the "X" reserved bits to ones with this call since no validity check is made by the MLI on `CREATE` (a check is made for `SET_FILE _INFO`, however).

+4 Type of data stored in the file. Commonly supported file types are:

`$Ø1`	`BAD`	File containing bad blocks.
`$Ø4`	`TXT`	File containing ASCII text (BASIC data file).
`$Ø6`	`BIN`	File containing a binary memory image or machine language program.
`$ØF`	`DIR`	File is a directory.
`$19`	`ADB`	AppleWorks data base file
`$1A`	`AWP`	AppleWorks word processing file
`$1B`	`ASF`	AppleWorks spread sheet file
`$FØ`	`CMD`	ProDOS added command file.
`$F1-$F8`		User defined file types.
`$FC`	`BAS`	File contains an Applesoft program.

$FD	VAR	File contains Applesoft variables (STORE/RESTORE).
$FE	REL	File contains a relocatable object module (EDASM).
$FF	SYS	File contains a ProDOS system program.

Other less commonly used file types are defined in Appendix E. Assignment of a file type is a convention which serves to inform the program which accesses a file what data format it should expect to find there. You are not prevented from storing binary data in a TXT file or ASCII text in a BIN file, but this runs counter to convention and is discouraged.

+5/+6 Auxiliary data pertaining to the file. Its usage is defined according to its file type above. The current uses of this field by the BI are:

TXT	contains the default record length (LO/HI).
BIN	contains the address (LO/HI) at which to load the image.
BAS	contains the address (LO/HI) of the BASIC program image.
VAR	contains the address (LO/HI) of the BASIC variables image.
SYS	contains $2ØØØ (LO/HI), the load address for system files.

+7 Storage type or type of file organization. If this byte contains $ØD, the file is a linked subdirectory file. If it is $Ø1, it is a standard seedling file (at the time of its creation). Other values are reserved for future use. If a value of $ØØ, $Ø2, or $Ø3 is given, $Ø1 is assumed. All values other than $ØØ-$Ø3 or $ØD will result in an error.

+8/+9 Date of creation. If this field is set to zero, the MLI uses the current system date (if any). If this field is non-zero, it is the creation date in the (LO/H1) form YYYYYYYM MMMDDDDD where YYYYYYY is the year past 1900, MMMM is the month (1-12) and DDDDD is the day of the month.

+A/+B Time of creation. If this field is set to zero, the MLI uses the current system time (if any). If this field is non-zero, it is the creation time in the (LO/HI) form HHHHHHHH MMMMMMMM where HHHHHHHH is the hour past midnight and MMMMMMMM is the minute within the hour.

RETURNED VALUES

Return Code $ØØ — No errors

$Ø4 — Parameter count is not $Ø7

$27—I/O error
$2B—Disk is write protected
$4Ø—Pathname has invalid syntax
$44—Path to file's subdirectory is bad
$45—Volume directory not found
$47—Duplicate file name already in use
$48—Disk full
$49—Volume directory full
$4B—Bad storage type (use only $ØD or $Ø1)
$53—Invalid parameter or address pointer
$5A—Damaged disk freespace bit map

$C1 DESTROY
DELETE A FILE OR DIRECTORY

FUNCTION

This function deletes a file or empty subdirectory. Open files may not be deleted. The Volume Directory may not be deleted. A subdirectory is considered "locked" if it contains any files at all, and may not be DESTROYed until all its files and subdirectories are DESTROYed.

PARAMETER LIST FORMAT

+0 $Ø1
+1/+2 Address of Pathname

REQUIRED INPUTS

+0 Parameter count (1 parameter in list).
+1/+2 Address (LO/HI) of pathname buffer for file to be deleted. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a "/", the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS.

RETURNED VALUES

Return Code
$ØØ—No errors
$Ø4—Parameter count is not $Ø1
$27—I/O error
$2B—Disk is write protected
$4Ø—Pathname has invalid syntax
$44—Path to file's subdirectory is bad
$45—Volume directory not found

$46 — File not found in specified directory
$4A — Incompatible file format
$4B — Bad storage type
$4E — Access refused: DESTROY bit not enabled or non-empty subdirectory
$5Ø — Access refused: File is currently open
$5A — Damaged disk freespace bit map

$C2 RENAME
RENAME A FILE OR DIRECTORY

FUNCTION

This function renames a file or subdirectory. Only the final name in the path specification may be renamed. This function will not rename multiple directories in a pathname specification (e.g. /project/myfile may not be renamed to /task/yourfile since this involves renaming something other than the final name in the pathname). RENAME will not create new subdirectories or move a file's entry from one directory to another (e.g. you may not rename /project/myfile to /project/another/myfile since this involves moving the file's entry to subdirectory "another"). A volume may be renamed if no files are currently opened for it. A file or subdirectory may be renamed if it is not open, or if it is a read-only file (WRITE access disabled). The new file name may not be the same as another in the same directory.

PARAMETER LIST FORMAT

+0	$Ø2
+1/+2	Address of Old Pathname
+3/+4	Address of New Pathname

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+1/+2	Address (LO/HI) of pathname buffer for file to be renamed. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a "/", the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS.

+3/+4 Address (LO/HI) of pathname buffer for the new name. The qualifying levels of the name, if any, should match those of the old pathname given at +1/+2. Only the final name should be different. The format of the new pathname buffer is identical to that of the old pathname buffer given above. The current default prefix, if any, will be added to a non-fully qualified pathname.

RETURNED VALUES

Return Code

$ØØ—No errors
$Ø4—Parameter count is not $Ø2
$27—I/O error
$2B—Disk is write protected
$4Ø—Pathname has invalid syntax
$44—Path to file's subdirectory is bad
$45—Volume directory not found
$46—File not found in specified directory
$47—New name duplicates one already in directory
$4A—Incompatible file format
$4B—Bad storage type
$4E—Access refused: RENAME bit not enabled
$5Ø—Access refused: File is currently open
$57—Two volumes are online with the same volume name

$C3 SET_FILE_INFO
CHANGE FILE ATTRIBUTES

FUNCTION This function changesthe attributes (e.g. file type, storage type, etc.) which are stored in the directory entry which describes a file. The file may be open or closed. SET_FILE_INFO will not act upon a Volume Directory (an error of $4Ø will result). Before issuing this function call, it is recommended that GET_FILE_INFO ($C4) be used to determine the current parameter settings for the file. (Note that the parameter lists for the two calls have a compatible format.)

PARAMETER LIST FORMAT

+0	$Ø7
+1/+2	Address of Pathname
+3	Access Bits
+4	File Type
+5/+6	Auxiliary File Type

+7	Not Used
+8/+9	Not Used
+A/+B	Date of Last Modification
+C/+D	Time of Last Modification

REQUIRED INPUTS

+0 Parameter count (7 parameters in list).

+1/+2 Address (LO/HI) of pathname buffer for file. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a "/", the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS.

+3 New access privileges to be associated with this file. The access bits are:

DNBXXXWR

(high bit to low bit) where...

D	(bit 7) if 1 allows the file to be `DESTROY`ed.
N	(bit 6) if 1 allows the file to be `RENAME`d.
B	(bit 5) if 1 indicates file needs backing up.
X	(bits 4, 3, and 2) are reserved for future use.
W	(bit 1) if 1 allows the file to be written.
R	(bit 0) if 1 allows the file to be read.

Full access is `$C3`. A file is "locked" in the BASIC interpreter sense if the D, N, W and R bits are all zeroes. It is unlocked if they are all ones. Note that a "locked" file is not protected against `SET_FILE_INFO` (how else would one unlock it?). If an attempt is made to use the "X" reserved bits, an error will occur. They should be set to zeroes.

+4 Type of data stored in the file. Commonly supported file types are:

`$Ø1`	`BAD`	File containing bad blocks.
`$Ø4`	`TXT`	File containing ASCII text (BASIC data file).
`$Ø6`	`BIN`	File containing a binary memory image or machine language program.
`$ØF`	`DIR`	File is a directory.
`$19`	`ADB`	AppleWorks data base file.
`$1A`	`AWP`	AppleWorks word processing file.
`$1B`	`ASF`	AppleWorks spread sheet file.
`$FØ`	`CMD`	ProDOS added command file.
`$F1-$F8`		User defined file types.

$FC	BAS	File contains an Applesoft program.
$FD	VAR	File contains Applesoft variables (STORE/RESTORE).
$FE	REL	File contains a relocatable object module (EDASM).
$FF	SYS	File contains a ProDOS system program.

Other less commonly used file types are defined in Appendix E. Assignment of a file type is a convention which serves to inform the program which accesses a file what data format it should expect to find there. You are not prevented from storing binary data in a TXT file or ASCII text in a BIN file, but this runs counter to convention and is discouraged.

+5/+6 Auxiliary data pertaining to the file. Its usage is defined according to its file type above. The current uses of this field by the BI are:

- TXT contains the default record length (LO/HI).
- BIN contains the address (LO/HI) at which to load the image.
- BAS contains the address (LO/HI) of the BASIC program image.
- VAR contains the address (LO/HI) of the BASIC variables image.
- SYS contains $2000 (LO/HI), the load address for system files.

+7 Ignored. May be set to zero.

+8/+9 Ignored. May be set to zero.

+A/+B Date of last modification. If this field is set to zero, the MLI uses the current system date (if any). If this field is non-zero, it is the modification date in the (LO/HI) form YYYYYYYM MMMDDDDD where YYYYYYY is the year past 1900, MMMM is the month (1-12) and DDDDD is the day of the month.

+C/+D` Time of last modification. If this field is set to zero, the MLI uses the current system time (if any). If this field is non-zero, it is the modification time in the (LO/HI) form HHHHHHHH MMMMMMMM where HHHHHHHH is the hour past midnight and MMMMMMMM is the minute within the hour.

RETURNED VALUES

Return Code $ØØ—No errors

$Ø4—Parameter count is not $Ø7

$27—I/O error

`$2B`—Disk is write protected
`$4Ø`—Pathname has invalid syntax
`$44`—Path to file's subdirectory is bad
`$45`—Volume directory not found
`$46`—File not found in specified directory
`$4A`—Incompatible file format
`$4B`—Bad storage type
`$4E`—Access refused: Reserved access bits were used
`$53`—Parameter value out of range
`$5A`—Damaged disk freespace bit map

$C4 GET_FILE_INFO RETURN FILE ATTRIBUTES

FUNCTION

This function reads the attributes (e.g. file type, storage type, etc.), which describe the file and are stored in the directory entry, and returns them in the parameter list provided by the caller. The file may be open or closed. If information about a Volume Directory is requested, the size of the volume in blocks and the blocks in use count are also returned.

PARAMETER LIST FORMAT

+0	`$ØA`
+1/+2	Address of Pathname
+3	Access Bits
+4	File Type
+5/+6	Auxiliary File Type
+7	Storage Type
+8/+9	Blocks Used
+A/+B	Date of Last Modification
+C/+D	Time of Last Modification
+E/+F	Creation Date
+10/+11	Creation Time

REQUIRED INPUTS

+0 Parameter count (`$A` parameters in list).

+1/+2 Address (LO/HI) of pathname buffer for file. The pathname buffer consists of a 1-byte length followed by 1 to 63 characters of name. If the first character is a "`/`", the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS.

+3 New access privileges to be associated with this file. The access bits are:

DNBXXXWR

(high bit to low bit) where...

D	(bit 7) if 1 allows the file to be `DESTROY`ed.
N	(bit 6) if 1 allows the file to be `RENAME`d.
B	(bit 5) if 1 indicates file needs backing up.
X	(bits 4, 3, and 2) are reserved for future use.
W	(bit 1) if 1 allows the file to be written.
R	(bit 0) if 1 allows the file to be read.

Full access is `$C3`. A file is "locked" in the BASIC interpreter sense if the D, N, W and R bits are all zeroes. It is unlocked if they are all ones.

+4 Type of data stored in the file. Commonly supported file types are:

`$Ø1`	`BAD`	File containing bad blocks.
`$Ø4`	`TXT`	File containing ASCII text (BASIC data file).
`$Ø6`	`BIN`	File containing a binary memory image or machine language program.
`$ØF`	`DIR`	File is a directory.
`$19`	`ADB`	AppleWorks data base file.
`$1A`	`AWP`	AppleWorks word processing file.
`$1B`	`ASF`	AppleWorks spread sheet file.
`$FØ`	`CMD`	ProDOS added command file.
`$F1-$F8`		User defined file types.
`$FC`	`BAS`	File contains an Applesoft program.
`$FD`	`VAR`	File contains Applesoft variables (`STORE/RESTORE`).
`$FE`	`REL`	File contains a relocatable object module (EDASM).
`$FF`	`SYS`	File contains a ProDOS system program.

Other less commonly used file types are defined in Appendix E. Assignment of a file type is a convention which serves to inform the program which accesses a file what data format it should expect to find there. You are not prevented from storing binary data in a `TXT` file or ASCII text in a `BIN` file, but this runs counter to convention and is discouraged.

+5/+6	Auxiliary data pertaining to the file. Its usage is defined according to its file type above. The current uses of this field by the BI are: `TXT` contains the default record length (LO/HI). `BIN` contains the address (LO/HI) at which to load the image. `BAS` contains the address (LO/HI) of the BASIC program image. `VAR` contains the address (LO/HI) of the BASIC variables image. `SYS` contains `$2ØØØ` (LO/HI), the load address for system files. If the `GET_FILE_INFO` request is for the Volume Directory; this field contains the size of this volume in blocks
+7	Storage type or type of file organization. Currently supported storage types are: `$ØD` Linked directory file `$Ø1` Seedling file (no index blocks) `$Ø2` Sapling file (one index level) `$Ø3` Tree file (two index levels) Other values are reserved for future use.
+8/+9	Number of 512-byte disk blocks in use by file including index blocks and data blocks. If the `GET_FILE_INFO` call is made on the volume itself (Volume Directory), this field contains the total number of disk blocks in use on the volume (including system overhead).
+A/+B	Date of last modification. If this field is set to zero, the MLI uses the current system date (if any). If this field is non-zero, it is the modification date in the (LO/HI) form `YYYYYYYM MMMDDDDD` where `YYYYYYY` is the year past 1900, `MMMM` is the month (1-12) and `DDDDD` is the day of the month.
+C/+D	Time of last modification. If this field is set to zero, the MLI uses the current system time (if any). If this field is non-zero, it is the modification time in the (LO/HI) form `HHHHHHHH MMMMMMMM` where `HHHHHHHH` is the hour past midnight and `MMMMMMMM` is the minute within the hour.
+E/+F	Date of file's creation. If this field is non-zero, it is the creation date in the (LO/ HI) form `YYYYYYYM MMMDDDDD` where `YYYYYYY` is the year past 1900, `MMMM` is the month (1-12) and `DDDDD` is the day of the month.

+10/+11 — Time of file's creation. If this field is non-zero, it is the creation time in the (LO/HI) form `HHHHHHHH MMMMMMMM` where `HHHHHHHH` is the hour past midnight and `MMMMMMMM` is the minute within the hour.

RETURNED VALUES

Return Code
- `$ØØ`—No errors
- `$Ø4`—Parameter count is not `$ØA`
- `$27`—I/O error
- `$4Ø`—Pathname has invalid syntax
- `$44`—Path to file's subdirectory is bad
- `$45`—Volume directory not found
- `$46`—File not found in specified directory
- `$4A`—Incompatible file format
- `$4B`—Bad storage type
- `$53`—Parameter value out of range
- `$5A`—Damaged disk freespace bit map

$C5 ONLINE
RETURN NAMES OF ONE OR ALL ONLINE VOLUMES

FUNCTION — This functlon examlnes all mounted disk volumes and returns their names in the buffer provlded by the caller. If a single volume is to be identified, the caller must provide a specific unit number (slot and drive).

PARAMETER LIST FORMAT

+0 — `$Ø2`
+1 — Unit Number
+2/+3 — Address of Data Buffer

REQUIRED INPUTS

+0 — Parameter count (2 parameters in list)

+1 — Unit number of specific device to be examined. If all online volumes are to be identified, set this field to zero. The bit assignment for a specific unit number is DSSS0000, where D is the drive number (0 = drive 1, 1 = drive 2) and SSS is the slot number (1 through 7)

+2/+3 — Address (LO/ HI) of a buffer to contain the volume names returned by ProDOS. If a specific unit is to be examined a 16-byte buffer must be provided. If the call is non-specific (UNIT = 0), then the buffer must be 256 bytes to allow for up to 16 online volumes.

RETURNED VALUES

Buffer

If the return code in the accumulator is zero, the caller's buffer will contain zero or more volume name entries of format described below. The volume names will be given in the order in which ProDOS searches for a volume, i.e., the boot volume first, followed by slot numbers lower than the boot slot, wrapping around to higher slots last.

Online Volume Entry

byte 0	DSSSLLLL: where D is the drive number (0=drive 1, 1=drive 2), SSS is the slot number (1 through 7), and LLLL is the length of the name which follows. If LLLL is zero, an error occurred in examining this volume. The return code is in the first byte of the name field. If byte 0 is zero, then there are no more volume entries in the buffer.
bytes 1-15	Volume name or 1-byte error code. No slash precedes the name.

Return Code

`$ØØ`—No errors

`$Ø4`—Parameter count is not `$Ø2`

`$55`—Volume Control Block full (too many open files)

`$56`—Bad buffer address (check system memory bit map)

The following error codes may appear for a specific unit in byte 1 of a buffer entry. If so, the return code above will be `$ØØ`.

`$27`—I/O error on this unit

`$28`—Device not connected (e.g. no drive 2)

`$2E`—Diskette switched while file was open

`$45`—Volume directory not found

`$52`—Not a ProDOS disk volume

`$57`—Duplicate volume—Byte 3 of buffer entry contains the unit number of the duplicate

$C6 SET_PREFIX
CHANGE DEFAULT PATHNAME PREFIX

FUNCTION This function changes the default prefix which is attached to any pathnames passed to the MLI which are not fully qualified (do not start with a slash "/"). The MLI follows the prefix given, locating each directory at each level of the prefix to make sure that they exist on a mounted volume.

PARAMETER LIST FORMAT

+0	`$Ø1`
+1/+2	Address of Pathname Buffer

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).
+1/+2	Address (LO/HI) of pathname buffer for the new prefix. The pathname buffer consists of one byte of length followed by 1 to 63 characters of name. If the first character is a "/", the name is considered to be fully qualified. If not, the old default prefix is added to the new one to form a completely qualified default prefix (for a total length of no more than 64 characters). The last name in the prefix must be that of a directory file. The prefix may be eliminated by specifying a null (0-length) prefix. An ending slash is assumed if it is omitted.

RETURNED VALUES

Return Code

`$ØØ`—No errors
`$Ø4`—Parameter count is not `$Ø1`
`$4Ø`—Pathname has invalid syntax or prefix too long
`$44`—Path to final subdirectory is bad
`$45`—Volume directory not found
`$46`—Final subdirectory file not found
`$4A`—Incompatible file format
`$4B`—Bad storage type
`$5A`—Damaged disk freespace bit map

$C7 GET_PREFIX
RETURN DEFAULT PATHNAME PREFIX

FUNCTION This function returns the default prefix, if any, to the caller's buffer.

PARAMETER LIST FORMAT

+0	`$Ø1`
+1/+2	Address of Pathname

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).
+1/+2	Address (LO/HI) of pathname buffer into which the MLI will copy the default prefix. The buffer must be at least 64 bytes long.

RETURNED VALUES

Buffer	The buffer will contain the current MLI default prefix. The prefix consists of one byte of length followed by up to 63 characters of prefix. If the length is zero, the prefix is null. Otherwise, the prefix starts and ends with a slash.
Return Code	`$ØØ`—No errors `$Ø4`—Parameter count is not `$Ø1` `$56`—Bad buffer address (check system memory bit map)

$C8 OPEN
OPEN A FILE

FUNCTION This function locates a file on a volume and sets up internal control blocks (a File Control Block—FCB, and a Volume Control Block—VCB) to allow the user to read or write it. A reference number (from 1 to 8) is assigned by the MLI to the open file for future identification. (The reference number uniquely identifies the FCB which is being used with the file.) The current position for reading or writing is set to zero (start of the file). At most, eight files may be open at one time. More than one `OPEN` may be issued to the same file if the file's access is `WRITE` disabled (read-only file).

Once a file is opened, it should always be closed (using the MLI `CLOSE` call). This is to permit the MLI to release the reference number for use by other `OPEN`s. In addition, the MLI keeps a count of the number of files which are open on a volume. If the diskette is switched while files are open, error return codes are produced.

A directory file may also be opened (for READs only). When accessing a directory, do not make assumptions about the length of an entry or the number of entries per block--use the fields in the directory header which are provided for this purpose. This will help to insure that your program will work for future releases of ProDOS. A directory file may be read only, not written.

PARAMETER LIST FORMAT

+0	$Ø3
+1/+2	Address of Pathname
+3/+4	Address of File Buffer
+5	Reference Number

REQUIRED INPUTS

+0	Parameter count (3 parameters in list).
+1/+2	Address (LO/HI) of pathname buffer for file. The pathname buffer consists of one byte of length followed by 1 to 63 characters of name. If the first character is a "/", the name is considered to be fully qualified. If not, the current default prefix is added to the name by ProDOS.
+3/+4	Address (LO/HI) of a 1024-byte file buffer, provided by the caller in his memory, to be used by the MLI while the file is open. The buffer must begin on an even page boundary (LO portion of address must be zero). The MLI uses the buffer to hold the current data block and the current index block respectively. Its contents need not be intitialized by the caller. It should not be tampered with by the caller while the file remains open.
$BF94	The LEVEL byte in the System Global Page may be set to indicate the level of this OPEN. If a subsequent CLOSE is issued with a REF_NUM of zero, then all files of a given level or higher will be closed. This feature is handy in that it allows group CLOSEs on user-defined classes of files. Normally, LEVEL is set to zero.

RETURNED VALUES

+5	A reference number assigned to this open file by the MLI (from $Ø1 to $Ø8). The caller should make a note of this number and use it in all future references to this open file. A reference number is used to identify open files instead of the pathname since it is possible to maintain multiple "opens" on the same read-only file.
Return Code	$ØØ—No errors
	$Ø4—Parameter count is not $Ø3

$27 — I/O error
$4Ø — Pathname has invalid syntax
$42 — Eight files are already open
$44 — Path to file's subdirectory is bad
$45 — Volume directory not found
$46 — File not found in specified directory
$4B — Bad storage type
$5Ø — File already open (WRITE enabled)
$53 — Parameter value out of range (REF_NUM)
$56 — Bad buffer address (check system memory bit map)
$5A — Damaged disk freespace bit map

$C9 NEWLINE
SET END OF LINE CHARACTER

FUNCTION

A file may be read as either a continuous stream of bytes or as a collection of lines, terminated by "newline" characters (such as a RETURN character). When a file is first opened, the former assumption is made. To enable the line by line mode, the NEWLINE function may be invoked, specifying the end of line character to be used. All future READ operations on the specified open file will be terminated either when a newline character is detected, or when the read length is exhausted (or at end of file).

PARAMETER LIST FORMAT

+0	$Ø3
+1	Reference Number
+2	AND Mask
+3	Newline Character

REQUIRED INPUTS

+0	Parameter count (3 parameters in list).
+1	Reference number for an open file as returned by OPEN.

+2	`AND` mask. The value given here is logically `AND`ed with the contents of each byte read before a comparison is made with the `NEWLINE` character given in +3. If the `AND` mask is zero, then the line by line mode is disabled and the continuous byte stream mode is enabled. If a mask of `$FF` is given, the `NEWLINE` character must exactly match what is read. Other values for the `AND` mask allow "don't care" bits. For example. `$7F` allows the MSB to be either on or off without affecting the comparison (e.g., `$ØD` or `$8D` will both be treated as newline if `$ØD` is the `NEWLINE` character and the `AND` mask is `$7F`).
+3	The actual value of the `NEWLINE` character. Normally, when line by line mode is used, this is should be set to `$ØD`. Note that if the `AND` mask is `$ØØ`, this character is ignored (even if it is also `$ØØ`; if `$ØØ` is to be the newline character, set the `AND` mask to `$FF`).

RETURNED VALUES

Return Code

`$ØØ`—No errors

`$Ø4`—Parameter count is not `$Ø3`

`$43`—Invalid reference number

$CA READ
READ ONE OR MORE BYTES FROM AN OPEN FILE

FUNCTION

This function reads a number of bytes, starting at the current file position in an open file. The number of bytes read depends upon the length requested by the caller, whether or not a newline character has been set (see the `$C9 NEWLINE` function call), and whether the end of file is reached during the read. The current file position is updated to point to the byte following the last byte read.

In general, read operations will be much more efficient if the amount of data transferred exceeds a block (512 bytes). Special "direct read" code exists within the MLI to prevent "double buffering" and allow direct reads to the caller's buffer without going through the I/O buffer attached to the file. This fast access is only used when whole blocks may be read at a time. Use of the `NEWLINE` feature automatically disables "direct reads." (NOTE: It is this "direct read" feature which makes ProDOS I/O faster than Apple DOS.)

Note that, once a file is opened, no check is made by the MLI that the user has not switched diskette volumes in the drive. If this occurs, it is possible to read random portions of the new diskette volume! If the programmer is issuing a `READ` after a period of disk inactivity, it is recommended that a `ONLINE` call `$C5`) be issued to make sure that the same diskette is still in the drive.

PARAMETER LIST FORMAT

+0	`$Ø4`
+1	Reference Number
+2/+3	Address of Data Buffer
+4/+5	Requested Length
+6/+7	Actual Length

REQUIRED INPUTS

+0	Parameter count (4 parameters in list).
+1	Reference number for an open file as returned by `OPEN`.
+2/+3	Address (LO/HI) of a sufficiently large buffer provided by the caller into which the data will be read. This buffer should not be confused with the "file buffer" passed to `OPEN` which is separate, and should not be used by the caller's program.
+4/+5	Maximum number (LO/HI) of bytes of data to read. This is usually the size of the data buffer. If lines are being read, make sure this value is as large as the longest line, including the end of line character itself.

RETURNED VALUES

+6/+7	Actual number (LO/HI) of bytes placed in the caller's data buffer by the MLI. This value will differ from the requested length in +4/+5 if a newline character was found, if the end of the file was reached, or if an error occurred during the read operation. If a newline character terminated the read, this length will include the newline character itself. If the read began at the end of file position, this field is set to zero, and the end of file return code (`$4C`) is placed in the A register.
Return Code	`$ØØ`—No errors
	`$Ø4`—Parameter count is not `$Ø4`
	`$27`—I/O error
	`$43`—Invalid reference number
	`$4C`—At end of file, nothing was read
	`$4E`—Access refused: Read bit not enabled
	`$56`—Bad buffer address (check System memory bit map)
	`$5A`—Damaged disk freespace bit map

$CB WRITE
WRITE ONE OR MORE BYTES TO AN OPEN FILE

FUNCTION

This function writes a number of bytes to disk, starting at the current file position in an open file. You may not write to a directory. The current file position is updated to point to the byte following the last byte written. The end of file mark is moved if necessary, and new data and/or index blocks are allocated to the file as necessary. In the interest of efficiency, the data may or may not be written to disk at this time. As much as one block's worth (512 bytes) may remain in the file buffer to be written later when the block is filled, the file is closed or flushed, or when the file position is changed. For this reason, it is important to close all files before powering off the machine.

Note that, once a file is opened, no check is made by the MLI that the user has not switched diskette volumes in the drive. If this occurs, it is possible to Write on random portions of the new diskette volume! If the programmer is issuing a `WRITE` after a period of disk inactivity, it is recommended that a RETURN `ONLINE` volumes call (`$C5`) be issued to make sure that the same diskette is still in the drive.

Note that there is no "direct write" feature similar to the "direct read" feature described under the `READ` MLI call.

PARAMETER LIST FORMAT

+0	`$Ø4`
+1	Reference Number
+2/+3	Address of Data Buffer
+4/+5	Requested Length
+6/+7	Actual Length

REQUIRED INPUTS

+0	Parameter count (4 parameters in list).
+1	Reference number for an open file as returned by `OPEN`.
+2/+3	Address (LO/HI) of the data to be written to disk. This buffer should not be confused with the "file buffer" passed to `OPEN` which is separate, and should not be used by the caller's program.
+4/+5	Number (LO/HI) of bytes of data to write from the data buffer.

RETURNED VALUES

+6/+7	Actual number of bytes written. Unless an error occurs during the operation, this field should match the requested length in +4/+5.
Return Code	`$ØØ`—No errors `$Ø4`—Parameter count is not `$Ø4` `$27`—I/O error `$2B`—Disk is write protected `$43`—Invalid reference number `$48`—Disk full `$4E`—Access refused: `WRITE` bit not enabled `$56`—Bad buffer address (check System memory bit map) `$5A`—Damaged disk freespace bit map

$CC CLOSE
CLOSE OPEN FILE(S), FLUSHING BUFFERS

FUNCTION For a specific open file, this function flushes any data which has not yet actually gone to disk from the file buffer, releases the file buffer to the caller for reuse, sets the `BACKUP` bit in the `ACCESS` flags for the file, updates the directory entry for the file with block count, etc., and frees the reference number and File Control Block (FCB) for use with a later `OPEN`. Each `OPEN` must have a corresponding `CLOSE`. If a non-specific call is made (`REFNUM = Ø`), all open files at the current `LEVEL` (`$BF94`) or higher are closed.

PARAMETER LIST FORMAT

+0	`$Ø1`
+1	Reference Number

REQUIRED INPUTS

+0	Parameter count (1 parameter in list).
+1	Reference number for an open file as returned by `OPEN` or `$ØØ` if all files at the current level or higher are to be closed. If a multiple file request is made and an error occurs on one file, this does not prevent the MLI from attempting to complete the close operation for any other files. If multiple errors occur, only the last error return code is passed back to the caller.
$BF94	Current file `LEVEL` in the System Global Page. If set to `$ØØ` before this call, all open files are closed.

RETURNED VALUES

Return Code

$ØØ—No errors
$Ø4—Parameter count is not $Ø1
$27—I/O error
$2B—Disk is write protected
$43—Invalid reference number
$5A—Damaged disk freespace bit map

$CD FLUSH
FLUSH ALL WRITE BUFFERS FOR FILES

FUNCTION

For a specific open file, this function flushes any data which has not yet actually gone to disk from the file buffer, updates the directory entry for the file, and sets the `BACKUP` bit in the `ACCESS` flags for the file (if data was written). If no write operations have occurred, then the `FLUSH` call is ignored. If a non-specific call is made (`REFNUM = Ø`), all open files at the current `LEVEL` (`$BF94`) or higher are flushed. The flush call is useful when it is desirable to force write data out to disk before a long period of inactivity in case of power loss or other disasters.

PARAMETER LIST FORMAT

+0 $Ø1
+1 Reference Number

REQUIRED INPUTS

+0 Parameter count (1 parameter in list).

+1 Reference number for an open file as returned by `OPEN`, or `$ØØ` if all files at the current level or higher are to be flushed. If a multiple file request is made and an error occurs on one file, this does not prevent the MLI from attempting to complete the flush operation for any other files. If multiple errors occur, only the last error return code is passed back to the caller.

$BF94 Current file `LEVEL` in the System Global Page. If set to `$ØØ` before this call, all open files are flushed.

RETURNED VALUES

Return Code

$ØØ—No errors
$Ø4—Parameter count is not $Ø1
$27—I/O error
$2B—Disk is write protected

$43—Invalid reference number

$5A—Damaged disk freespace bit map

$CE SET_MARK
CHANGE FILE POSITION WITHIN AN OPEN FILE

FUNCTION

When a file is first opened, the MLI establishes a "file position" at which reading or writing will occur at the beginning of the file (zero). As data is read or written, the file position is moved to allow sequential access to the file. This file position describes the relative byte offset to the next byte in the file to be accessed. If random access to a file is desired, the caller may use this function to change the position to another location in the file before issuing a `READ` or `WRITE` call. If the file position is moved to an area of the file where no data exists (i.e. an area which has never been written), new data and/or index blocks will be allocated when the next `WRITE` call is made. This function may be used in conjunction with the `GET_EOF` call (`$D1`) to append data to the end of a file.

PARAMETER LIST FORMAT

+0	$Ø2
+1	Reference Number
+2/+3/+4	New File Position

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+1	Reference number for an open file as returned by `OPEN`.
+2/+3/+4	The new file position to be set. This is a 3-byte number (least significant byte first, most significant byte last) representing the byte offset into the file. The position of the first byte in a file is zero. The position may not exceed the current end of file position.

RETURNED VALUES

Return Code

$ØØ—No errors

$Ø4—Parameter count is not $Ø2

$43—Invalid reference number

$4D—File position beyond end of file

$5A—Damaged disk freespace bitmap

$CF GET_MARK
RETURN FILE POSITION WITHIN AN OPEN FILE

FUNCTION

When a file is first opened, the MLI establishes a "file position" at which reading or writing will occur at the beginning of the file (zero). As data is read or written, the file position is moved to allow sequential access to the file. This file position describes the relative byte offset to the next byte in the file to be accessed. This function will return the current value of the file position.

PARAMETER LIST FORMAT

+0	`$Ø2`
+1	Reference Number
+2/+3/+4	File Position

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+1	Reference number for an open file as returned by `OPEN`.

RETURNED VALUES

+2/+3/+4	The current file position value. This is a 3-byte number (least significant byte first, most significant byte last) representing the byte offset into the file of the next byte to be read or written. The position of the first byte in a file is zero.
Return Code	`$ØØ`—No errors `$Ø4`—Parameter count is not `$Ø2` `$43`—Invalid reference number

$D0 SET_EOF
CHANGE END OF FILE POSITION OF AN OPEN FILE

FUNCTION

This function changes the end of file mark (or file size). It is not normally necessary to change the end of file mark since the WRITE function will automatically extend the EOF mark as new data is written to the end of the file. This function is useful, however, to truncate a file or to allow random positioning within a very large sparse file. If the new end of file position passed by the caller is less than the old one, the file is truncated and excess data and index blocks are freed for reuse by the system. If it exceeds or equals the old value, no new blocks will be allocated until they are needed in a WRITE operation. If the new end of file would leave the current file position outside the limits of the file, it is forced back to the new end of file position. The EOF mark of a directory file may not be changed with SET_EOF. Note that the file size does not necessarily represent the amount of disk space the file requires, since the file may be sparse (see Chapter 4).

PARAMETER LIST FORMAT

+0	$Ø2
+1	Reference Number
+2/+3/+4	New EOF Position

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+1	Reference number for an open file as returned by OPEN.
+2/+3/+4	The new end of file position. This is a 3-byte number (least significant byte first, most significant byte last) representing the byte offset into the file of the last byte plus one. The position of the first byte in the file is zero (the EOF of an empty file).

RETURNED VALUES

Return Code

$ØØ—No errors
$Ø4—Parameter count is not $Ø2
$27—I/O error
$43—Invalid reference number
$4D—Position is too large for volume
$4E—Access refused: WRITE bit not enabled
$5A—Damaged disk freespace bit map

$D1 GET_EOF
RETURN END OF FILE POSITION OF AN OPEN FILE

FUNCTION

This function returns the value of the end of file mark for an open file. `GET_EOF` may be used to determine the size of a sequential file or to find the end of a file so that data may be appended to it. `GET_EOF` for a directory file will return the number of blocks used multiplied by 512 bytes. Note that the file size does not necessarily represent the amount of disk space the file requires, since the file may be sparse (see Chapter 4).

PARAMETER LIST FORMAT

+0	`$Ø2`
+1	Reference Number
+2/+3/+4	`EOF` Position

REQUIRED INPUTS

+0	Parameter count (2 parameters in list).
+1	Reference number for an open file as returned by `OPEN`.

RETURNED VALUES

+2/+3/+4	The current end of file position. This is a 3-byte number (least significant byte first, most significant byte last) representing the byte offset into the file of the last byte plus one. The position of the first byte in the file is zero (the `EOF` of an empty file).
Return Code	`$ØØ`—No errors
	`$Ø4`—Parameter count is not `$Ø2`
	`$43`—Invalid reference number

$D2 SET_BUF
CHANGE OPEN FILE'S BUFFER ADDRESS

FUNCTION

This function allows the caller to move an open file's file buffer to another location in memory. Since `READ` and `WRITE` references are by Reference Number, the MLI must memorize the location of the file buffer at `OPEN` time. If the buffer must be moved, this call allows the programmer to inform the MLI and allow it to move the contents of the buffer to the new location. The system memory bit map is updated to reflect the change.

PARAMETER LIST FORMAT

+0	`$Ø2`
+1	Reference Number

+2/+3 New Address of File Buffer

REQUIRED INPUTS

+0 Parameter count (2 parameters in list).

+1 Reference number for an open file as returned by `OPEN`.

+2/+3 The address (LO/HI) of a new 1024-byte location in which the MLI may maintain the open file's buffer. It must be on an even page boundary (LO byte of address is zero) and not be allocated by the MLI to any open file. The contents of the current file buffer are transferred to this new area, and the old buffer is marked released in the System Global Page memory bit map.

RETURNED VALUES

Return Code `$ØØ`—No errors

`$Ø4`—Parameter count is not `$Ø2`

`$43`—Invalid reference number

`$56`—Buffer already in use by MLI

$D3 GET_BUF
RETURN OPEN FILE'S BUFFER ADDRESS

FUNCTION This function returns the address of the file buffer associated with an open file to the caller.

PARAMETER LIST FORMAT

+0 `$Ø2`

+1 Reference Number

+2/+3 Address of File Buffer

REQUIRED INPUTS

+0 Parameter count (2 parameters in list).

+1 Reference number for an open file as returned by `OPEN`.

RETURNED VALUES

+2/+3 The address (LO/HI) of the 1024-byte file buffer in use by the MLI for this file.

Return Code `$ØØ`—No errors

`$Ø4`—Parameter count is not `$Ø2`

`$43`—Invalid reference number

MLI ERROR CODES

`$ØØ` No error occurred. Operation completed successfully.

`$Ø1` Invalid MLI function code number.

`$Ø4` Incorrect parameter count in parameter list for the function code used.

`$25` The ProDOS interrupt handler vector table is full. There are already four addresses stored there.

`$27` A device driver reported an Input/Output error on the media. This could be anything from the diskette drive door being open to a real error on the surface of the diskette.

`$28` No device is connected for the unit number given. This can happen if no identifiable controller ROM was present in the indicated slot.

`$2B` An attempt was made to write to the disk, but it was write protected. Remove the tape over the write-protect notch if you wish to write on this diskette.

`$2E` In the process of performing an `ONLINE` call, the MLI discovered that a diskette for which there were open files had been removed from its drive and replaced by another volume. Since no check is made when writing to an open file, it is possible that some blocks on the new volume have been damaged.

`$4Ø` The pathname has invalid syntax. Check to make sure the first byte is a count of the number of characters that follow. Also, be sure that each sub-level index begins with an alphabetic character and that each level is separated from the next by a slash (/).

`$42` Eight files are open and there is no more room in the MLI's File Control Block (FCB) table for another open file. If you didn't expect any files to be open, set the `LEVEL` to zero and issue a global `CLOSE`.

`$43` The reference number passed in the parameter list does not denote an open file. Make sure that the `OPEN` call was successful before issuing other calls by reference number.

`$44` The pathname supplied could not be followed to the final directory. One or more of the subordinate directories in the path did not exist.

`$45` The volume indicated by the pathname is not currently mounted on any drive.

$46 The file indicated by the last name in the pathname was not found in the final directory.

$47 A CREATE or RENAME was attempted and the file named already exists. To perform the operation would create a duplicate entry in the directory.

$48 An attempt was made to find one or more free disk blocks (to extend a directory, add a new data block for a file, etc.), but the Volume Bit Map indicates that the diskette is now full.

$49 An attempt was made to CREATE another file in the Volume Directory, but there are no free entries. Unlike subdirectories, the Volume Directory is of a fixed size (51 entries) and cannot be extended.

$4A An earlier version of the ProDOS MLI is being used to read a file which was created with a later version. The older MLI cannot handle this file properly. Use a newer version of ProDOS. This error can also occur if the final subdirectory header has an improper format. The byte at +$14 in the subdirectory key block (reserved bytes) must contain 5 and only 5 one bits (it is usually $75).

$4B The storage type of a file is not one of the storage types currently supported by this version of ProDOS. Currently, only Seedlings ($Ø1), Saplings ($Ø2), Trees ($Ø3) and Directories ($ØD) are supported.

$4C A READ operation was attempted and the current file position is at the End of File mark. No data was transferred.

$4D An attempt was made to move the file position past the End of File mark. If this position is desired, first move the EOF mark.

$4E An error occurred having to do with the ACCESS bits for a file. Usually this means you attempted to WRITE to a write protected file, or you attempted to DESTROY or RENAME a locked file. You can also get this error if any of the reserved bits are ones for the ACCESS byte of a SET_FILE_INFO call.

$5Ø An attempt was made to OPEN, RENAME, or DESTROY a previously OPENed file. Multiple OPENs are only allowed if the file's WRITE ACCESS bit is off (write disabled).

$51 When searching a directory, it was determined that the count of active file entries in the directory header was larger than the number of entries actually encountered. The directory is damaged and some file entries may be lost.

$52 The disk volume which was accessed is not a ProDOS disk. The criteria for determining whether a volume is a ProDOS formatted volume are: the first two bytes of the Volume Directory key block must be zero (previous block pointer); and the byte at offset 4 into the Volume Directory key block must be $E or $F (storage type).

$53 One or more of the values in the parameter list is not within its acceptable range. For example, an interrupt handler address of $ØØØØ was passed to ALLOC_INTERRUPT.

$55 At most, only eight "mounted" volumes may be known to ProDOS at one time. Usually this is no problem since only eight files may be open at a time. However, if a single file is open on each of eight different volumes and an ONLINE call is made requesting the volume name mounted on a ninth device, this error will result.

$56 The address of the I/O file buffer passed to OPEN or SET_BUF is invalid. The buffer overlaps a previously assigned buffer, memory below $2ØØ, or ProDOS itself. The buffer must be in the caller's memory, and all four of its pages must be marked free in the System Global Page memory bit map.

$57 In the process of mounting volumes and recording their names in the Volume Control Block (VCB) table, the MLI discovered two volumes with the same name. Since all file references must be made by volume name and not necessarily by slot and drive, this condition is not permitted.

$5A The Volume Bit Map describing the freespace on the volume is damaged. A one bit was found, indicating a free block, for a block outside the legal extent of the volume (for a block number beyond the end of the volume).

PASSING COMMAND LINES TO THE BASIC INTERPRETER

For machine language programs running under the ProDOS **BASIC Interpreter** (BI), an interface is provided to allow execution of command lines created by a program, as if they had been entered from the keyboard. This is the highest level and perhaps the easiest to use ProDOS interface. Through it, a machine language program may easily produce CATALOG listings, DELETE or RENAME files, etc.

To call the BI command handler, place the command string in the monitor GETLN line input buffer at $2ØØ. The line may be up to 255 characters in

length, and must be followed by a carriage return character (`$8D`). The most significant bit of each character should be set, and all alphabetics should be in upper case. Once this has been done, call `$BEØ3` in the BI's Global Page (`JSR $BEØ3`).

If an error occurs, the BI error code will be placed in the accumulator. Possible codes are listed in Table 6.6.

Table 6.6 BASIC Interpreter Error Codes

Code	Message
$ØØ	No error
$Ø1	Not used
$Ø2	RANGE ERROR
$Ø3	NO DEVICE CONNECTED
$Ø4	WRITE PROTECTED
$Ø5	END OF DATA
$Ø6	PATH NOT FOUND
$Ø7	Not used
$Ø8	I/O ERROR
$Ø9	DISK FULL
$ØA	FILE LOCKED
$ØB	INVALID PARAMETER
$ØC	NO BUFFERS AVAILABLE
$ØD	FILE TYPE MISMATCH
$ØE	PROGRAM TOO LARGE
$ØF	NOT DIRECT COMMAND
$1Ø	SYNTAX ERROR
$11	DIRECTORY FULL
$12	FILE NOT OPEN
$13	DUPLICATE FILE NAME
$14	FILE BUSY
$15	FILE(S) STILL OPEN

If you wish to print an error message, you need not have a table of messages similar to the above. Instead, place the error number in the A register and call `$BEØC` (`JSR $BEØC`).

Keep in mind that, unless the machine language program was called by a BASIC program, only direct commands may be issued (as if from the keyboard). BASIC file commands such as `OPEN`, `READ`, `WRITE`, `APPEND`, and `POSITION`

will result in a `NOT DIRECT COMMAND` error. Under Apple DOS, commands could be printed with a control-D from an assembly language program, exactly as with BASIC programs. Under ProDOS, this method no longer works. This is because the intercepts used for the "control-D interface" are no longer in the screen output vector, but are now in the Applesoft trace facility, which, of course, isn't active when your machine language program is running.

COMMON ALGORITHMS

Given below are several pieces of code which may be used when working with ProDOS.

IS PRODOS ACTIVE?

The following series of instructions should be used prior to attempting to call the ProDOS MLI.

```
          LDA $BFØØ        GET MLI VECTOR JMP
          CMP #$4C         IS IT A JUMP?
          BNE NOPRODS      NO, PRODOS NOT ACTIVE
```

WHAT KIND OF MACHINE IS THIS?

This code will test to determine what type of Apple is running the program.

```
          LDA #$Ø8
          BIT $BF98        TEST MACHID FROM GLOBAL PAGE
          BEQ OLDSYS       OLDER SYSTEM
          BPL UNKN         FUTURE SYSTEM - UNKNOWN
          BVC APIIC        IT'S AN APPLE IIC
          BVS UNKN         OTHERWISE, UNKNOWN
OLDSYS    BMI EOR3         EITHER A IIE OR A III
          BVC APII         IT'S AN APPLE II
          BVS APIIP        IT'S AN APPLE II+
EOR3      BVS APIII        IT'S AN APPLE III
          ...              OTHERWISE IT'S AN APPLE IIE
```

HOW MUCH MEMORY IS IN THIS MACHINE?

This code will determine whether the Apple has 48K, 64K or 128K of RAM.

```
          LDA $BF98        GET MACHID FROM GLOBAL PAGE
          ASL A            MOVE BITS TO TEST POSITION
          ASL A
          ASL A
          BCC SMLMEM       48K
          BMI MEM128       128K
          ...              OTHERWISE 64K
```

GIVEN A PAGE NUMBER, SEE IF IT IS FREE

This code examines ProDOS's memory bitmap to see if a page is marked free. If so, the page is marked as allocated.

```
BITMAP     EQU $BF58         SEE PAGE 299
           LDA #PAGE         GET PAGE NUMBER (MSB OF ADDR)
           JSR LOCATE        LOCATE ITS BIT IN BITMAP
           AND BITMAP,Y      IS IT ALLOCATED?
           BNE INUSE         YES, CAN'T TOUCH IT
           TXA               PUT BIT PATTERN IN ACCUM
           ORA BITMAP,Y      MARK THIS PAGE AS IN USE
           STA BITMAP,Y      UPDATE MAP
           ...               WE'VE GOT IT NOW
LOCATE     PHA               SAVE PAGE NUMBER
           AND #$Ø7          ISOLATE BIT POSITION
           TAY               THIS IS INDEX INTO MASK TABLE
           LDX BITMASK,Y     PUT PROPER BIT PATTERN IN X
           PLA               RESTORE PAGE NUMBER
           LSR A             DIVIDE PAGE BY 8
           LSR A
           LSR A
           TAY               Y-REG IS OFFSET INTO BITMAP
           TXA               PUT BIT PATTERN IN ACCUM
           RTS               DONE
BITMASK    DFB               $8Ø,$4Ø,$2Ø,$1Ø      BIT MASK PATTERNS
           DFB               $Ø8,$Ø4,$Ø2,$Ø1
```

IS A BASIC PROGRAM RUNNING?

This code will allow your machine language program to determine whether it was called by a BASIC program.

```
           LDA $BE42         CHECK BI'S STATE
           BEQ NOTRUN        IN IMMEDIATE MODE
           ...               ELSE, BASIC PROGRAM RUNNING
```

SETTING UP YOUR OWN RESET VECTOR

The code below will set up a user-defined RESET handler.

```
           LDA #>RESRTN      SET UP LSB
           STA $3F2
           LDA #<RESRTN      SET UP MSB
           STA $3F3
           EOR #$A5          MAKE POWER-UP BYTE
           STA $3F4
           ...               RESET VECTOR READY
RESRTN     ...               RESET HANDLER ROUTINE
```

Note: If you're using the Merlin Assembler, the first line should be `LDA #<RESRTN` and the third line should be `LDA #>RESRTN`.

ACTIVATE A PRINTER OR OTHER PERIPHERAL

To activate a printer or other peripheral driver under the ProDOS BASIC Interpreter, do **not** modify the vectors in zero page (CSWL/CSWH or KSWL/KSWH). Doing so will "disconnect" the interpreter and prevent it from intercepting command lines. Instead, store the address of the peripheral driver in BI Global Page in the VECTOUT ($BE3Ø) or VECTIN ($BE32) words. The following code will start up a printer in Slot 1.

```
LDA $BE3Ø        SAVE ORIGINAL CONTENTS OF VECTOUT
STA OLDVEC       IN MEMORY SO I CAN TURN THE
LDA $BE31        PRINTER OFF WHEN I'M THRU
STA OLDVEC+1
LDA #$ØØ         PLACE $C1ØØ IN VECTOUT
STA $BE3Ø
LDA #$C1
STA $BE31
...              BEGIN PRINTING VIA COUT
LDA OLDVEC
STA $BE3Ø        RESTORE PREVIOUS OUTPUT VECTOR
LDA OLDVEC+1
STA $BE31
```

Chapter 7
Customizing ProDOS

SYSTEM PROGRAMMING WITH PRODOS

Apple has provided a number of customizing interfaces to ProDOS which allows a programmer to tailor the operation of the system to his specific application needs. These interfaces are considered "safe" and acceptable when working with ProDOS.

Before discussing specific system programming considerations, it is important to understand how ProDOS uses memory and what areas are reserved for its use versus those available for applications programs. Referring to Figure 7.1, the following areas of memory are officially "owned" by the ProDOS Kernel: `$DØØØ-$FFFF` in the language card (primary `$DØØØ-$DFFF` bank); `$BFØØ-$BFFF`; Zero page locations `$3A-$4F`; and part of the second 4K bank of the language card (starting at `$D1ØØ`). The rest of this 4K bank is reserved for the `QUIT` code driver and future uses. The ProDOS Kernel also reserves portions of auxiliary memory (128K) for future use—namely, the same locations it uses in main memory, zero page locations `$8Ø-$FF`, and locations `$2ØØ-$3FF`. Apple's future plans for these memory areas include networking and menu managers, so if you use them you do so at your own risk. In a 128K machine, ProDOS currently sets up an electronic "RAM drive" volume in the auxiliary memory. At present, this volume encompasses most of the auxiliary 64K. In the future, its size may be reduced to accomodate enhancements as mentioned above. You can use auxiliary memory for your own applications if you disable the `/RAM` device driver (see instructions later in this chapter). If the BASIC Interpreter is used, an additional area of memory from `$96ØØ-$BEFF` is allocated to its use. `$3DØ-$3FF` is used as a system vector area as defined by the *Apple II Reference Manual for the IIe Only*.

Note that ProDOS routines, including the clock driver, make heavy use of `$2ØØ-$2FF`, the monitor `GETLN` input line buffer. If your programs use this area, you should not depend upon it across ProDOS system calls. You should also be aware of the fact that the MLI cannot be called from memory in the

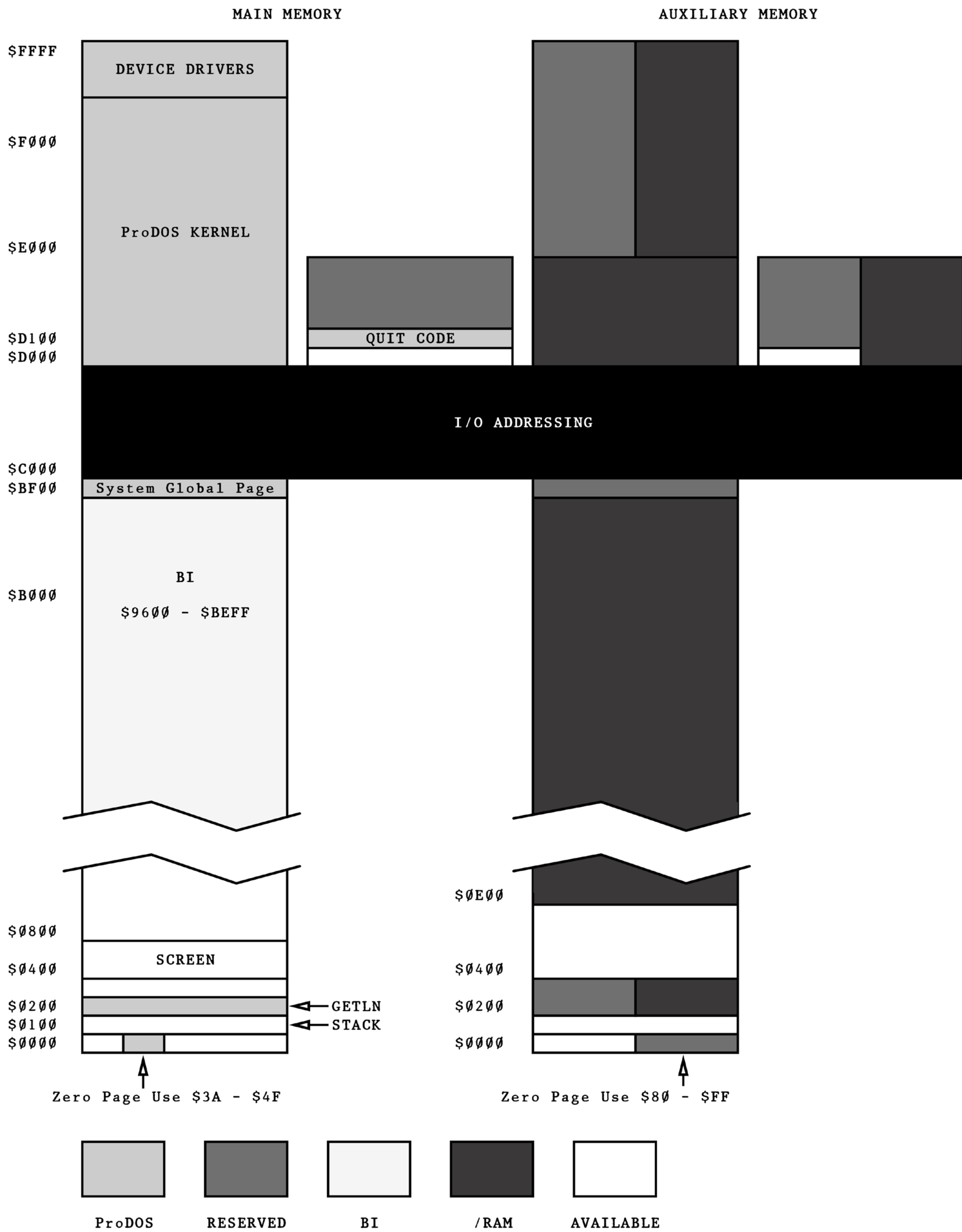


Figure 7.1 ProDOS Memory Usage

auxiliary bank, and that memory outside the area between $2ØØ and $BEFF in the main RAM bank may not be used for buffers passed to the MLI.

INSTALLING A PROGRAM BETWEEN THE BI AND ITS BUFFERS

Once in a while it is useful to find a "safe" place in memory to put a machine language program (a printer driver, or external command handler, perhaps) where BASIC and ProDOS will never walk over it. If the program is less than 200 bytes long, $3ØØ is a good choice. For larger programs, it is usually better to "tuck" the program in between the ProDOS BASIC Interpreter and its file I/O buffers. The program need not be relocatable, since the BI will always be in the same place in memory, and the program can be placed at a fixed location just below it (see Figure 5.1). More than one program may be "tucked" in this area, but this may require one or more of them to be relocated, depending upon the order in which they are loaded.

To request space for a program, you must execute a call to the BI's buffer allocation subroutine using a vector in the BI Global Page. You may request a buffer of any size as long as it an even multiple of pages (one page is 256 bytes). When called, the buffer allocation routine relocates any open file buffers as well as its General Purpose Buffer downward in memory, lowering Applesoft's HIMEM pointer as necessary, and returns the address of the first page in the new buffer. The new buffer will be placed directly below $9AØØ. Subsequent calls to the buffer allocation routine will cause allocations of buffers below earlier ones. The BI file buffers will always be lower in memory than any externally allocated buffers. When you are finished with all of the buffers you have allocated, you may free all of them with a single call. There is no provision for freeing individual buffers.

To allocate a buffer, invoke the following subroutine:

```
GBUFF    LDA #4         ALLOCATE 4 PAGES (1Ø24 BYTES)
         JSR $BEF5      CALL GETBUFR
         BCS ERROR      DID AN ERROR OCCUR?
         STA BUFMSB     STORE BUFFER ADDRESS MSB
         LDA #Ø
         STA BUFLSB     STORE BUFFER ADDRESS LSB
         RTS            ALL DONE
```

To free all buffers you have allocated:

```
FBUFFS   JSR $BEF8 CALL FREEBUFR
```

Note that you may allocate as many buffers as you wish using the GBUFF subroutine, but that a single call to FBUFFS frees all buffers.

ADDING COMMANDS TO THE PRODOS BASIC INTERPRETER

There exists a well defined interface to allow you to write your own command handlers for the ProDOS BASIC Interpreter. Suppose, for example, that you wish to add a COPY command which will accept an input pathname, followed by a comma and an output pathname. You can write a handler for such a command in assembly language, install the handler between the BI and its buffers (see the previous section), and then inform the BI of its existence. Every time the BI receives a command line it doesn't recognize, it will pass it through to your handler before passing it to Applesoft. Note that this implies that your command's name must be different from any existing ProDOS command name. You may not replace or supercede an existing ProDOS command.

To install your own command handler, place its entry point address in the vector in the BI Global Page at $BEØ7 and $BEØ8. These two bytes are the address portion of a Jump (JMP) instruction (EXTERNCMD) which normally points to a Return from Subroutine (RTS) instruction within the BI. It is not a good idea to assume that this address is pointing to an RTS since someone else's command handler could have been previously installed. To make sure you do not "disconnect" an earlier installed command handler and yours is "daisy chained" to it, save the address you find in EXTERNCMD + 1 and branch to it from your handler if the command line passed is not your command.

Each time the BI scans a command line and cannot find the command name in its table of valid names, it will call your routine. Your program should compare the command in the command line with yours. The address of the command line is in VPATH1 ($BE6C/$BE6D) in the BI Global Page. The command line consists of a length byte followed by one or mroe ASCII characters with their most significant bit off. If the command is not yours, jump to the next handler (previous contents of $BEØ7/$BEØ8) with the carry set (SEC) to indicate the command is not yours. If the command is yours, there are two options. If the command's syntax is not compatible with other ProDOS commands (i.e. it has non-standard operands or keywords), you may immediately begin performing the function indicated. When the program finishes, it should store a zero in PBITS in the BI Global Page ($BE54) to indicate no operands are to be parsed, and return (RTS) with the carry clear (CLC). In this case, do

not JMP to the next handler as you would if the command was not yours. If, on the other hand, the command has standard ProDOS syntax, you can use the BI's syntax scanner to pick off the operands and optional keywords. To do this, once you have identified the command as yours, store the address of the beginning of your code which will process the command (after the syntax scan) in XTERNADDR ($BE5Ø/$BE51) in the BI Global Page, store a $ØØ in XCNUM ($BE53) to indicate that this is an external command, and store the length of your command name (less one) in XLEN ($BE52) so that the BI will know where to start looking for operands. You should also set up PBITS (two bytes of flags) in the BI Global Page to describe the operands the BI is likely to find on your command. If you have a very simple command with only a pathname as an operand, you can set PBITS to $Ø1, $ØØ. If you want the BI to automatically provide the prefix of the current volume (default slot, drive) as well as allow the S and D keywords, set PBITS to $Ø1, $Ø4. Once you have set up XTERNADDR, XCNUM, XLEN, and PBITS, return (RTS) to the BI with the carry clear (CLC). When the command line has been successfully scanned, control will return to your handler at the location you indicated in XTERNADDR. If a SYNTAX ERROR occurs, control will not return. When your command handler completes its tasks, it may return to the BI with an RTS instruction (the carry here is insignificant). Your handler need not save or restore any registers.

An example of a command handler is given in Appendix A. This program installs a handler between the BI and its buffers, and connects it to ProDOS through the EXTERNCMD vector. If the ProDOS user enters the command "TYPE" followed by a pathname, the command handler reads the indicated file and prints it on the screen.

DISABLE /RAM VOLUME FOR 128K MACHINES

If your application needs to use the additional 64K in the Extended 80-column Card (or the alternate 64K bank in the IIc) for its own purposes, rather than as an electronic disk drive (RAM drive), you should disable the /RAM device driver. You might want to do this if you plan to use the "double HIRES" graphics feature of the Apple IIe and IIc, for example.

The /RAM device driver is installed by the ProDOS Loader/Relocator when the Kernel is loaded. Part of it resides in the Kernel itself (from $FFØØ–$FF7F), and the remainder remains in auxiliary memory at $2ØØ–$3FF. Its address is placed in the list of device drivers for Slot 3, Drive 2 in the System Global Page.

One way to avoid conflicts between the /RAM and your application is the BSAVE a dummy file such that its blocks will coincide with the area of memory you will be using. If you BSAVE an 8K file to /RAM (before any other operations on the /RAM volume), it will fall across $2ØØØ-$3FFF, the primary HIRES buffer. If you save a second 8K file it will fall across $4ØØØ-$5FFF, the secondary HIRES buffer. This is the easiest way to use "double HIRES" graphics while leaving the /RAM volume partially available for your use as an electronic disk drive.

If you want to totally disable the /RAM device driver, you must remove its entry from the System Global Page device driver vector list (DEVADR32). You must also remove the device number for Slot 3, Drive 2 from the online devices list (DEVLST), and reduce the device count (DEVCNT) by one. If you plan to reinstall the /RAM volume later, be sure to save the contents of DEVADR32 in a safe place so you can later restore it. Note that it is good programming practice to leave /RAM installed upon exiting your program so that other applications may use it. Reinstalling /RAM erases ("formats") the volume, so you should not reinstall it upon entry to an application which will be reading files passed via the /RAM volume by a previous application.

The following subroutine will remove the /RAM driver, allowing alternate uses of the auxilary 64K:

```
                2    * START BY CHECKING TO SEE IF /RAM COULD BE THERE
ØØØØ: AD 98 BF  4    REMOVE   LDA   $BF98      CHECK MACHID
ØØØ3: 29 3Ø     5             AND   #$3Ø       ISOLATE MEMORY BITS
ØØØ5: C9 3Ø     6             CMP   #$3Ø       128K?
ØØØ7: DØ 1Ø     7             BNE   NORAM      NO - NO AUX MEMORY
ØØØ9: AD 26 BF  8             LDA   $BF26
ØØØC: CD 16 BF  9             CMP   $BF16      IF SLOT 3, DRV <> DRV 2 VECTOR..
ØØØF: DØ ØA     1Ø            BNE   GOTRAM     THEN IT'S INSTALLED
ØØ11: AD 27 BF  11            LDA   $BF27
ØØ14: CD 17 BF  12            CMP   $BF17
ØØ17: DØ Ø2     13            BNE   GOTRAM
ØØ19: 38        14   NORAM    SEC              ;
ØØ1A: 6Ø        15   OKXIT    RTS
                17   * SAVE OLD VECTOR AND REMOVE IT
ØØ1B: AD 26 BF  19   GOTRAM   LDA   $BF26      SAVE OLD VECTOR CONTENTS
ØØ1E: 85 59     2Ø            STA   OLDVEC
ØØ2Ø: AD 27 BF  21            LDA   $BF27
ØØ23: 85 5A     22            STA   OLDVEC+1
ØØ25: AD 16 BF  23            LDA   $BF16      POINT IT AT NISTALLED DEV
ØØ28: 8D 26 BF  24            STA   $BF26
ØØ2B: AD 17 BF  25            LDA   $BF17
ØØ2E: 8D 27 BF  26            STA   $BF27
                28   * SQUISH OUT DEVICE NUMBER FROM DEVLST
ØØ31: AE 31 BF  3Ø            LDX   $BF31      GET DEVCNT
ØØ34: BD 32 BF  31   DEVLP    LDA   $BF32,X    PICK UP LAST DEVICE NUM
ØØ37: 29 7Ø     32            AND   #$7Ø       ISOLATE SLOT
ØØ39: C9 3Ø     33            CMP   #$3Ø       SLOT = 3?
ØØ3B: FØ Ø5     34            BEQ   GOTSLT     YES, CONTINUE
ØØ3D: CA        35            DEX
ØØ3E: 1Ø F4     36            BPL   DEVLP      CONTINUE SEARCH BACKWARDS
ØØ4Ø: 3Ø D7     37            BMI   NORAM      CAN'T FIND IT IN DEVLST
ØØ42: BD 33 BF  38   GOTSLT   LDA   $BF32+1,X  GET NEXT NUMBER
```

```
ØØ45: 9D 32 BF  39           STA   $BF32,X    AND MOVE THEM FORWARD
ØØ48: E8        4Ø           INX
ØØ49: EC 31 BF  41           CPX   $BF31      REACHED LAST ENTRY?
ØØ4C: DØ F4     42           BNE   GOTSLT     NO, LOOP
ØØ4E: CE 31 BF  43           DEC   $BF31      REDUCE DEVCNT BY 1
ØØ51: A9 ØØ     44           LDA   #$ØØ       ZERO LAST ENTRY IN TABLE
ØØ53: 9D 32 BF  45           STA   $BF32,X
ØØ56: 18        46           CLC
ØØ57: 9Ø C1     47           BCC   OKXIT      BRANCH ALWAYS
ØØ59: ØØ ØØ     49  OLDVEC   DW    Ø          OLD VECTOR SAVE AREA
```

To reinstall the /RAM driver, execute this subroutine:

```
                51  * SEE IF SLOT 3 HAS A DRIVER ALREADY
                53  HIMEM    EQU   $73        PTR TO BI'S GENERAL PURPOSE BUFFER
ØØ5B: AE 31 BF  55  INSTALL  LDX   $BF31      GET DEVCNT
ØØ5E: BD 32 BF  56  INSLP    LDA   $BF32,X    GET A DEVNUM
ØØ61: 29 7Ø     57           AND   #$7Ø       ISOLATE SLOT
ØØ63: C9 3Ø     58           CMP   #$3Ø       SLOT 3?
ØØ65: FØ 3F     59           BEQ   INSOUT     YES, SKIP IT
ØØ67: CA        6Ø           DEX
ØØ68: 1Ø F4     61           BPL   INSLP      KEEP UP THE SEARCH
                63  * RESTORE THE DEVNUM TO THE LIST
ØØ6A: AE 31 BF  64           LDX   $BF31      GET DEVCNT AGAIN
ØØ6D: E4 ØD     65           CPX   $ØD        DEVICE TABLE FULL
ØØ6F: DØ ØØ     66           BNE   INSLP2
                67  * ERROR ... YOUR ERROR ROUTINE
ØØ71: BD 31 BF  68  INSLP2   LDA   $BF32-1,X  MOVE ALL ENTRIES DOWN
ØØ74: 9D 32 BF  69           STA   $BF32,X    TO MAKE ROOM AT FRONT
ØØ77: CA        7Ø           DEX
ØØ78: DØ F7     71           BNE   INSLP2
ØØ7A: A9 BØ     72           LDA   #$BØ
ØØ7C: 8D 32 BF  73           STA   $BF32      SLOT 3, DRIVE 2 AT TOP OF LIST
ØØ7F: EE 31 BF  74           INC   $BF31      UPDATE DEVCNT
                76  * NOW PUT BACK THE DEVICE DRIVER VECTOR
ØØ82: A5 59     78           LDA   OLDVEC
ØØ84: 8D 26 BF  79           STA   $BF26
ØØ87: A5 5A     8Ø           LDA   OLDVEC+1
ØØ89: 8D 27 BF  81           STA   $BF27
                83  * FINALLY, REFORMAT THE /RAM VOLUME
ØØ8C: AD 32 BF  85           LDA   $BF32
ØØ8F: 85 43     86           STA   $43        DEVNUM = SLOT 3, DRIVE 2
ØØ91: A9 Ø3     87           LDA   #3
ØØ93: 85 42     88           STA   $42        CMD = FORMAT
ØØ95: A5 73     89           LDA   HIMEM      512-BYTE BLOCK BUFFER
ØØ97: 85 44     9Ø           STA   $44        (PAGE ALIGNED)
ØØ99: A5 74     91           LDA   HIMEM+1    WE CAN USE BI'S G.P. BUFFER
ØØ9B: 85 45     92           STA   $45        (IF BI IS AROUND)
ØØ9D: 8D 8Ø CØ  93           STA   $CØ8Ø      SELECT L.C. FOR DRIVER
ØØAØ: 2Ø A7 ØØ  94           JSR   RAMDRV     GO FORMAT THE VOLUME
ØØA3: 8D 81 CØ  95           STA   $CØ81      SELECT MOTHERBOARD ROMS
ØØA6: 6Ø        96  INSOUT   RTS              ; AND EXIT TO CALLER
ØØA7: 6C 26 BF  97  RAMDRV   JMP   ($BF26)    <<< JUMP TO /RAM DRIVER >>>
```

WRITING YOUR OWN INTERPRETER

A ProDOS "Interpreter" (also known as a "System Program") is a machine language program which stands between the user and the ProDOS MLI, providing a function. An interpreter may be executed by the smart RUN

command ("-"), may be invoked at boot time, or may be executed upon leaving another ProDOS interpreter. Interpreters are stored in SYS files on a ProDOS volume, and are initially loaded at $2000, although they may include code to relocate themselves elsewhere once they begin execution. Examples of interpreters are BASIC.SYSTEM (the "BI"), FILER, CONVERT, and EDASM.SYSTEM. According to convention, an interpreter must be able to pass control to any other interpreter when it exits.

When writing your own interpreter, you must be aware of these considerations:

1. You must BSAVE your interpreter as a "SYS" type file from location $2000. If you want to include code to execute elsewhere in the machine, you may include a front-end which relocates the rest of the program (this is what the BI does). Normally, the memory available to you in a 64K system includes $800-$BEFF. If you are running in a 48K machine, the ProDOS Kernel occupies memory from $9000-$BFFF so you are limited to $800-$8FFF for your program.
2. If you want your interpreter to be automatically executed as the first interpreter when ProDOS boots, you must name it "xxxx.SYSTEM", here xxxx can be any name. It must also be the first SYS file using that naming convention to be found in the Volume Directory of the boot diskette.
3. In order to insure correct operation of the interrupt handler in the ProDOS Kernel, set the stack register (S) to point to the top of the stack page ($FF) upon entry, and do not use more than the top three quarters of the stack. The interrupt handler assumes that the last item on your stack is stored at $1FF, when it makes its determination of whether or not to save part of the contents of the stack before invoking an interrupt driver routine.
4. As soon as your program begins execution, it should set up the POWERUP byte in page 3 and three areas in the System Global Page as follows.

```
$03F4: POWERUP byte
$BF58: BITMAP (system memory bit map)
$BFFC: IBAKVER (minimum version of MLI acceptable)
$BFFD: IVERSION (version number of your interpreter)
```

When your interpreter gets control, it should first set up the RESET vector at $3F2/$3F3 to point to its own RESET handler and fix the POWERUP byte at $3F4 accordingly. The POWERUP byte should be fixed even if you do not replace the RESET handler address (unless you want to reboot on RESET). To fix the POWERUP byte, exclusive OR the contents of $3F3 with #$A5 and store the result in $3F4.

A subroutine for checking the system memory bit map was given in Chapter 6. Use this to mark those areas in memory which your program will use. Do not mark areas which may be used for MLI buffers. By doing this, the MLI can keep a watchful eye on the execution of your program to prevent accidental overlay of your code with buffers. To determine what values to use for IBAKVER and IVERSION, examine memory in the version of ProDOS you are using for development and note the values at $BFFE (KBAKVER) and $BFFF (KVERSION). Assemble the values you find there as constants into your program, and use these to initialize IBAKVER and IVERSION.

5. If you wish to use 80 columns, first check the MACHID byte in the System Global Page to see if 80 columns are available and then call (JSR) $C3ØØ. To disable 80-column hardware, load a #$15 into the A register and call $C3ØØ. Avoid using the Apple IIe and IIc 80-column soft switches, because these will not work for third party 80-column cards or in an Apple II or Apple II Plus.
6. When your program is ready to exit, close all open files, reinstall the /RAM driver if you disconnected it previously, and execute the following code.

```
EXIT         DEC   $3F4      FORCE REBOOT ON RESET
             JSR   $BFØØ     CALL THE MLI
             DFB   $65       QUIT CALL
             DW    PARMS
PARMS        DFB   4         4 PARMS
             DFB   Ø         QUIT TYPE = Ø
             DW    Ø         RESERVED
             DFB   Ø         RESERVED
             DW    Ø         RESERVED
```

The MLI will free any memory you have allocated in the system bit map. It will then prompt the user for a new prefix and pathname for the next interpreter, and will load it and execute it. The code which performs these tasks is at $D1ØØ–$D3FF in the secondary 4K block of the language card. It is moved by the MLI to $1ØØØ–$12FF before execution. You may create your own quit code by replacing the three pages of code image in the language card if you wish.

INSTALLING NEW PERIPHERAL DRIVERS

If you are writing for a peripheral, such as a printer or disk drive, you should be aware of the conventions to which ProDOS adheres when examining and calling drivers.

If your driver is in ROM on the peripheral card itself, it should follow the Apple II standards for peripherals as follows.

For Non-Disk Devices

```
Address     Value
$CsØ5       $38 (standard BI requirement)
$CsØ7       $18 (standard BI requirement)
$CsØB       $Ø1 (generic signature of firmware cards)
$CsØC       $ci (specific device signature)
```

The device signature is made up of two nibbles. "c" defines the class of the device as shown below. The second nibble, "i", is a specific device identifier assigned by Apple Computer, Inc.

```
"c" Nibble  Class
$Ø          reserved
$1          printer
$2          joystick or X-Y input device
$3          serial or parallel card
$4          modem
$5          sound or speech device
$6          clock
$7          mass storage device
$8          8Ø-column card
$9          network or bus interface
$A          special purpose (other)
$B-$F       reserved
```

ProDOS makes the following special check for a clock:

```
Address     Value
$CsØØ       $Ø8 (unique device signature for the
            Thunderclock)
$CsØ2       $28
$CsØ4       $58
$CsØ6       $7Ø
```

For Disk Devices

```
Address     Value
$CsØ1       $2Ø (unique disk device signature)
$CsØ3       $ØØ
$CsØ5       $Ø3
$CsØ7       $3C
$CsFC/D     Disk capacity in blocks (non-DISK II)
$CsFE       Status bits (non-DISK II)
            1... .... removable media
            .1.. .... interruptable device
            ..nn .... number of volumes on device
```

```
.... 1... format allowed
.... .1.. write allowed
.... ..1. read allowed
.... ...1 status read allows
ProFILE status bits = $47
$CsFF  $ØØ = DISK II
$xx = LSB of Block device driver in ROM
      for non-DISK II ($Csxx).
      ProFILE hard disk $xx = $EA
      $xx may not equal $FF
```

If your device driver is in RAM (below `$CØØØ`), and you are invoking it using the BASIC Interpreter's commands `PR# A$xxxx` or `IN# A$xxxx`, the first byte of your code must be a `CLD` instruction (`$D8`); otherwise, the BI will not recognize your routine as a valid driver. If your routine is short, you can place it in the `$3ØØ-$3EF` range. If it is longer, you can call the BI's buffer allocation routine (previously covered in this chapter) to place it between the BI and its buffers.

INSTALLING AN INTERRUPT HANDLER

If you plan to use a peripheral card which supports interrupts, you may want to write an interrupt handler for that card. You should use the ProDOS first level interrupt handler in the Kernel so that other cards may also service their interrupts. To do this, use the MLI:`ALLOC_INTERRUPT` call to install your interrupt handler's entry address in the interrupt vector table within the ProDOS System Global Page. When writing an interrupt handler, follow these steps in the order indicated.

1. Make sure your interrupt handler is stored in main memory between `$2ØØ` and `$BEFF`.
2. Call the MLI with the `ALLOC_INTERRUPT` (`$4Ø`) call to cause your routine's entry point to be placed in the vector table.
3. Perform whatever I/O is necessary specific to your peripheral to enable its interrupt generating mechanism.

When your interrupt routine is called, the first instruction executed should be a `CLD` (to let ProDOS know that this is a valid externally written routine). You should then determine whether the interrupt which caused your routine to be invoked was indeed from your peripheral. If it was not, return to the Kernel with the carry flag set. If it was, service the interrupt, and upon completion,

return to the Kernel with the carry flag clear. Your interrupt handler need not save or restore any registers, and it may use up to 16 bytes of stack space and zero page locations `$FA` through `$FF` (these are saved and restored by the Kernel). The Kernel assumes that the "bottom" of the stack is at `$1FF` when it determines what to save. Your application should always start the stack pointer at `$FF`. Note that the motherboard ROM is deactivated in an interrupt handler routine (do not attempt to print via `$FDED`, for example).

If you wish to remove a previously installed interrupt routine, first disable the interrupt generation mechanism on your peripheral card to prevent further interrupts from occurring, then call the MLI:`DEALLOC_INTERRUPT` function to remove your handler from the list.

When writing an interrupt service routine, you should minimize the actual function performed "on the interrupt." If you are collecting data from a serial port which will later be written to disk, do not write the data while in the interrupt service routine, since this may adversely impact the performance of the program which was executing when the interrupt occurred, or it may cause you to "lose" subsequent interrupts while processing the first. Instead, use the interrupt routine to fill a "circular buffer" which is periodically dumped to disk by the interrupted program. An example of this technique and of writing interrupt handlers in general is given in the `DUMBTERM` program in Appendix A.

If you wish to call the MLI while in an interrupt routine, you should take steps to allow any interrupted MLI call to complete before using the MLI yourself (the MLI is not reentrant). Check the `MLIACTV` flag (`$BF9B`) in the System Global Page to see if the MLI is active. If the MLI is not active, you may issue MLI calls immediately. If the MLI is active, save the contents of `CMDADR` (`$BF9C`) and replace it with an address within your service routine. Then return to the Kernel with the carry clear. When the MLI call completes, control will be passed to you instead of the original MLI caller. You should carefully save all registers, perform your processing, as needed, restore the registers again, and jump to the saved contents of `CMDADR` to allow the original caller to continue. Note that you can be interrupted during your processing unless you disable interrupts. If you are not careful, a subsequent interrupt could cause your interrupt service routine to overwrite the saved contents of `CMDADR` with an address within your own program, causing an infinite loop! It might be a good idea to set a flag when saving `CMDADR` and clear it only when you have completed all processing. Your interrupt service routine can then check the flag and discard any interrupts which occur while you are finishing up processing of the first interrupt.

DIRECT MODIFICATION OF PRODOS — A WORD OF WARNING

Making changes to your copy of ProDOS should only be undertaken when absolutely necessary. In the past, many third party software packages were sold for DOS, the earlier Apple II operating system, which patched or made wholesale changes. Because of the dependency these programs had on fixed locations within DOS and their importance to the collective software offering for the machine, programmers at Apple felt hampered in their efforts to improve DOS. Bugs in DOS could only be fixed with patches to existing code—no reassembly could be performed on the DOS code as this would cause critical locations to move "out from under" existing applications. With the introduction of ProDOS, Apple started out fresh. Earlier shortcomings in DOS have been corrected with ProDOS and numerous enhancements have been added. Hopefully, most packages written for ProDOS will not have to depend on changing the operating system's code itself. In any case, be forewarned: Apple will not hesitate to reassemble the ProDOS Kernel or the BASIC Interpreter or other ProDOS components if changes are desirable, and the stated policy is that programs which depend on locations or entry points which are not published by Apple will do so at their own risk.

Although ProDOS provides most of the functionality needed by the BASIC or assembly language programmer, at times a custom change is desirable. When making a change, weigh its value against the difficulty of reconstructing and reapplying it for later versions of ProDOS as they become available. Of course, if you never plan to upgrade your version of ProDOS this is not a concern. In addition, wholesale modification of ProDOS without a clear understanding of the full implications of each change can result in an unreliable system.

APPLYING PATCHES TO PRODOS

The usual procedure for making changes to ProDOS involves "patching" the object or machine language code in ProDOS. Once a desired change is identified, a few instructions are stored over other instructions within ProDOS to modify the program. There are three levels at which changes to ProDOS may be applied.

- New code may be written and added to ProDOS through a "standard" interface. If this is done, as in the case of an interrupt handler, for example,

there need not be any ProDOS version dependencies involved. Examples of this type of modification have been given earlier in this chapter.

- A patch may be applied to a ProDOS system component, such as the Kernel or the BASIC Interpreter, directly in memory. If this is done, a later reboot will cause the change to "fall out" or be removed. This method is usually used to test a change before making it permanent.
- A patch may be made directly to the diskette containing the ProDOS system component in question. Most ProDOS components are stored as SYS files and may be BLOADed, modified using the monitor, and BSAVEd back to diskette. If a change is to be made to the bootstrap loader (stored in block 0 of the volume), a sector edit/or or the ZAP program given in Appendix A must be used. When applying patches to the BASIC.SYSTEM or PRODOS files, you can find a location within the unrelocated image of the BI or the Kernel if you know its address in the relocated and running version. To do this, refer to Table 7.1. For example, if you wish to patch $9B7C in the BI, you must patch $257C after BLOADing BASIC.SYSTEM. If you wish to change $D32A in the MLI, BLOAD PRODOS and change $3Ø2A.

Table 7.1a ProDOS Patch Locations for FILE = "PRODOS" (64K)

EXECUTION ADDRESS	IMAGE ADDRESS	EXECUTION ADDRESS	IMAGE ADDRESS
BFØØ	4EØØ (64K system global page image)	E8ØØ	45ØØ
DØØØ	2DØØ (alternate 4K: 59ØØ—QUIT Code)	E9ØØ	46ØØ
D1ØØ	2EØØ (alternate 4K: 5AØØ)	EAØØ	47ØØ
D2ØØ	2FØØ (alternate 4K: 5BØØ)	EBØØ	48ØØ
D3ØØ	3ØØØ	ECØØ	49ØØ
D4ØØ	31ØØ	EDØØ	4AØØ
D5ØØ	32ØØ	EEØØ	4BØØ
D6ØØ	33ØØ	EFØØ	4CØØ
D7ØØ	34ØØ	FØØØ	4DØØ
D8ØØ	35ØØ	F1ØØ	zeroed (clock code from F142 from 5ØØØ)
D9ØØ	36ØØ	F2ØØ	zeroed
DAØØ	37ØØ	F3ØØ	zeroed
DBØØ	38ØØ	F4ØØ	zeroed
DCØØ	39ØØ	F5ØØ	zeroed
DDØØ	3AØØ	F6ØØ	zeroed
DEØØ	3BØØ	F7ØØ	zeroed
DFØØ	3CØØ	F8ØØ	52ØØ (diskette driver)
EØØØ	3DØØ	F9ØØ	53ØØ
E1ØØ	3EØØ	FAØØ	54ØØ
E2ØØ	3FØØ	FBØØ	55ØØ

EXECUTION ADDRESS	IMAGE ADDRESS
E300	4000
E400	4100
E500	4200
E600	4300
E700	4400
FC00	5600
FD00	5700
FE00	5800
FF00	2B00 (/RAM device driver through FF99)
FF80	5080 (Interrupt vectors and handler)

Table 7.1b ProDOS Patch Locations for FILE = "BASIC.SYSTEM"

EXECUTION ADDRESS	IMAGE ADDRESS
9A00	2400 (BI image)
9B00	2500
9C00	2600
9D00	2700
9E00	2800
9F00	2900
A000	2A00
A100	2B00
A200	2C00
A300	2D00
A400	2E00
A500	2F00
A600	3000
A700	3100
A800	3200
A900	3300
AA00	3400
AB00	3500
AC00	3600
AD00	3700
AE00	3800
AF00	3900
B000	3A00
B100	3B00
B200	3C00
B300	3D00
B400	3E00
B500	3F00
B600	4000
B700	4100
B800	4200
B900	4300
BA00	4400
BB00	4500
BC00	4600
BE00	4700 (BI Global Page image)

The patches given here are applied directly to a diskette with ProDOS Version 1.0.1 (1 January 1984). You must reboot after making any changes in order to cause them to take effect. Do not make these changes to your original ProDOS System diskette. Modify a copy so you can "back out" any changes you make by copying the original again.

CHANGING THE NAME OF THE STARTUP FILE

You can change the name of the STARTUP file which the BI executes at bootup by patching the first block of BASIC.SYSTEM as follows.

```
BLOAD BASIC.SYSTEM,TSYS,A$2000
CALL -151
21E5:05 48 45 4C 4C 4F
BSAVE BASIC.SYSTEM,TSYS,A$2000
```

Here we are changing the name from STARTUP to HELLO. The first byte indicates the number of characters in the name (5) and may be a maximum of 7 characters. Each ASCII byte should have its most significant bit off. The Startup file may be of any type which can be run using the "-" (Smart RUN) command.

PUT THE CURSOR ON COMMAND THAT CAUSED PRODOS ERROR

When you get a ProDOS error message such as "PATH NOT FOUND" or "FILE TYPE MISMATCH" because you typed the wrong file name or misspelled it slightly, it would be nice if ProDOS would return the cursor on the line with your faulty command so you could easily retype it. To make ProDOS do this from now on, apply the following patches.

```
BLOAD BASIC.SYSTEM,TSYS,A$2000
CALL -151
257C:4C C0 BB
45C0:A4 25 88 88 88 84 25 20 22 FC 4C 3F D4
BSAVE BASIC.SYSTEM,TSYS,A$2000
```

NOTE: The patches described on this page are for Version 1.0.1 of ProDOS and probably will not work with other versions.

HOW TO WRITE TO A DIRECTORY FILE

The ProDOS MLI will not allow explicit WRITEs to a directory file under any circumstances (it makes no difference whether the DIR file is "locked" or not). Under normal conditions, the only program which may modify a directory file is the MLI itself (when CREATEing a new file, updating the info in an old one, or DESTROYing one). If you wish to directly modify a directory entry with your own program, you should follow this procedure to circumvent the MLI.

1. Open the directory file using MLI:OPEN.
2. READ the block requiring update.

3. Execute the following code to find the block number.

```
LDA $CØ8B          SELECT RAM CARD
LDA $CØ8B
LDA REFNUM         PICK UP REF NUM OF FILE
CLC
SBC #Ø             MAKE IT AN OFFSET
LSR A              *32 FOR INDEX INTO FCB'S
ROR A
ROR A
ROR A
TAX
LDA $F31Ø,X        GET CURRENT BLOCK NO.
STA BLKNUM
LDA $F311,X
STA BLKNUM+1
STA $CØ81          SELECT MOTHERBOARD ROMS
```

4. Use MLI:`WRITE_BLOCK` to write back the block. Note that `$F31Ø` and `$F311` may be version-dependent locations.

CREATING A NEW FILE TYPE

When you `CREATE` a file with the MLI, you may specify any file type you wish. If you wish to define a new file type for your application, pick a number between `$F1` and `$F8`. When a `CAT` or `CATALOG` command is issued in the BASIC Interpreter, the file type listed will be a "`$Fn`". If you want to use a three letter abbreviation instead, you must modify the table in the BI. The

A rare moment when a new ProDOS file type is born...

patch given below is highly version dependent and will only work for ProDOS Version 1.0.1 (1 January 1984).

The first thing to do is examine the table of file types in the BI at `$B9DB`. This table consists of 14 entries of one byte each, giving the ProDOS file type number for each of the supported types. You will have to replace one of the entries that you never use with your own file type. The entries need not be in numerical order. Immediately following the type table is a table of 3-byte entries giving the names which correspond to the numeric types. This table is in reverse order to the first and begins at `$B9E9`. As an example, suppose you wished to replace the last entry in the tables, `$19` "`ADB`", with `$F1` "`ABC`".

```
BLOAD BASIC.SYSTEM,TSYS,A$2ØØØ
CALL -151
43E8:F1
43E9:C1 C2 C3
BSAVE BASIC.SYSTEM,TSYS,A$2ØØØ
```

Notice that `$B9E8` maps to `$43E8` in the unrelocated image of `BASIC.SYSTEM.`

Note: The patches described above are for version 1.0.1 of ProDOS and probably will not work with other versions.

RECOVERING DATA FROM A DAMAGED DISK

If one of the sectors which makes up a block is damaged, ProDOS will return with an I/O error. If in fact the error was in the second half of the block, the first half will be read into memory before the I/O error occurs. However, if the error is in the first half of the block, ProDOS will not attempt to read the second half. To recover the second, undamaged sector of the block, the following patch will force ProDOS to ignore any errors while reading the first half of a block. Errors while reading the second half will still behave normally.

```
BLOAD PRODOS,TSYS,A$2ØØØ
CALL -151
5228:ØØ
BSAVE PRODOS,TSYS,A$2ØØØ
```

The above patch, while it will work properly with undamaged blocks, is not advisable in normal use as it will fail to indicate when errors have occurred.

USING PRODOS WITH 40-TRACK DRIVES

The device driver supplied with ProDOS supports only 35 tracks. The code can be modified easily to support third party disk drives with 40 tracks, but there

are a couple of things to consider. The patch will apply to all drives (regardless of the number of tracks supported) connected to Disk II or compatible controller cards. This should cause no difficulties even if one 35-track and one 40-track drive are on the same controller card. Because you will also want to format 40-track disks, it will be necessary to modify FILER. The patch to FILER will apply to all disks that you format, and will produce an error if you attempt to format a disk on a drive supporting less than 40 tracks.

This patch modifies the Disk II Driver, which is a part of the "PRODOS" file (or "P8" file), so that it allows 320 blocks per volume instead of 280 blocks per volume. First set the prefix to the directory that contains the file you want to modify. This file will normally be called "PRODOS" on an 8-bit Apple II and "P8" on a 16-bit Apple IIGS. If the file name is not "PRODOS," substitute the correct filename wherever "PRODOS" appears.

```
UNLOCK PRODOS
BLOAD PRODOS,TSYS,A$2ØØØ
CALL -151
address*:4Ø
3DØG
BSAVE PRODOS,TSYS,A$2ØØØ
LOCK PRODOS
```

* "address" varies with the version of ProDOS, as follows:

ProDOS Version	address
1.Ø.1	52ØD
1.Ø.2	52CD
1.1.1	56E3
1.2	58E3
1.3	58E3

The following patch modifies the program FILER* to format 40 tracks instead of 35. After this modification is made, only 40-track drives may be formatted with FILER.

```
UNLOCK FILER
BLOAD FILER,TSYS,A$2ØØØ
CALL -151
addr**:4Ø
79F4:28
3DØG
BSAVE FILER,TSYS,A$2ØØØ
LOCK FILER
```

** "addr" depends on the release date of FILER. Here are the values of "addr" for two different

* Unlike the patch to ProDOS. this patch need not be applied to the disk. You may wish simply to make the patch and execute the program. Tu do this, replace 3D0G with 2000G, and don't BSAVE FILER. This patch works on the version of FILER released in 1984. It does not work with some pre-release versions, and may not work with future releases of FILER.

release dates:

```
Release date     address
1 JAN 84           4244
18 JUN 84          426A
```

FORCING PRODOS TO LOAD IN 48K

It is possible to load the ProDOS Kernel in main RAM (rather than in the bank switched memory or Language Card). In this case, you cannot use the BASIC Interpreter (since it is assembled for a fixed location which conflicts with the alternate location of the Kernel), or `EDASM.SYSTEM` from the toolkit package. You can, however, use other programs, such as the `EXERCISER` or `BUGBYTER`. Forcing a 48K load is sometimes useful even in a larger machine if you want to trace execution into the ProDOS Kernel itself using `BUGBYTER`. Under ordinary circumstances, as soon as the bank switched memory is enabled, the ROM monitor disappears and `BUGBYTER` goes berserk! If the Kernel is in main RAM, however, this does not occur. To force a 48K load you must first place a "`SYSTEM`" program (with type `SYS`) on the diskette to be booted (make a copy of `BUGBYTER` called `BUGBYTER.SYSTEM`). This must be the first file whose name ends with "`.SYSTEM`" in the Volume Directory. You can then apply the following patch and reboot.

```
BLOAD BUGBYTER
CREATE BUGBYTER.SYSTEM,TSYS
BSAVE BUGBYTER.SYSTEM,TSYS,A$2ØØØ,L7177
BLOAD PRODOS,TSYS,A$2ØØØ
CALL -151
23FC:A9 5Ø
BSAVE PRODOS,TSYS,A$2ØØØ
```

Note that the value of `$5Ø` above is the `MACHID` desired (Apple II Plus with 48K). You may add to that the bits necessary for an 80-column card or Thunderclock if you like. You may wish to append your own program to the `BUGBYTER.SYSTEM` image before `BSAVE`ing it so that it will be available to you once the 48K system is loaded. You can do this by inserting a `BLOAD MYPGM,A$3DØØ` after the `CREATE`, and changing the length of the `BSAVE` to accomodate `BUGBYTER` and your program combined. When you boot the 48K diskette, your program will be loaded at `$3DØØ`, immediately following `BUGBYTER`.

Chapter 8
ProDOS Global Pages

Readers of *Beneath Apple DOS* may remember that Chapter 8 of that book was devoted to a detailed analysis of DOS program logic. The contents of that chapter comprised one quarter of the book, and represented a complete description of more than 10K of machine language code. Two factors have led to the approach taken in *Beneath Apple ProDOS*. First, ProDOS code is expected to be much more volatile than that of DOS. If material similar to Chapter 8 in *Beneath Apple DOS* had been published here, it would have quickly become obsolete because of reassemblies of the operating system components by Apple. Throughout its lifetime, DOS was only completely reassembled once — when the change was made from 3.1 to 3.2 in 1979. Our book documented a form of DOS in which most of the instructions had not "moved" in nearly five years! In contrast, before *Beneath Apple ProDOS* was published, two different versions of ProDOS were already being distributed — 1.0.1 and 1.0.2. Although the differences between them are very minor, insertions of instructions and data have caused the shifting of major sections of code. Similar changes are expected in the future.

A second factor in the decision to shorten Chapter 8 involved the physical size of ProDOS. The equivalent components of ProDOS, compared to the DOS code covered in our earlier book, occupy over 22K of memory. A complete treatment of this code would be a book in and of itself. Coupled with the increased complexity of ProDOS which has resulted in longer chapters overall, as well as the previously mentioned volatility of the code, we decided that an in-depth coverage of ProDOS program logic did not belong here.

However, recognizing the importance of this material to many of our readers, a special supplement has been prepared that provides a detailed description of every piece of code and data within the major ProDOS components. It is available directly from Quality Software. Updated editions of this supplement will be available on a periodic basis as Apple releases new versions of ProDOS. In addition, any errata and changes to the main body of *Beneath Apple ProDOS* will be found in future supplements, eliminating the need to buy future editions of this book.

BASIC INTERPRETER GLOBAL PAGE

This page of memory is rigidly defined by the ProDOS BI. Fields given here will not move in later versions of ProDOS and may be referenced by external, user-written programs. Future additions to the global page may be made in areas which are marked "Not used".

Address	Label	Contents
BEØØ-BEØ2	BIENTRY	JMP to WARMDOS (BI warmstart vector).
BEØ3-BEØ5	DOSCMD	JMP to SYNTAX (BI command line parse and execute).
BEØ6-BEØ8	EXTRNCMD	JMP to user-installed external command parser.
BEØ9-BEØB	ERROUT	JMP to BI error handler.
BEØC-BEØE	PRINTERR	JUMP to BI error message print routine. Place error number in A-register.
BEØF	ERRCODE	ProDOS error code (also at $DE. Applesoft ONERR code).
BE1Ø-BE1F	OUTVEC	Default output vector in monitor and for each slot (1-7).
BE2Ø-BE2F	INVEC	Default input vector in monitor and for each slot (1-7).
BE3Ø-BE31	VECTOUT	Current output vector.
BE32-BE33	VECTIN	Current input vector
BE34-BE35	VDOSIO	BI's output intercept address.
BE36-BE37		BI's input intercept address.
BE38-BE3B	VSYSIO	BI's internal redirection by STATE.
BE3C	DEFSLT	Default slot.
BE3D	DEFDRV	Default drive.
BE3E	PREGA	A-register save area.
BE3F	PREGX	X-register save area.
BE4Ø	PREGY	Y-register save area.
BE41	DTRACE	Applesoft TRACE is enabled flag (MSB on).
BE42	STATE	Current intercept state. Ø = immediate command mode. > Ø = deferred.
BE43	EXACTV	EXEC file active flag (MSB on).
BE44	IFILACTV	READ file active flag (MSB on).
BE45	OFILACTV	WRITE file active flag (MSB on).
BE46	PFXACTV	PREFIX read active flag (MSB on).
BE47	DIRFLG	File being READ is a DIR file (MSB on).
BE48	EDIRFLG	End of directory flag (no longer used).
BE49	STRINGS	String space count used to determine when to garbage collect.
BE4A	TBUFPTR	Buffered WRITE data length.
BE4B	INPTR	Command line assembly length.

Address	Label	Contents		
BE4C	CHRLAST	Previous output character (for recursion check).		
BE4D	OPENCNT	Number of files open (not counting EXEC).		
BE4E	EXFILE	EXEC file being closed flag (MSB on).		
BE4F	CATFLAG	Line type to format next in DIR file READ.		
BE5Ø-BE51	XTRNADDR	External command handler address.		
BE52	XLEN	Length of command name (less one).		
BE53	XCNUM	Number of command:		
		$ØØ = external	$ØA = OPEN	$14 = WRITE
		$Ø1 = IN#	$ØB = READ	$15 = APPEND
		$Ø2 = PR#	$ØC = SAVE	$16 = CREATE
		$Ø3 = CAT	$ØD = BLOAD	$17 = DELETE
		$Ø4 = FRE	$ØE = BSAVE	$18 = PREFIX
		$Ø5 = RUN	$ØF = CHAIN	$19 = RENAME
		$Ø6 = BRUN	$1Ø = CLOSE	$1A = UNLOCK
		$Ø7 = EXEC	$11 = FLUSH	$1B = VERIFY
		$Ø8 = LOAD	$12 = NOMON	$1C = CATALOG
		$Ø9 = SAVE	$13 = STORE	$1D = RESTORE
				$1E = POSITION
BE54-BE55	PBITS	Permited command operand bits:		
		$8ØØØ Prefix needed. Pathname optional.		
		$4ØØØ Slot number only (PR# or IN#).		
		$2ØØØ Deferred command.		
		$1ØØØ File name optional.		
		$Ø8ØØ If file does not exist, create it.		
		$Ø4ØØ T: file type permitted.		
		$Ø2ØØ Second file name required.		
		$Ø1ØØ First file name required.		
		$ØØ8Ø AD: address keyword permitted.		
		$ØØ4Ø B: byte offset permitted		
		$ØØ2Ø E: ending address permitted.		
		$ØØ1Ø L: length permitted.		
		$ØØØ8 @: line number permitted.		
		$ØØØ4 S or D: slot/drive permitted.		
		$ØØØ2 F: field permitted.		
		$ØØØ1 R: record permitted.		
		(V always permitted but ignored).		
BE56-BE57	FBITS	Operands found on command line. Same bit assignments as above.		

Address	Label	Contents
BE58-BE59	VADDR	A keyword value.
BE5A-BE5C	VBYTE	B keyword value.
BE5D-BE5E	VENDA	E keyword value.
BE5F-BE60	VLNTH	L keyword value.
BE61	VSLOT	S keyword value.
BE62	VDRIV	D keyword value.
BE63-BE64	VFELD	F keyword value.
BE65-BE66	VRECD	R keyword value.
BE67	VVOLM	V keyword value (ignored).
BE68-BE69	VLINE	@ keyword value.
BE6A	VTYPE	T keyword value (in hex).
BE6B	VIOSLT	PR# or IN# slot number value.
BE6C-BE6D	VPATH1	Primary pathname buffer (address of length byte).
BE6E-BE6F	VPATH2	Secondary pathname buffer (address of length byte).
BE70-BE84	GOSYSTEM	Call the MLI using the parameter tables which follow.
BE85	SYSCALL	MLI call number for this call.
BE86-BE87	SYSPARM	Address of the MLI parameter list for this call.
BE88-BE8A		Return from MLI call.
BE8B-BE9E	BADCALL	MLI error return; translate error code to BI error number.
BE9F	BISPARE1	Not used.
BEA0-BEAB	SCREATE	CREATE parameter list.
BEAC-BEAE	SSGPRFX	GET_PREFIX, SET_PREFIX, DESTROY parameter list.
BEAF-BEB3	SRENAME	RENAME parameter list.
BEB4-BEC5	SSGINFO	GET_FILE_INFO, SET_FILE_INFO parameter list.
BEC6-BECA	SONLINE	ONLINE, SET_MARK, GET_MARK, SET_EOF, GET_EOF, SET_BUF, GET_BUF, QUIT parameter list.
BECB-BED0	SOPEN	OPEN parameter list.
BED1-BED4	SNEWLN	SET_NEWLINE parameter list.
BED5-BEDC	SREAD	READ, WRITE parameter list.
BEDD-BEDE	SCLOSE	CLOSE, FLUSH parameter list.
BEDF-BEF4	CCCSPARE	"COPYRIGHT APPLE, 1983"
BEF5-BEF7	GETBUFR	GETBUFR buffer allocation subroutine vector
BEF8-BEFA	FREEBUFR	FREEBUFR buffer free subroutine vector
BEFB		Original HIMEM MSB
BEFC-BEFF		Not used.

PRODOS SYSTEM GLOBAL PAGE — MLI GLOBAL PAGE

Portions of this page of memory are rigidly defined by the MLI and are unlikely to move in later versions of ProDOS. However, some portions are less stable and could change in future releases.

Address	Label	Contents
Jump Vectors		
BFØØ-BFØ2	ENTRY	JMP to MLI
BFØ3-BFØ5	JSPARE	Jump to system death code (via $BFF6)
BFØ6-BFØ8	DATETIME	Jump to Date/Time routine (RTS if no clock).
BFØ9-BFØB	SYSERR	JMP to system error handler.
BFØC-BFØE	SYSDEATH	JMP to system death handler.
BFØF	SERR	System error number.
Device Information		
BF1Ø-BF11	DEVADRØ1	Slot Ø reserved.
BF12-BF13	DEVADR11	Slot 1, drive 1 device driver address.
BF14-BF15	DEVADR21	Slot 2, drive 1 device driver address.
BF16-BF17	DEVADR31	Slot 3, drive 1 device driver address.
BF18-BF19	DEVADR41	Slot 4, drive 1 device driver address.
BF1A-BF1B	DEVADR51	Slot 5, drive 1 device driver address.
BF1C-BF1D	DEVADR61	Slot 6, drive 1 device driver address.
BF1E-BF1F	DEVADR71	Slot 7, drive 1 device driver address.
BF2Ø-BF21	DEVADRØ2	Slot Ø reserved.
BF22-BF23	DEVADR12	Slot 1, drive 2 device driver address.
BF24-BF25	DEVADR22	Slot 2, drive 2 device driver address.
BF26-BF27	DEVADR32	/RAM device driver address (need extra 64K).
BF28-BF29	DEVADR42	Slot 4, drive 2 device driver address.
BF2A-BF2B	DEVADR52	Slot 5, drive 2 device driver address.
BF2C-BF2D	DEVADR62	Slot 6, drive 2 device driver address.
BF2E-BF2F	DEVADR72	Slot 7, drive 2 device driver address.
BF3Ø	DEVNUM	Slot and drive (DSSSØØØØ) of last device.
BF31	DEVCNT	Count (minus 1) of active devices.
BF32-BF3F	DEVLST	List of active devices (slot, drive and identification—DSSSIIII).
BF4Ø-BF4F		Copyright notice.
BF5Ø-BF55	IRQXITX	Switch in language card and call IRQ handler at $FFD8.
BF56-BF57	TEMP	Temporary storage for IRQ code.
BF58-BF6F	BITMAP	Bitmap of low 48K of memory.

Address	Label	Contents
BF7Ø-BF71	BUFFER1	Open file 1 buffer address.
BF72-BF73	BUFFER2	Open file 2 buffer address.
BF74-BF75	BUFFER3	Open file 3 buffer address.
BF76-BF77	BUFFER4	Open file 4 buffer address.
BF78-BF79	BUFFER5	Open file 5 buffer address.
BF7A-BF7B	BUFFER6	Open file 6 buffer address.
BF7C-BF7D	BUFFER7	Open file 7 buffer address.
BF7E-BF7F	BUFFER8	Open file 8 buffer address.
Interrupt Information		
BF8Ø-BF81	INTRUPT1	Interrupt handler address (highest priority).
BF82-BF83	INTRUPT2	Interrupt handler address.
BF84-BF85	INTRUPT3	Interrupt handler address.
BF86-BF87	INTRUPT4	Interrupt handler address (lowest priority).
BF88	INTAREG	A-register save area.
BF89	INTXREG	X-register save area.
BF8A	INTYREG	Y-register save area.
BF8B	INTSREG	S-register save area.
BF8C	INTPREG	P-register save area.
BF8D	INTBANKID	Bank ID byte (ROM, RAM1, or RAM2).
BF8E-BF8F	INTADDR	Interrupt return address.
General System Information		
BF9Ø-BF91	DATE	YYYYYYYM MMMDDDDD.
BF92-BF93	TIME	...HHHHH ..MMMMMM.
BF94	LEVEL	Current file level.
BF95	BUBIT	Backup bit.
BF96-BF97	SPARE1	Currently unused.
BF98	MACHID	Machine ID Byte.
		ØØ.. Ø... II
		Ø1.. Ø... II+
		1Ø.. Ø... IIe
		11.. Ø... III emulation
		ØØ.. 1... Future expansion
		Ø1.. 1... Future expansion
		1Ø.. 1... IIc
		11.. 1... Future expansion
		..ØØ .... Unused
		..Ø1 .... 48K

Address	Label	Contents
		..1Ø 64K
		..11 .... 128K
		.... .X.. Reserved
		.... ..Ø. No 8Ø-column card
		.... ..1. 8Ø-column card present
		....Ø No compatible clock
		....1 Compatible clock present
BF99	SLTBYT	Slot ROM map (bit on indicates ROM present).
BF9A	PFIXPTR	Prefix flag (Ø indicates no active prefix).
BF9B	MLIACTV	MLI active flag (1... .... indicates active).
BF9C-BF9D	CMDADR	Last MLI call return address.
BF9E	SAVEX	X-register save area for MLI calls.
BF9F	SAVEY	Y-register save area for MLI calls.
Language Card Bank Switching Routines		
BFAØ-BFCF		Language card entry and exit routines.
BFAØ	EXIT	
BFAA	EXIT1	
BFB5	EXIT2	
BFB7	MLIENT1	
Interrupt Routines		
BFDØ-BFF3		Interrupt entry and exit routines.
BFDØ	IRQXIT	
BFDF	IRQXIT1	
BFE2	IRQXIT2	
BFE7	ROMXIT	
BFEB	IRQENT	
Data		
BFF4	BNKBYT1	Storage for byte at $EØØØ.
BFF5	BNKBYT2	Storage for byte at $DØØØ.
BFF6-BFFB		Switch on language card and call system death handler ($D1E4).
Version Information		
BFFC	IBAKVER	Minimum version of Kernel needed for this interpreter.
BFFD	IVERSION	Version number of this interpreter.
BFFE	KBAKVER	Minimum version of Kernel compatible with this Kernel.
BFFF	KVERSION	Version number of this Kernel.

The ProDOS garbage collector
starts his day

Appendix A

Example Programs

This section is intended to supply the reader with utility programs which can be used to examine and repair diskettes, as well as typical programming applications for ProDOS. These programs are provided in their source form to serve as examples of the programming necessary to interface practical programs to ProDOS. The reader who does not know assembly language may also benefit from these programs by entering them from the monitor in their binary form and saving them to disk for later use. The use of diskettes is assumed, although most of the programs will work with a hard disk or can be easily modified for this purpose. It is recommended that, until you are completely familiar with the operation of these programs, you should use them on an "expendable" diskette. None of the programs can physically damage a diskette, but they can, if used improperly, destroy the data on a diskette, requiring it to be reinitialized.

Seven programs are provided:

DUMP **TRACK DUMP UTILITY**

This is an example of how to access the disk drive directly through its I/O select addresses. `DUMP` may be used to dump to memory any given track in its raw, prenibblized form. This can be useful both in understanding how disks are formatted, and in diagnosing clobbered diskettes. `DUMP` will only operate on a Disk II drive or its equivalent.

FORMAT **REFORMAT A RANGE OF TRACKS**

This program will initialize a single track or a range of tracks on a diskette. `FORMAT` is useful in restoring a track whose sectoring has been damaged without reinitializing the entire diskette. `FORMAT` will only operate on a Disk II drive or its equivalent.

ZAP **DISK UPDATE UTILITY**

This program is the backbone of any attempt to patch a disk directory back together. It is also useful in examining the structure of files stored on disk and in applying patches to files or ProDOS directly. `ZAP` allows its user to read, and optionally write, any block on a disk volume. As such, it serves as a good example of a program which issues direct block I/O calls to the MLI.

MAP **MAP FREESPACE ON A VOLUME**

`MAP` is written in BASIC and proves that direct block I/O can be done directly from a BASIC program as well as from assembly language. `MAP` reads the volume freespace bit map and displays a map of freespace versus blocks in use on the screen.

FIB **FIND INDEX BLOCKS UTILITY**

`FIB` may be used when a directory for a volume has been destroyed. It searches every block on a volume for what appear to be index blocks, printing the block number location of each index block it finds. Knowing the locations of the index blocks and employing `ZAP`, the user can patch together a new directory.

TYPE **TYPE COMMAND**

The `TYPE` command may be added to the ProDOS BI as a new command. It allows a user to type (display) the contents of a file to the screen or a printer. `TYPE` serves as an example of an external command handler.

DUMBTERM **DUMB TERMINAL PROGRAM**

`DUMBTERM` serves as an example of programming with interrupts. It implements a simple terminal emulator program, using a CCS 7710 serial interface card. Interrupts are used to fill a circular buffer, allowing higher baud rates to be used.

STORING THE PROGRAMS ON DISKETTE

The enterprising programmer may wish to key in the source code for each program into an assembler and assemble the programs onto disk. The Merlin Pro Assembler was used to produce the listings presented here, and interested programmers should consult the documentation for that assembler for more information on the pseudo-opcodes used. For the non-assembly language programmer, the binary object code of each program may be entered from the monitor using the following procedure.

The assembly language listings consist of columns of information as follows.

1. The address of some object code
2. The object code which should be stored there
3. The statement number
4. The statement itself

For example,

```
2ØØØ:A9 Ø2    36 FIB    LDA #2 BLOCK = 2
```

indicates that the binary code "A9Ø2" should be stored at $2ØØØ and that this is statement 36. To enter a program in the monitor, the reader must type in each address and its corresponding object code. The following is an example of how to enter the FIB program.

```
CALL -151 (enter the monitor)
2ØØØ:A9 Ø2
2ØØ2:8D E9 2Ø
2ØØ5:A9 ØØ
2ØØ7:8D EA 2Ø
  ...etc...
2ØEB:ØØ ØØ
2ØED:ØØ ØØ
BSAVE FIB,A$2ØØØ,L$EF
```

Note that if a line (such as line 2 in FIB) has no object bytes associated with it, it may be ignored. Also, never type in a four digit hex number, such as the ones found in FIB on lines 22 through 27 or the "2Ø44" on line 41-type only two digit object code numbers.

When the program is to be run:

```
BLOAD FIB
CALL -151
2ØØØG
```

The BSAVE commands which must be used with the other programs are:

```
BSAVE DUMP,A$2ØØØ,L$1ØØ
BSAVE FORMAT,A$2ØØØ,L$51C
BSAVE ZAP,A$2ØØØ,L$47
BSAVE FIB,A$2ØØØ,L$EF
BSAVE DUMBTERM,A$2ØØØ,L$F7
BSAVE TYPE,A$2ØØØ,L$1B4
```

DUMP—TRACK DUMP UTILITY

The DUMP program will dump any track on a diskette in its raw, pre-nibbilized format, allowing the user to examine the sector address and data fields and the formatting of the track. This allows the inquisitive reader to examine his own diskettes to better understand the concepts presented in the preceeding chapters. DUMP may also be used to examine some protected disks to see how they differ from normal ones and to diagnose diskettes with clobbered sector address or data fields with the intention of recovering from disk I/O errors. The DUMP program serves as an example of direct use of the Disk II hardware from assembly language, with no use of ProDOS.

To use DUMP, first store the number of the track you wish dumped at location $2003, the device number you wish to use at location $2004 (the program defaults to slot 6, drive 1), and begin execution at $2000. DUMP will return to the monitor after displaying the first part of the track in hexadecimal on the screen. The entire track image is stored, starting at $4000. For example:

```
BLOAD DUMP (Load the DUMP program)
CALL -151 (Get into the monitor from BASIC)
```

(Now insert the diskette to be DUMPed)

```
2003:11 N 2000G (Store an 11 (track 17) in $2003, N
                terminates the store command, go to
                location $2000)
```

The output might look like this...

```
4000- D5 AA 96 AA AB AA BB AB (Start of sector address)
4008- AA AB BA DE AA E8 C0 FF
4510- 9E FF FF FF FF FF D5 AA (Start Of sector data)
4018- AD AE B2 9D AC AE 96 96 (Sector data)
  ...etc.
```

Quite often, a sector with an I/O error has only one bit which is in error, either in the address or data fields. A particularly patient programmer in some circumstances can determine the location of the error and devise a means to correct it.

```
1    *      DUMP -- TRACK DUMP UTILITY.
2             ORG   $2000

4    *********************************************************
5    *                                                       *
6    * DUMP:THIS PROGRAM WILL ALLOW USER TO DUMP AN ENTIRE   *
7    *  TRACK IN ITS RAW FORM INTO MEMORY FOR EXAMINATION.   *
8    *                                                       *
```

```
                 9    * INPUT: $2ØØ3 = TRACK TO BE READ (DEFAULTS TO $ØØ)      *
                 1Ø   *        $2ØØ4 = UNIT NUMBER (DEFAULTS TO $6Ø)           *
                 11   *                                                        *
                 12   * OUTPUT:$4ØØØ = ADDRESS OF TRACK IMAGE                  *
                 13   *                                                        *
                 14   * ENTRY POINT: $2ØØØ                                     *
                 15   *                                                        *
                 16   * PROGRAMMER: PIETER M LECHNER 5/29/84                   *
                 17   *                                                        *
                 18   **********************************************************

                 2Ø   *  ZPAGE DEFINITIONS

                 22   PTR      EQU   $Ø          WORK POINTER
                 23   A1L      EQU   $3C         MONITOR POINTER
                 24   A2L      EQU   $3E         MONITOR POINTER

                 26   *  OTHER ADDRESSES

                 28   BUFFER   EQU   $4ØØØ       TRACK IMAGE AREA
                 29   DELAY    EQU   $FCA8       MONITOR DELAY ROUTINE
                 3Ø   XAM      EQU   $FDB3       MONITOR HEX DUMP SUBRTN

                 32   *  DISK I/O SELECTS

                 34   DRVSMØ   EQU   $CØ8Ø       STEP MOTOR POSITIONS
                 35   DRVSM1   EQU   $CØ81
                 36   DRVSM2   EQU   $CØ82
                 37   DRVSM4   EQU   $CØ84
                 38   DRVSM6   EQU   $CØ86
                 39   DRVOFF   EQU   $CØ88       TURN DRIVE OFF AFTER 6 REVS
                 4Ø   DRVON    EQU   $CØ89       TURN DRIVE ON
                 41   DRVSL1   EQU   $CØ8A       SELECT DRIVE 1
                 42   DRVRD    EQU   $CØ8C       READ DATA LATCH
                 43   DRVRDM   EQU   $CØ8E       SET READ MODE

                 45   *  RECALIBRATE AND POSITION THE ARM TO THE DESIRED TRACK

2ØØØ: 4C ØA 2Ø   47   ENTRY    JMP   START       SKIP DATA

2ØØ3: ØØ         49   TRACK    DFB   $ØØ         TRACK TO DUMP
2ØØ4: 6Ø         5Ø   UNITNUM  DFB   $6Ø         UNIT NUMBER TO USE
2ØØ5: 6Ø         51   SLOT     DFB   $6Ø         SLOT NUMBER TO USE
2ØØ6: ØØ         52   DESTRK   DFB   $ØØ         DESTINATION TRACK
2ØØ7: ØØ         53   CURTRK   DFB   $ØØ         CURRENT TRACK
2ØØ8: ØØ         54   DELTA    DFB   $ØØ         NUMBER OF TRACKS TO MOVE
2ØØ9: ØØ         55   FLAG     DFB   $ØØ         DIRECTION & ODD/EVEN FLAGS

2ØØA: AD Ø4 2Ø   57   START    LDA   UNITNUM     GET UNIT NUMBER
2ØØD: 48         58            PHA   SAVE        FOR LATER
2ØØE: 29 7Ø      59            AND   #$7Ø        GET SLOT ONLY
2Ø1Ø: 8D Ø5 2Ø   6Ø            STA   SLOT
2Ø13: AA         61            TAX   PUT         SLOT IN X REG
2Ø14: 68         62            PLA
2Ø15: 1Ø Ø1      63            BPL   DRIVE1      SELECT DRIVE 1
2Ø17: E8         64            INX   SELECT      DRIVE 2
2Ø18: BD 8A CØ   65   DRIVE1   LDA   DRVSL1,X    SELECT APPROPRIATE DRIVE
2Ø1B: AE Ø5 2Ø   66            LDX   SLOT        GET SLOT
2Ø1E: BD 89 CØ   67            LDA   DRVON,X     TURN DRIVE ON
2Ø21: BD 8E CØ   68            LDA   DRVRDM,X    INSURE READ MODE

2Ø24: 2Ø 82 2Ø   7Ø            JSR   RECALC      MOVE ARM TO TRACK Ø

2Ø27: AD Ø3 2Ø   72            LDA   TRACK       GET TRACK TO READ
2Ø2A: 8D Ø6 2Ø   73            STA   DESTRK
```

```
2Ø2D: 2Ø 9F 2Ø  74            JSR   ARMOVE     GO THERE

                76    *  PREPARE TO DUMP TRACK TO MEMORY

2Ø3Ø: A9 ØØ     78            LDA   #<BUFFER   POINT AT DATA
2Ø32: 85 ØØ     79            STA   PTR
2Ø34: A9 4Ø     8Ø            LDA   #>BUFFER
2Ø36: 85 Ø1     81            STA   PTR+1
2Ø38: AØ ØØ     82            LDY   #Ø

                84    *  START DUMPING AT THE BEGINNING OF A SECTOR ADDRESS
                85    *  FIELD OR A SECTOR DATA FIELD

2Ø3A: AE Ø5 2Ø  87            LDX   SLOT
2Ø3D: BD 8C CØ  88    LOOP1   LDA   DRVRD,X    WAIT FOR NEXT BYTE
2Ø4Ø: 1Ø FB     89            BPL   LOOP1
2Ø42: C9 FF     9Ø            CMP   #$FF       AUTOSYNC?
2Ø44: DØ F7     91            BNE   LOOP1      NO, DON'T START IN MIDDLE
2Ø46: BD 8C CØ  92    LOOP2   LDA   DRVRD,X    WAIT FOR NEXT BYTE
2Ø49: 1Ø FB     93            BPL   LOOP2
2Ø4B: C9 FF     94            CMP   #$FF       TWO AUTOSYNCS?
2Ø4D: DØ EE     95            BNE   LOOP1      NOT YET
2Ø4F: BD 8C CØ  96    LOOP3   LDA   DRVRD,X
2Ø52: 1Ø FB     97            BPL   LOOP3
2Ø54: C9 FF     98            CMP   #$FF       STILL AUTOSYNCS?
2Ø56: FØ F7     99            BEQ   LOOP3      YES, WAIT FOR DATA BYTE
2Ø58: DØ Ø5     1ØØ           BNE   LOOP4      ELSE, START STORING DATA

                1Ø2   *  ONCE ALIGNED, BEGIN COPYING THE TRACK TO MEMORY.
                1Ø3   *  COPY AT LEAST TWICE ITS LENGTH TO INSURE WE GET IT
                1Ø4   *  ALL.

2Ø5A: BD 8C CØ  1Ø6   LOOPD   LDA   DRVRD,X    WAIT FOR NEXT DATA BYTE
2Ø5D: 1Ø FB     1Ø7           BPL   LOOPD
2Ø5F: 91 ØØ     1Ø8   LOOP4   STA   (PTR),Y    STORE IN MEMORY
2Ø61: E6 ØØ     1Ø9           INC   PTR        BUMP POINTER
2Ø63: DØ F5     11Ø           BNE   LOOPD
2Ø65: E6 Ø1     111           INC   PTR+1      DONE $4ØØØ BYTES YET?
2Ø67: 1Ø F1     112           BPL   LOOPD      NO, CONTINUE
2Ø69: AE Ø5 2Ø  113           LDX   SLOT
2Ø6C: BD 88 CØ  114           LDA   DRVOFF,X   TURN DRIVE OFF

                116   *  WHEN FINISHED, DUMP SOME OF TRACK IN HEX ON SCREEN

2Ø6F: A9 ØØ     118           LDA   #<BUFFER   DUMP 4ØØØ.4ØAF
2Ø71: 85 3C     119           STA   A1L
2Ø73: A9 4Ø     12Ø           LDA   #>BUFFER
2Ø75: 85 3D     121           STA   A1L+1
2Ø77: A9 AF     122           LDA   #<BUFFER+$AF
2Ø79: 85 3E     123           STA   A2L
2Ø7B: A9 4Ø     124           LDA   #>BUFFER+$AF
2Ø7D: 85 3F     125           STA   A2L+1
2Ø7F: 4C B3 FD  126           JMP   XAM        EXIT VIA HEX DISPLAY

                128   *  RECALIBRATE ARM

2Ø82: A9 3Ø     13Ø   RECALC  LDA   #$3Ø       PRETENT TO BE ON TRACK 48
2Ø84: 8D Ø7 2Ø  131           STA   CURTRK
2Ø87: A9 ØØ     132           LDA   #$ØØ       SELECT TRACK ØØ
2Ø89: 8D Ø6 2Ø  133           STA   DESTRK
2Ø8C: 2Ø 9F 2Ø  134           JSR   ARMOVE     GO THERE
2Ø8F: AE Ø5 2Ø  135           LDX   SLOT       GET SLOT NUMBER
2Ø92: BD 8Ø CØ  136           LDA   DRVSMØ,X   TURN ALL PHASES OFF
2Ø95: BD 82 CØ  137           LDA   DRVSM2,X
2Ø98: BD 84 CØ  138           LDA   DRVSM4,X
```

```
2Ø9B: BD 86 CØ  139          LDA   DRVSM6,X
2Ø9E: 6Ø        14Ø          RTS   RETURN     TO CALLER

                142  *  ARM MOVE ROUTINE

2Ø9F: A9 ØØ     144  ARMOVE  LDA   #$ØØ
2ØA1: 8D Ø9 2Ø  145          STA   FLAG       INITIALIZE FLAG
2ØA4: AD Ø7 2Ø  146          LDA   CURTRK     GET CURRENT TRACK
2ØA7: 38        147          SEC
2ØA8: ED Ø6 2Ø  148          SBC   DESTRK     SUBTRACT DESTINATION TRACK
2ØAB: FØ 36     149          BEQ   DONE       IF EQUAL THEN EXIT
2ØAD: BØ Ø4     15Ø          BCS   OK         POSITIVE RESULT? YES, GO ON
2ØAF: 49 FF     151          EOR   #$FF       MAKE RESULT POSITIVE
2ØB1: 69 Ø1     152          ADC   #$Ø1
2ØB3: 8D Ø8 2Ø  153  OK      STA   DELTA      SAVE RESULT
2ØB6: 2E Ø9 2Ø  154          ROL   FLAG       SET IN/OUT FLAG
2ØB9: 4E Ø7 2Ø  155          LSR   CURTRK     ON ODD OR EVEN TRACK?
2ØBC: 2E Ø9 2Ø  156          ROL   FLAG       PUT RESULT IN FLAG
2ØBF: ØE Ø9 2Ø  157          ASL   FLAG       ADJUST FOR TABLE OFFSET
2ØC2: AC Ø9 2Ø  158          LDY   FLAG       GET TABLE OFFSET
2ØC5: B9 F8 2Ø  159  LOOP    LDA   TABLE,Y    GET PHASE TO TURN ON
2ØC8: 2Ø E4 2Ø  16Ø          JSR   PHASE
2ØCB: B9 F9 2Ø  161          LDA   TABLE+1,Y  GET NEXT PHASE TO TURN ON
2ØCE: 2Ø E4 2Ø  162          JSR   PHASE
2ØD1: 98        163          TYA
2ØD2: 49 Ø2     164          EOR   #$Ø2       ADJUST OFFSET
2ØD4: A8        165          TAY
2ØD5: CE Ø8 2Ø  166          DEC   DELTA      DECREMENT NUMBER OF TRACKS TO MOVE
2ØD8: AD Ø8 2Ø  167          LDA   DELTA
2ØDB: DØ E8     168          BNE   LOOP       IF NOT DONE, DO ANOTHER
2ØDD: AD Ø6 2Ø  169          LDA   DESTRK     UPDATE CURRENT TRACK WITH
2ØEØ: 8D Ø7 2Ø  17Ø          STA   CURTRK     WHERE THE ARM IS NOW
2ØE3: 6Ø        171  DONE    RTS   DONE,      RETURN TO CALLER

                173  * TURN A PHASE ON, WAIT THEN TURN IT OFF

2ØE4: ØD Ø5 2Ø  175  PHASE   ORA   SLOT       ADD SLOT TO PHASE
2ØE7: AA        176          TAX
2ØE8: BD 81 CØ  177          LDA   DRVSM1,X   TURN ON A PHASE
2ØEB: 2Ø F2 2Ø  178          JSR   WAIT       WAIT FOR ARM TO SETTLE
2ØEE: BD 8Ø CØ  179          LDA   DRVSMØ,X   TURN OFF PHASE
2ØF1: 6Ø        18Ø          RTS   RETURN     TO CALLER

                182  * 2Ø MILLISECOND DELAY ROUTINE

2ØF2: A9 56     184  WAIT    LDA   #$56       WAIT ABOUT 2Ø MILLISECONDS
2ØF4: 2Ø A8 FC  185          JSR   DELAY
2ØF7: 6Ø        186          RTS   RETURN     TO CALLER

                188  * PHASE TABLE

2ØF8: Ø2 Ø4 Ø6  19Ø  TABLE   DFB   $Ø2,$Ø4,$Ø6,$ØØ
2ØFB: ØØ
2ØFC: Ø6 Ø4 Ø2  191          DFB   $Ø6,$Ø4,$Ø2,$ØØ
2ØFF: ØØ

 --End assembly, 256 bytes, Errors: Ø
```

FORMAT—REFORMAT A RANGE OF TRACKS

FORMAT can be used to selectively format a single track, a range of tracks or an entire diskette. While it is primarily meant to be educational, it can assist in repairing damaged diskettes. For example, if a single sector was damaged, it could be repaired by FORMATting the particular track on which it resides. To avoid losing data, all other sectors on the track should be read and copied to another diskette prior to reFORMATting. After FORMAT is run, they can be copied back to the repaired diskette and data can be written to the previously damaged sector.

Note that FORMAT has very limited error handling capabilities; in addition, it may not work well on drives that are out of adjustment (too fast or slow). The method used to do the formatting, that of building an image of the track in memory and then writing that image to the diskette, is similar to the method used by "nibble" copy programs.

To run FORMAT, store the starting track number at location $2003, the ending track number at location $2004, the volume number at location $2005, and the device number at location $2006, then begin execution at $2000. FORMAT will return to the monitor upon completion. If a track cannot be formatted for some reason (e.g., physical damage, etc.), an error will be indicated. For example:

```
BLOAD FORMAT (Load the FORMAT program)
CALL -151 (Get into the monitor from BASIC)
```

(Now insert the diskette to be FORMATted)

```
2003:11 11 FE 60 N 2000G (Store an 11 (track 17) in $2003
                         store an 11 in $2004, store an FE
                         (volume 254) in $2005, Store a
                         60 (slot 6 drive 1) in $2006, N
                         terminates the store command, go
                         to location $2000)
```

The output might look like this:

```
FORMATTING TRACK 22
```

WARNING: FORMAT will destroy existing data on the diskette in the indicated drive without allowing the user an opportunity to abort the program. Be sure the diskette in the drive is the one you wish to FORMAT.

```
                 1             ORG   $2000

                 3    ********************************************************
                 4    *                                                      *
                 5    * FORMAT: THIS PROGRAM WILL INITIALIZE A RANGE OF      *
                 6    *         TRACKS WITH ANY VOLUME NUMBER DESIRED.       *
                 7    *                                                      *
                 8    * INPUT: $2003 = FIRST TRACK TO BE INITIALIZED         *
                 9    *                DEFAULT ($00)                         *
                 10   *        $2004 = LAST TRACK TO BE INITILIZED           *
                 11   *                DEFAULT ($22)                         *
                 12   *        $2005 = VOLUME NUMBER                         *
                 13   *                DEFAULT ($FE)                         *
                 14   *        $2006 = UNIT NUMBER                           *
                 15   *                DEFAULT ($60)                         *
                 16   *                                                      *
                 17   * ENTRY POINT: $2000                                   *
                 18   *                                                      *
                 19   * PROGRAMMER: PIETER LECHNER 5/19/84                   *
                 20   *                                                      *
                 21   ********************************************************

                 23   *  ZPAGE DEFINITIONS

                 25   PTR      EQU   $0          WORK POINTER
                 26   A1       EQU   $3C         MONITOR POINTERS
                 27   A2       EQU   $3E                 |
                 28   A4       EQU   $42                 |

                 30   *  MONITOR ROUTINES

                 32   BS       EQU   $FC10       BACKSPACE
                 33   HOME     EQU   $FC58       CLEAR SCREEN
                 34   DELAY    EQU   $FCA8       DELAY ROUTINE
                 35   NXTA4    EQU   $FCB4       INCREMENT POINTERS (A4/A1)
                 36   NXTA1    EQU   $FCBA       INCREMENT POINTER (A1)
                 37   COUT     EQU   $FDED       CHARACTER OUTPUT
                 38   PRBYTE   EQU   $FDDA       HEX OUTPUT
                 39   MOVE     EQU   $FE2C       MOVE ROUTINE

                 41   *  DISK I/O SELECTS

                 43   DRVSM0   EQU   $C080       STEP MOTOR POSITIONS
                 44   DRVSM1   EQU   $C081                |
                 45   DRVSM2   EQU   $C082                |
                 46   DRVSM4   EQU   $C084                |
                 47   DRVSM6   EQU   $C086                |
                 48   DRVOFF   EQU   $C088       TURN DRIVE OFF
                 49   DRVON    EQU   $C089       TURN DRIVE ON
                 50   DRVSL1   EQU   $C08A       SELECT DRIVE 1
                 51   DRVSL2   EQU   $C08B       SELECT DRIVE 2
                 52   DRVRD    EQU   $C08C       READ DATA LATCH
                 53   DRVWR    EQU   $C08D       WRITE DATA LATCH
                 54   DRVRDM   EQU   $C08E       SET READ MODE
                 55   DRVWRM   EQU   $C08F       SET WRITE MODE

                 57   *  FORMAT PROGRAM

2000: 4C 07 20   59   FORMAT   JMP   ENTRY       SKIP OVER DATA

2003: 00         61   STRK     DFB   $00         STARTING TRACK
2004: 22         62   ETRK     DFB   $22         ENDING TRACK
2005: FE         63   VOLNUM   DFB   $FE         VOLUME NUMBER
2006: 60         64   UNITNUM  DFB   $60         UNIT NUMBER
```

```
                 66   *  RECALIBRATE ARM TO THE DESIRED TRACK

2ØØ7: 2Ø F7 22  68   ENTRY    JSR   SCREEN     CLEAR SCREEN/DISPLAY MESSAGE
2ØØA: AD Ø6 2Ø  69            LDA   UNITNUM    GET UNIT NUMBER
2ØØD: 48        7Ø            PHA   SAVE       FOR LATER
2ØØE: 29 7Ø     71            AND   #$7Ø       GET SLOT ONLY
2Ø1Ø: 8D D6 24  72            STA   SLOT
2Ø13: AA        73            TAX   PUT        SLOT IN X REG
2Ø14: 68        74            PLA   CHECK      WHICH DRIVE TO USE
2Ø15: 1Ø Ø1     75            BPL   DRIVE1     SELECT DRIVE 1
2Ø17: E8        76            INX   SELECT     DRIVE 2
2Ø18: BD 8A CØ  77   DRIVE1   LDA   DRVSL1,X   SELECT APPROPRIATE DRIVE
2Ø1B: AE D6 24  78            LDX   SLOT       GET SLOT
2Ø1E: BD 89 CØ  79            LDA   DRVON,X    TURN DRIVE ON
2Ø21: BD 8E CØ  8Ø            LDA   DRVRDM,X   INSURE READ MODE
2Ø24: BD 8C CØ  81            LDA   DRVRD,X

2Ø27: 2Ø 79 22  83            JSR   RECALC     RECALIBRATE ARM
2Ø2A: AD Ø3 2Ø  84            LDA   STRK       GET STARTING TRACK
2Ø2D: 8D D9 24  85            STA   DESTRK
2Ø3Ø: 2Ø 96 22  86            JSR   ARMOVE     GO THERE

                 88   *  BUILD TRACK IMAGE IN MEMORY

2Ø33: 2Ø 3B 21  9Ø            JSR   FILLDATA   FILL DATA AREA WITH $96'S
2Ø36: AD Ø3 2Ø  91            LDA   STRK
2Ø39: 8D D7 24  92            STA   TRACK      INITIALIZE TRACK NUMBER
2Ø3C: A9 Ø5     93            LDA   #$Ø5
2Ø3E: 8D CE 24  94            STA   RETRYCNT   INITIALIZE RETRY COUNT
2Ø41: A9 CB     95            LDA   #<END
2Ø43: 8D D2 24  96            STA   CUREND
2Ø46: A9 24     97            LDA   #>END      SAVE CURRENT LENGTH
2Ø48: 8D D3 24  98            STA   CUREND+1   OF SECTOR IMAGE
2Ø4B: 38        99   LOOPA    SEC
2Ø4C: AD D2 24  1ØØ           LDA   CUREND     COMPUTE LENGTH OF
2Ø4F: E9 4Ø     1Ø1           SBC   #<IMAGE    CURRENT SECTOR IMAGE
2Ø51: 8D DØ 24  1Ø2           STA   LENGTH
2Ø54: AD D3 24  1Ø3           LDA   CUREND+1
2Ø57: E9 23     1Ø4           SBC   #>IMAGE
2Ø59: 8D D1 24  1Ø5           STA   LENGTH+1
2Ø5C: EE DØ 24  1Ø6           INC   LENGTH
2Ø5F: 2Ø 62 21  1Ø7           JSR   FINDSTART  COMPUTE START OF TRACK IMAGE
2Ø62: 2Ø 83 21  1Ø8           JSR   BLDGAP1    BUILD GAP1 (128 BYTES)
2Ø65: 2Ø 9F 21  1Ø9           JSR   BLDTRK     BUILD TRACK IMAGE IN MEMORY
2Ø68: 2Ø C9 21  11Ø  ANOTHER  JSR   FIXTRK     UPDATE ADDRESS INFO

                 112  *  INITIALIZE TRACK

2Ø6B: 2Ø Ø6 23  114           JSR   PRTRK      PRINT TRACK NUMBER
2Ø6E: 2Ø B5 2Ø  115           JSR   WRITE      WRITE A TRACK
2Ø71: BØ 1F     116           BCS   ERROR1     WRITE PROTECT ERROR
2Ø73: 2Ø 31 21  117           JSR   VERIFY     VERIFY THE TRACK
2Ø76: 9Ø 2Ø     118           BCC   NEXT       IF OK DO ANOTHER
2Ø78: 38        119           SEC   ELSE       ADJUST GAP SIZE
2Ø79: AD D2 24  12Ø           LDA   CUREND
2Ø7C: E9 Ø2     121           SBC   #$Ø2
2Ø7E: 8D D2 24  122           STA   CUREND
2Ø81: AD D3 24  123           LDA   CUREND+1
2Ø84: E9 ØØ     124           SBC   #$ØØ
2Ø86: 8D D3 24  125           STA   CUREND+1
2Ø89: CE CE 24  126           DEC   RETRYCNT   DECREMENT RETRY COUNT
2Ø8C: 1Ø BD     127           BPL   LOOPA      IF OK TRY AGAIN

                 129  *  ERROR OCCURED
```

```
2Ø8E: A9 Ø2     131          LDA   #$Ø2
2Ø9Ø: DØ Ø2     132          BNE   ERR2
2Ø92: A9 Ø1     133 ERROR1   LDA   #$Ø1
2Ø94: 2Ø 13 23  134 ERR2     JSR   ERRHDL     PRINT ERROR MESSAGE
2Ø97: 6Ø        135          RTS   EXIT       PROGRAM

2Ø98: EE D7 24  137 NEXT     INC   TRACK      INCREMENT TRACK NUMBER
2Ø9B: AD D7 24  138          LDA   TRACK
2Ø9E: CD Ø4 2Ø  139          CMP   ETRK       COMPAIR WITH LAST TRACK TO DO
2ØA1: FØ Ø2     14Ø          BEQ   LASTONE    IF EQUAL DO LAST ONE
2ØA3: BØ Ø9     141          BCS   FINISH     IF DONE THEN EXIT PROGRAM
2ØA5: 8D D9 24  142 LASTONE  STA   DESTRK
2ØA8: 2Ø 96 22  143          JSR   ARMOVE     GO TO DESIRED TRACK
2ØAB: 4C 68 2Ø  144          JMP   ANOTHER    GO BACK AND DO ANOTHER

                146 *  WHEN DONE, EXIT

2ØAE: AE D6 24  148 FINISH   LDX   SLOT       GET SLOT
2ØB1: BD 88 CØ  149          LDA   DRVOFF,X   TURN DRIVE OFF
2ØB4: 6Ø        15Ø          RTS   EXIT       PROGRAM

                152 *  WRITE MEMORY TO DISK

2ØB5: A9 ØØ     154 WRITE    LDA   #$ØØ
2ØB7: 85 ØØ     155          STA   PTR
2ØB9: AD D5 24  156          LDA   START+1    POINT AT START OF DATA
2ØBC: 85 Ø1     157          STA   PTR+1
2ØBE: AC D4 24  158          LDY   START
2ØC1: AE D6 24  159          LDX   SLOT
2ØC4: 38        16Ø          SEC   ASSUME     ERROR
2ØC5: BD 8D CØ  161          LDA   DRVWR,X
2ØC8: BD 8E CØ  162          LDA   DRVRDM,X   CHECK WRITE PROTECT STATUS
2ØCB: 3Ø 3Ø     163          BMI   WPERR
2ØCD: A9 FF     164          LDA   #$FF
2ØCF: 9D 8F CØ  165          STA   DRVWRM,X   WRITE 1 $FF
2ØD2: DD 8C CØ  166          CMP   DRVRD,X
2ØD5: EA        167          NOP
2ØD6: 4C EØ 2Ø  168          JMP   LOOP1
2ØD9: 49 8Ø     169 ASYNC    EOR   #$8Ø       TURN HIGH BIT ON
2ØDB: EA        17Ø          NOP
2ØDC: EA        171          NOP
2ØDD: 4C E9 2Ø  172          JMP   WRIT       DELAY EXTRA 8 CYCLES
2ØEØ: 48        173 LOOP1    PHA
2ØE1: 68        174          PLA
2ØE2: B1 ØØ     175 LOOP2    LDA   (PTR),Y    GET BYTE TO WRITE
2ØE4: C9 8Ø     176          CMP   #$8Ø       IS IT "SYNC" BYTE
2ØE6: 9Ø F1     177          BCC   ASYNC      YES, THEN MAKE ADJUSTMENT
2ØE8: EA        178          NOP
2ØE9: 9D 8D CØ  179 WRIT     STA   DRVWR,X    WRITE A BYTE
2ØEC: DD 8C CØ  18Ø          CMP   DRVRD,X
2ØEF: C8        181          INY   INCREMENT  OFFSET
2ØFØ: DØ EE     182          BNE   LOOP1
2ØF2: E6 Ø1     183          INC   PTR+1      INCREMENT POINTER
2ØF4: 1Ø EC     184          BPL   LOOP2
2ØF6: BD 8E CØ  185          LDA   DRVRDM,X
2ØF9: BD 8C CØ  186          LDA   DRVRD,X    PUT IN READ MODE
2ØFC: 18        187          CLC
2ØFD: 6Ø        188 WPERR    RTS

                19Ø *  PREPARE TO DUMP TRACK TO MEMORY

2ØFE: A9 ØØ     192 DUMP     LDA   #$ØØ       POINT AT DATA
21ØØ: 85 ØØ     193          STA   PTR
21Ø2: A9 4Ø     194          LDA   #$4Ø
21Ø4: 85 Ø1     195          STA   PTR+1
```

```
21Ø6: AØ ØØ     196           LDY   #Ø

                198  *  START DUMPING AS SOON AS TWO SYNC BYTES FOUND

21Ø8: AE D6 24  2ØØ           LDX   SLOT
21ØB: BD 8C CØ  2Ø1  LOOP3    LDA   DRVRD,X    WAIT FOR NEXT BYTE
21ØE: 1Ø FB     2Ø2           BPL   LOOP3
211Ø: C9 FF     2Ø3           CMP   #$FF       AUTOSYNC?
2112: DØ F7     2Ø4           BNE   LOOP3      NO, DON'T START YET
2114: BD 8C CØ  2Ø5  LOOP4    LDA   DRVRD,X    WAIT FOR NEXT BYTE
2117: 1Ø FB     2Ø6           BPL   LOOP4
2119: C9 FF     2Ø7           CMP   #$FF       TWO AUTOSYNCS?
211B: DØ EE     2Ø8           BNE   LOOP3      NOT YET

                21Ø  *  ONCE ALIGNED, BEGIN COPYING THE TRACK TO MEMORY.
                211  *  COPY ENOUGH TO INSURE WE GET IT ALL

211D: BD 8C CØ  213  LOOPD    LDA   DRVRD,X    WAIT FOR NEXT DATA BYTE
212Ø: 1Ø FB     214           BPL   LOOPD
2122: 91 ØØ     215  LOOP5    STA   (PTR),Y    STORE IN MEMORY
2124: E6 ØØ     216           INC   PTR        BUMP POINTER
2126: DØ F5     217           BNE   LOOPD
2128: E6 Ø1     218           INC   PTR+1      DONE $2ØØØ BYTES YET?
212A: A5 Ø1     219           LDA   PTR+1
212C: C9 6Ø     22Ø           CMP   #$6Ø       NO, CONTINUE
212E: DØ ED     221           BNE   LOOPD
213Ø: 6Ø        222           RTS

                224  *  READ SECTOR ZERO TO VERIFY FORMATTING

2131: 2Ø FE 2Ø  226  VERIFY   JSR   DUMP       DUMP TRACK TO MEMORY
2134: 2Ø F1 21  227           JSR   SETUP
2137: 2Ø 39 22  228           JSR   COMPARE    CHECK WHAT WE JUST WROTE
213A: 6Ø        229           RTS

                231  *  FILL DATA AREA WITH $96'S

213B: 18        233  FILLDATA CLC
213C: A9 57     234           LDA   #<DATA
213E: 85 3C     235           STA   A1         SET A1 TO DATA START
214Ø: 69 56     236           ADC   #<DATALTH
2142: 85 3E     237           STA   A2         SET A1 TO DATA END
2144: A9 23     238           LDA   #>DATA
2146: 85 3D     239           STA   A1+1
2148: 69 Ø1     24Ø           ADC   #>DATALTH
214A: 85 3F     241           STA   A2+1
214C: A9 96     242           LDA   #$96
214E: 8D CF 24  243           STA   BYTE       INDICATE FILL BYTE
2151: 2Ø 55 21  244           JSR   FILL       CALL FILL ROUTINE
2154: 6Ø        245           RTS

                247  *  FILL MEMORY WITH A CONSTANT

2155: AØ ØØ     249  FILL     LDY   #$ØØ       INITIALIZE OFFSET
2157: AD CF 24  25Ø  LOOP     LDA   BYTE       GET BYTE TO USE
215A: 91 3C     251           STA   (A1),Y     STORE A BYTE
215C: 2Ø BA FC  252           JSR   NXTA1      CALL MONITOR INCREMENT
215F: 9Ø F6     253           BCC   LOOP       LOOP UNTIL DONE
2161: 6Ø        254           RTS

                256  * COMPUTE START OF TRACK IMAGE

2162: AØ Ø3     258  FINDSTART LDY  #$Ø3
2164: AD DØ 24  259           LDA   LENGTH     MULTIPLY LENGTH BY 16
2167: ØA        26Ø  MORE     ASL   A
```

```
2168: 2E D1 24  261           ROL   LENGTH+1
216B: 88        262           DEY
216C: 1Ø F9     263           BPL   MORE
216E: 8D DØ 24  264           STA   LENGTH

2171: 38        266           SEC   SUBTRACT    IT FROM $7F8Ø
2172: A9 8Ø     267           LDA   #$8Ø        TO FIND START
2174: ED DØ 24  268           SBC   LENGTH
2177: 8D D4 24  269           STA   START
217A: A9 7F     27Ø           LDA   #$7F
217C: ED D1 24  271           SBC   LENGTH+1
217F: 8D D5 24  272           STA   START+1
2182: 6Ø        273           RTS

                275  *  BUILD GAP1 AT START OF TRACK IMAGE

2183: 18        277  BLDGAP1  CLC
2184: AD D4 24  278           LDA   START
2187: 85 3C     279           STA   A1          SET A1 TO START
2189: 69 8Ø     28Ø           ADC   #$8Ø
218B: 85 3E     281           STA   A2          SET A2 TO START + $8Ø
218D: AD D5 24  282           LDA   START+1
219Ø: 85 3D     283           STA   A1+1
2192: 69 ØØ     284           ADC   #$ØØ
2194: 85 3F     285           STA   A2+1
2196: A9 7F     286           LDA   #$7F        USE $7F FOR SYNC BYTE
2198: 8D CF 24  287           STA   BYTE
219B: 2Ø 55 21  288           JSR   FILL        CALL FILL ROUTINE
219E: 6Ø        289           RTS

                291  * BUILD TRACK IMAGE USING SECTOR IMAGE

219F: AD D2 24  293  BLDTRK   LDA   CUREND
21A2: 85 3E     294           STA   A2          SET A2 TO SECTOR IMAGE END
21A4: AD D3 24  295           LDA   CUREND+1
21A7: 85 3F     296           STA   A2+1
21A9: A5 3C     297           LDA   A1          SET A4 TO CURRENT POSITION
21AB: 85 42     298           STA   A4          IN TRACK IMAGE
21AD: A5 3D     299           LDA   A1+1
21AF: 85 43     3ØØ           STA   A4+1
21B1: A9 ØF     3Ø1           LDA   #$ØF        SET COUNT TO 16
21B3: 8D CC 24  3Ø2           STA   COUNT
21B6: AØ ØØ     3Ø3           LDY   #$ØØ        CLEAR Y FOR MOVE ROUTINE
21B8: A9 23     3Ø4  MORE2    LDA   #>IMAGE
21BA: 85 3C     3Ø5           STA   A1          SET A1 TO SECTOR IMAGE START
21BC: A9 23     3Ø6           LDA   #>IMAGE
21BE: 85 3D     3Ø7           STA   A1+1
21CØ: 2Ø 2C FE  3Ø8           JSR   MOVE        MOVE SECTOR IMAGE TO TRACK IMAGE
21C3: CE CC 24  3Ø9           DEC   COUNT       DECREMENT COUNT
21C6: 1Ø FØ     31Ø           BPL   MORE2       LOOP UNTIL WE HAVE 16 SECTORS
21C8: 6Ø        311           RTS

                313  * FIX ADDRESS INFORMATION IN TRACK IMAGE

21C9: A9 ØØ     315  FIXTRK   LDA   #$ØØ
21CB: 8D D8 24  316           STA   SECTOR      START WITH SECTOR ZERO
21CE: AD D4 24  317           LDA   START
21D1: 85 3C     318           STA   A1          SET A1 TO TRACK IMAGE START
21D3: AD D5 24  319           LDA   START+1
21D6: 85 3D     32Ø           STA   A1+1
21D8: A9 ØØ     321           LDA   #$ØØ
21DA: 85 3E     322           STA   A2          SET A2 TO $8ØØØ
21DC: A9 8Ø     323           LDA   #$8Ø
21DE: 85 3F     324           STA   A2+1
21EØ: 2Ø 22 22  325  AGAIN    JSR   POSITION    POSITION TO ADDRESS INFO
```

```
21E3: 2Ø 4C 22  326          JSR   CMPADD     UPDATE ADDRESS INFO
21E6: EE D8 24  327          INC   SECTOR     NEXT SECTOR
21E9: AD D8 24  328          LDA   SECTOR
21EC: C9 1Ø     329          CMP   #$1Ø       DONE ALL 16 YET?
21EE: DØ FØ     33Ø          BNE   AGAIN      IF NOT DO ANOTHER
21FØ: 6Ø        331          RTS

                333 * SET POINTERS FOR VERIFY ROUTINE
                334 * ON EXIT: A1 > 1ST ADDRESS FIELD IN TRACK IMAGE
                335 *          A4 > 1ST ADDRESS FIELD IN LIVE DATA FROM DISK

21F1: A9 ØØ     337 SETUP    LDA   #$ØØ
21F3: 85 3C     338          STA   A1         SET A1 TO $4ØØØ
21F5: A9 4Ø     339          LDA   #$4Ø
21F7: 85 3D     34Ø          STA   A1+1
21F9: A9 ØØ     341          LDA   #$ØØ
21FB: 85 3E     342          STA   A2         SET A2 TO $6ØØØ
21FD: A9 6Ø     343          LDA   #$6Ø
21FF: 85 3F     344          STA   A2+1
22Ø1: 2Ø 22 22  345          JSR   POSITION   LOCATE 1ST ADDRESS FIELD
22Ø4: A5 3C     346          LDA   A1
22Ø6: 85 42     347          STA   A4         SET A4 TO POSITION FOUND
22Ø8: A5 3D     348          LDA   A1+1
22ØA: 85 43     349          STA   A4+1
22ØC: AD D4 24  35Ø          LDA   START
22ØF: 85 3C     351          STA   A1         SET A1 TO START OF TRACK IMAGE
2211: AD D5 24  352          LDA   START+1
2214: 85 3D     353          STA   A1+1
2216: A9 FE     354          LDA   #$FE
2218: 85 3E     355          STA   A2         SET A2 TO $7FFE
221A: A9 7F     356          LDA   #$7F
221C: 85 3F     357          STA   A2+1
221E: 2Ø 22 22  358          JSR   POSITION   LOCATE 1ST ADDRESS FIELD
2221: 6Ø        359          RTS

                361 * LOCATE AN ADDRESS FIELD

2222: AØ ØØ     363 POSITION LDY   #$ØØ       INITIALIZE OFFSET
2224: A2 Ø2     364 POS2     LDX   #$Ø2       INITIALIZE COUNT
2226: 2Ø BA FC  365 POS3     JSR   NXTA1      INCREMENT POINTER
2229: BØ ØD     366          BCS   DONE       IF PAST DATA THEN EXIT
222B: B1 3C     367          LDA   (A1),Y     GET A BYTE
222D: DD DD 24  368          CMP   TABLE,X    CHECK IT IN TABLE
223Ø: DØ F2     369          BNE   POS2       NOT THERE, THEN TRY AGAIN
2232: CA        37Ø          DEX   FOUND      ONE, COUNT IT
2233: 1Ø F1     371          BPL   POS3       GO UNTIL WE HAVE THREE
2235: 2Ø BA FC  372          JSR   NXTA1      POINT ONE PAST
2238: 6Ø        373 DONE     RTS

                375 * COMPAIR TWO AREAS OF MEMORY

2239: AØ ØØ     377 COMPARE  LDY   #$ØØ       INITIALIZE OFFSET
223B: B1 3C     378 LOOPC    LDA   (A1),Y     GET A BYTE (TRACK IMAGE)
223D: Ø9 8Ø     379          ORA   #$8Ø       TURN 7F'S TO FF'S
223F: D1 42     38Ø          CMP   (A4),Y     COMPAIR WITH DISK IMAGE
2241: DØ Ø7     381          BNE   MISMATCH   EXIT ON MISMATCH
2243: 2Ø B4 FC  382          JSR   NXTA4      INCREMENT BOTH POINTERS
2246: 9Ø F3     383          BCC   LOOPC      LOOP UNTIL DONE
2248: 18        384          CLC   INDICATE   SUCCESS
2249: 6Ø        385          RTS
224A: 38        386 MISMATCH SEC   INDICATE   ERROR
224B: 6Ø        387          RTS

                389 * COMPUTE ADDRESS FIELD INFORMATION AND STORE IN TRACK IMAGE
```

```
224C: AD Ø5 2Ø  391  CMPADD  LDA   VOLNUM      GET VOLUME NUMBER
224F: 2Ø 6B 22  392          JSR   COMPUTE     COMPUTE AND STORE IT
2252: AD D7 24  393          LDA   TRACK       GET CURRENT TRACK
2255: 2Ø 6B 22  394          JSR   COMPUTE     COMPUTE AND STORE IT
2258: AD D8 24  395          LDA   SECTOR      GET CURRENT SECTOR
225B: 2Ø 6B 22  396          JSR   COMPUTE
225E: AD Ø5 2Ø  397          LDA   VOLNUM
2261: 4D D7 24  398          EOR   TRACK
2264: 4D D8 24  399          EOR   SECTOR      GET CHECKSUM
2267: 2Ø 6B 22  4ØØ          JSR   COMPUTE
226A: 6Ø        4Ø1          RTS

                4Ø3  * NIBBLIZE A BYTE

226B: 48        4Ø5  COMPUTE PHA   SAVE        A-REGISTER
226C: 4A        4Ø6          LSR   A           ØABCDEFG  H
226D: Ø9 AA     4Ø7          ORA   #$AA        1A1C1E1G
226F: 91 3C     4Ø8          STA   (A1),Y      STORE IT
2271: 68        4Ø9          PLA   ABCDEFGH
2272: Ø9 AA     41Ø          ORA   #$AA        1B1D1F1H
2274: C8        411          INY
2275: 91 3C     412          STA   (A1),Y      STORE IT
2277: C8        413          INY
2278: 6Ø        414          RTS

                416  *  RECALIBRATE DISK ARM

2279: A9 3Ø     418  RECALC  LDA   #$3Ø        PRETENT TO BE ON TRACK 48
227B: 8D DA 24  419          STA   CURTRK
227E: A9 ØØ     42Ø          LDA   #$ØØ        SELECT TRACK ØØ
228Ø: 8D D9 24  421          STA   DESTRK
2283: 2Ø 96 22  422          JSR   ARMOVE      GO THERE
2286: AE D6 24  423          LDX   SLOT        GET SLOT NUMBER
2289: BD 8Ø CØ  424          LDA   DRVSMØ,X    TURN ALL PHASES OFF
228C: BD 82 CØ  425          LDA   DRVSM2,X
228F: BD 84 CØ  426          LDA   DRVSM4,X
2292: BD 86 CØ  427          LDA   DRVSM6,X
2295: 6Ø        428          RTS   RETURN      TO CALLER

                43Ø  *  ARM MOVE ROUTINE

2296: A9 ØØ     432  ARMOVE  LDA   #$ØØ
2298: 8D DC 24  433          STA   FLAG        INITIALIZE FLAG
229B: AD DA 24  434          LDA   CURTRK      GET CURRENT TRACK
229E: 38        435          SEC
229F: ED D9 24  436          SBC   DESTRK      SUBTRACT DESTINATION TRACK
22A2: FØ 36     437          BEQ   DONE2       IF EQUAL THEN EXIT
22A4: BØ Ø4     438          BCS   OK          POSITIVE RESULT? YES, GO ON
22A6: 49 FF     439          EOR   #$FF        MAKE RESULT POSITIVE
22A8: 69 Ø1     44Ø          ADC   #$Ø1
22AA: 8D DB 24  441  OK      STA   DELTA       SAVE RESULT
22AD: 2E DC 24  442          ROL   FLAG        SET IN/OUT FLAG
22BØ: 4E DA 24  443          LSR   CURTRK      ON ODD OR EVEN TRACK?
22B3: 2E DC 24  444          ROL   FLAG        PUT RESULT IN FLAG
22B6: ØE DC 24  445          ASL   FLAG        ADJUST FOR TABLE OFFSET
22B9: AC DC 24  446          LDY   FLAG        GET TABLE OFFSET
22BC: B9 EF 22  447  LOOP6   LDA   PTABLE,Y    GET PHASE TO TURN ON
22BF: 2Ø DB 22  448          JSR   PHASE
22C2: B9 FØ 22  449          LDA   PTABLE+1,Y GET NEXT PHASE TO TURN ON
22C5: 2Ø DB 22  45Ø          JSR   PHASE
22C8: 98        451          TYA
22C9: 49 Ø2     452          EOR   #$Ø2        ADJUST OFFSET
22CB: A8        453          TAY
22CC: CE DB 24  454          DEC   DELTA       DECREMENT NUMBER OF TRACKS TO MOVE
22CF: AD DB 24  455          LDA   DELTA
```

```
22D2: DØ E8     456          BNE   LOOP6      IF NOT DONE, DO ANOTHER
22D4: AD D9 24  457          LDA   DESTRK     UPDATE CURRENT TRACK WITH
22D7: 8D DA 24  458          STA   CURTRK     WHERE THE ARM IS NOW
22DA: 6Ø        459 DONE2    RTS   DONE,      RETURN TO CALLER

                461 *  TURN A PHASE ON, WAIT THEN TURN IT OFF

22DB: ØD D6 24  463 PHASE    ORA   SLOT       ADD SLOT TO PHASE
22DE: AA        464          TAX
22DF: BD 81 CØ  465          LDA   DRVSM1,X   TURN ON A PHASE
22E2: 2Ø E9 22  466          JSR   WAIT       WAIT FOR ARM TO SETTLE
22E5: BD 8Ø CØ  467          LDA   DRVSMØ,X   TURN OFF PHASE
22E8: 6Ø        468          RTS   RETURN     TO CALLER

                47Ø *  2Ø MILLISECOND DELAY ROUTINE

22E9: A9 56     472 WAIT     LDA   #$56       WAIT ABOUT 2Ø MILLISECONDS
22EB: 2Ø A8 FC  473          JSR   DELAY
22EE: 6Ø        474          RTS   RETURN     TO CALLER

                476 *  PHASE TABLE

22EF: Ø2 Ø4 Ø6  478 PTABLE   DFB   $Ø2,$Ø4,$Ø6,$ØØ
22F2: ØØ
22F3: Ø6 Ø4 Ø2  479          DFB   $Ø6,$Ø4,$Ø2,$ØØ
22F6: ØØ

                481 * CLEAR SCREEN AND DISPLAY MESSAGE

22F7: 2Ø 58 FC  483 SCREEN   JSR   HOME       CLEAR SCREEN
22FA: A9 EØ     484          LDA   #<MESSAGE
22FC: 85 ØØ     485          STA   PTR        POINT AT MESSAGE
22FE: A9 24     486          LDA   #>MESSAGE
23ØØ: 85 Ø1     487          STA   PTR+1
23Ø2: 2Ø 33 23  488          JSR   PRINT      PRINT IT
23Ø5: 6Ø        489          RTS

                491 * PRINT TRACK NUMBER

23Ø6: AD D7 24  493 PRTRK    LDA   TRACK      GET TRACK NUMBER
23Ø9: 2Ø DA FD  494          JSR   PRBYTE     PRINT IT
23ØC: 2Ø 1Ø FC  495          JSR   BS
23ØF: 2Ø 1Ø FC  496          JSR   BS         MOVE CURSOR BACK
2312: 6Ø        497          RTS

                499 * ERROR HANDLER

2313: C9 Ø1     5Ø1 ERRHDL   CMP   #$Ø1       IS IT ERROR #1
2315: DØ ØA     5Ø2          BNE   SECOND     NO, THEN ASSUME #2
2317: A9 F3     5Ø3          LDA   #<MESSAGE1
2319: 85 ØØ     5Ø4          STA   PTR        POINT AT MESSAGE 1
231B: A9 24     5Ø5          LDA   #>MESSAGE1
231D: 85 Ø1     5Ø6          STA   PTR+1
231F: DØ Ø8     5Ø7          BNE   PRINTIT    ALWAYS TAKEN
2321: A9 25     5Ø8 SECOND   LDA   #>MESSAGE2
2323: 85 ØØ     5Ø9          STA   PTR        POINT AT MESSAGE 2
2325: A9 25     51Ø          LDA   #>MESSAGE2
2327: 85 Ø1     511          STA   PTR+1
2329: 2Ø 33 23  512 PRINTIT  JSR   PRINT      PRINT IT
232C: AE D6 24  513          LDX   SLOT       GET SLOT
232F: BD 88 CØ  514          LDA   DRVOFF,X   TURN DRIVE OFF
2332: 6Ø        515          RTS   EXIT       PROGRAM

                517 * PRINT ROUTINE
```

```
2333: AØ ØØ      519  PRINT    LDY  #$ØØ       INITIALIZE OFFSET
2335: B1 ØØ      52Ø  CHAR     LDA  (PTR),Y    GET CHARACTER
2337: FØ Ø6      521           BEQ  TERMINATE  IF ZERO THEN EXIT
2339: 2Ø ED FD   522           JSR  COUT       PRINT CHARACTER
233C: C8         523           INY
233D: DØ F6      524           BNE  CHAR       DO ANOTHER
233F: 6Ø         525  TERMINATE RTS

                 527  *  DATA AREA

                 529  IMAGE    EQU  *

234Ø: D5 AA 96   531  HEADER1  DFB  $D5,$AA,$96

                 533  ADDRESS  EQU  *
2343: AA AA      534  VOL      DFB  $AA,$AA
2345: AA AA      535  TRK      DFB  $AA,$AA
2347: AA AA      536  SEC      DFB  $AA,$AA
2349: AA AA      537  CHK      DFB  $AA,$AA

234B: DE AA EB   539  TRAILER1 DFB  $DE,$AA,$EB

234E: 7F 7F 7F   541  GAP2     DFB  $7F,$7F,$7F
2351: 7F 7F 7F   542           DFB  $7F,$7F,$7F

2354: D5 AA AD   544  HEADER2  DFB  $D5,$AA,$AD

2357: ØØ ØØ ØØ   546  DATA     DS   $56
235A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2362: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
236A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2372: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
237A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2382: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
238A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2392: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
239A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23A2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23AA: ØØ ØØ ØØ
23AD: ØØ ØØ ØØ   547           DS   $1ØØ
23BØ: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23B8: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23CØ: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23C8: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23DØ: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23D8: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23EØ: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23E8: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23FØ: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
23F8: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
24ØØ: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
24Ø8: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
241Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2418: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
242Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2428: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
243Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2438: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
244Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2448: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
245Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2458: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
246Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2468: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
247Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
```

```
2478: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
248Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2488: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
249Ø: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2498: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
24AØ: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
24A8: ØØ ØØ ØØ ØØ ØØ
24AD: ØØ        548              DS    $Ø1
                549    DATAEND   EQU   *-1
                55Ø    DATALTH   EQU   DATAEND-DATA

24AE: DE AA EB  552    TRAILER2  DFB   $DE,$AA,$EB

24B1: 7F 7F 7F  554    GAP3      DFB   $7F,$7F,$7F
24B4: 7F 7F 7F  555              DFB   $7F,$7F,$7F
24B7: 7F 7F 7F  556              DFB   $7F,$7F,$7F
24BA: 7F 7F 7F  557              DFB   $7F,$7F,$7F
24BD: 7F 7F 7F  558              DFB   $7F,$7F,$7F
24CØ: 7F 7F 7F  559              DFB   $7F,$7F,$7F
24C3: 7F 7F 7F  56Ø              DFB   $7F,$7F,$7F
24C6: 7F 7F 7F  561              DFB   $7F,$7F,$7F
24C9: 7F 7F 7F  562              DFB   $7F,$7F,$7F
                563    END       EQU   *-1
                564    LEN       EQU   END-IMAGE+1

24CC: ØØ ØØ     566    COUNT     DFB   $ØØ,$ØØ
24CE: ØØ        567    RETRYCNT  DFB   $ØØ
24CF: ØØ        568    BYTE      DFB   $ØØ
24DØ: ØØ ØØ     569    LENGTH    DFB   $ØØ,$ØØ
24D2: ØØ ØØ     57Ø    CUREND    DFB   $ØØ,$ØØ
24D4: ØØ ØØ     571    START     DFB   $ØØ,$ØØ
24D6: 6Ø        572    SLOT      DFB   $6Ø
24D7: ØØ        573    TRACK     DFB   $ØØ
24D8: ØØ        574    SECTOR    DFB   $ØØ
24D9: ØØ        575    DESTRK    DFB   $ØØ        DESTINATION TRACK
24DA: ØØ        576    CURTRK    DFB   $ØØ        CURRENT TRACK
24DB: ØØ        577    DELTA     DFB   $ØØ        NUMBER OF TRACKS TO MOVE
24DC: ØØ        578    FLAG      DFB   $ØØ        DIRECTION & ODD/EVEN FLAGS
24DD: 96 AA D5  579    TABLE     DFB   $96,$AA,$D5

24EØ: C6 CF D2  581    MESSAGE   ASC   "FORMATTING TRACK  "
24E3: CD C1 D4 D4 C9 CE C7 AØ
24EB: D4 D2 C1 C3 CB AØ AØ
24F2: ØØ        582              DFB   $ØØ
24F3: 8D        583    MESSAGE1  DFB   $8D
24F4: D7 D2 C9  584              ASC   "WRITE PROTECT ERROR"
24F7: D4 C5 AØ DØ D2 CF D4 C5
24FF: C3 D4 AØ C5 D2 D2 CF D2
25Ø7: 87 ØØ     585              DFB   $87,$ØØ
25Ø9: 8D        586    MESSAGE2  DFB   $8D
25ØA: D5 CE C1  587              ASC   "UNABLE TO FORMAT"
25ØD: C2 CC C5 AØ D4 CF AØ C6
2515: CF D2 CD C1 D4
251A: 87 ØØ     588              DFB   $87,$ØØ

--End assembly, 13Ø8 bytes, Errors: Ø
```

ZAP—DISK UPDATE UTILITY

The next step up the ladder from DUMP and FORMAT is accessing data on the diskette at the block level. The ZAP program allows its user to specify a block number to be read into memory. The user can then make changes to the image of the block in memory, and subsequently use ZAP to write the modified image back over the block on disk. ZAP is particularly useful when it is necessary to patch up a damaged directory. Its use in this regard will be covered in more detail when FIB is explained.

To use ZAP, store the number of the block you wish to access at $2007 and $2008. Store the least significant byte of the number in $2007 and the most significant byte in $2008. For example, the key block of the Volume Directory may be read by entering 2007:02 00. $2009 should be initialized with either $80 to indicate that a sector is to be read into memory, or $81 to ask that memory be written out to the block on the disk. You may also specify the disk drive to be used (slot 6, drive 1 is assumed) by storing a hex value of $s0 at $2004, where "s" is the slot to be used. If you wish to access drive 2 for a given slot, turn on the most significant bit in $2004 (e.g. slot 6, drive 2 would be 2004:E0). An example to illustrate the use of ZAP follows.

```
CALL -151 (Get into the monitor)
BLOAD ZAP (Load the ZAP program)
```

(Now insert the diskette to be ZAPped)

```
2007:02 00 80 N 2000G (Store a 2 (key block of the Volume
                      directory) in $2007/8 and $80 (read
                      block) at $2009. N ends the store
                      command and 2000G runs ZAP.)
```

The output might look like this...

```
1000- 00 00 03 00 FA 55 53 45
1008- 52 53 2E 44 49 53 4B 00
1010- 00 00 00 00 00 00 00 00
1018- 00 00 00 00 00 00 00 00
  ...etc...
```

In the example above; if the byte at offset 6 (the second character of the volume name, "USERS.DISK") is to be changed to "O", the following would be entered.

```
1006:4F (Change +$06 to $4F ("O"))
2009:81 N 2000G (Change ZAP to write mode and do it)
```

Note that `ZAP` will remember the previous values in `$2004` through `$2009`. If something is wrong with the block to be read or written (an I/O error, perhaps), `ZAP` will print an error message of the form:

```
RC = 2B
```

A return code of `$2B`, in this case, means that the diskette was write protected and a write operation was attempted. Other error codes are `$27` (I/O error), and `$28` (no device connected). Refer to the documentation on `READ_BLOCK` and `WRITE_BLOCK` in Chapter 6 for more information on these errors.

```
                 1             ORG   $2000

                 3    *******************************************************
                 4    *                                                     *
                 5    * ZAP:  THIS PROGRAM WILL ALLOW ITS USER TO READ/WRITE *
                 6    *       INDIVIDUAL BLOCKS FROM/TO THE DISKETTE        *
                 7    *                                                     *
                 8    * INPUT: $2004   = UNIT NUMBER (DSSS0000)             *
                 9    *                  DEFAULTS TO SLOT 6, DRIVE 1 ($60)  *
                 10   *                  (SLOT 6, DRIVE 2 IS $E0)           *
                 11   *        $2005/6 = ADDRESS OF AREA IN MEMORY TO BE    *
                 12   *                  READ/WRITTEN.                      *
                 13   *                  DEFAULTS TO $1000                  *
                 14   *        $2007/8 = BLOCK NUMBER TO BE READ/WRITTEN    *
                 15   *                  DEFAULTS TO $0000                  *
                 16   *        $2009   = OPERATION TO BE PERFORMED:         *
                 17   *                     $80 = READ BLOCK                *
                 18   *                     $81 = WRITE BLOCK               *
                 19   *                  DEFAULTS TO READ BLOCK.            *
                 20   *                                                     *
                 21   * ENTRY POINT: $2000                                  *
                 22   *                                                     *
                 23   * PROGRAMMER: DON D WORTH - 2/25/84                   *
                 24   *                                                     *
                 25   *******************************************************

                 27   *       FIXED LOCATIONS WE NEED

                 29   A1L      EQU   $3C        MONITOR POINTERS
                 30   A1H      EQU   $3D
                 31   A2L      EQU   $3E
                 32   A2H      EQU   $3F
                 33   MLI      EQU   $BF00      MACHINE LANGUAGE INTERFACE
                 34   COUT     EQU   $FDED      MONITOR PRINT VECTOR
                 35   PRBYTE   EQU   $FDDA      MONITOR PRINT HEX BYTE
                 36   XAM      EQU   $FDB3      MONITOR HEX DUMP SUBRTN

                 38   *       ENTRY POINT, JUMP AROUND PARMS

2000: 4C 0A 20   40   ZAP      JMP   START      BR AROUND DATA

                 42   *       MLI READ/WRITE BLOCK PARAMETER LIST

2003: 03         44   RWBLP    DFB   $03        PARM COUNT = 3
2004: 60         45            DFB   $60        UNIT NUMBER
2005: 00 10      46   BUFF     DW    $1000      BUFFER ADDRESS
2007: 00 00      47            DW    $0000      BLOCK NUMBER
2009: 80         48   OPER     DFB   $80        OPERATION TO BE PERFORMED

                 50   *       START OF CODE, CALL MLI
```

```
200A: AD 09 20  52   START    LDA   OPER        PASS OPERATION CODE
200D: 8D 13 20  53            STA   OP
2010: 20 00 BF  54            JSR   MLI         CALL MLI
2013: 00        55   OP       DFB   $00
2014: 03 20     56            DW    RWBLP
2016: 90 19     57            BCC   EXIT        ALL WENT WELL???

                59   *       IF ERROR OCCURS, PRINT MESSAGE

2018: 48        61            PHA               ; SAVE ERROR CODE
2019: A9 87     62            LDA   #$87        BEEP THE SPEAKER
201B: 20 ED FD  63            JSR   COUT
201E: A9 D2     64            LDA   #"R"        PRINT THE "RC="
2020: 20 ED FD  65            JSR   COUT
2023: A9 C3     66            LDA   #"C"
2025: 20 ED FD  67            JSR   COUT
2028: A9 BD     68            LDA   #"="
202A: 20 ED FD  69            JSR   COUT
202D: 68        70            PLA
202E: 4C DA FD  71            JMP   PRBYTE      PRINT THE HEX VALUE

                73   *       WHEN FINISHED, DUMP SOME OF BLOCK IN HEX

2031: 18        75   EXIT     CLC
2032: AD 05 20  76            LDA   BUFF        DUMP $2000-$20B7
2035: 85 3C     77            STA   A1L
2037: 69 AF     78            ADC   #$AF
2039: 85 3E     79            STA   A2L
203B: AD 06 20  80            LDA   BUFF+1
203E: 85 3D     81            STA   A1H
2040: 69 00     82            ADC   #0
2042: 85 3F     83            STA   A2H
2044: 4C B3 FD  84            JMP   XAM         EXIT VIA HEX DISPLAY

--End assembly, 71 bytes, Errors: 0
```

MAP—MAP FREESPACE ON A VOLUME

The MAP program is written in BASIC and calls a tiny assembly language subroutine to read blocks from a ProDOS volume. It first reads the Volume Directory key block to determine the length and location of the Volume Bit Map. It then reads the bit map and prints a map of the volume's freespace on the screen.

To run MAP against a disk volume, first LOAD the program into BASIC, place the disk to be MAPped in slot 6, drive 1, and then RUN the program. The output from such a run might look like this:

```
FREESPACE MAP FOR VOLUME /USERS.DISK/
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU.......
........................................
........................................
U = USED BLOCK . = FREE BLOCK
BLOCKS USED: 193 BLOCKS FREE: 87
```

The MAP program first reads a short machine language program from data statements and pokes it into memory at $3ØØ. The machine language program is as follows.

```
Ø3ØØ:    PHA            ; SAVE REGISTERS UPON ENTRY
Ø3Ø1:    TYA
Ø3Ø2:    PHA
Ø3Ø3:    TXA
Ø3Ø4:    PHA
Ø3Ø5:    JSR $BFØØ      CALL MLI
Ø3Ø8:    DFB $8Ø        READ_BLOCK CALL
Ø3Ø9:    DW $315        PARAMETERS AT $315
Ø3ØB:    STA $314       SAVE RETURN CODE
Ø3ØE:    PLA            ; RESTORE REGISTERS
Ø3ØF:    TAX
Ø31Ø:    PLA
Ø311:    TAY
Ø312:    PLA
Ø313:    RTS            ; AND RETURN TO BASIC PROGRAM
Ø314:    DFB $ØØ        RETURN CODE SAVED HERE
Ø315:    DFB $Ø3        3 PARAMETERS
Ø316:    DFB $6Ø        SLOT 6, DRIVE 1
Ø317:    DW $4ØØØ       BLOCK BUFFER IS AT $4ØØØ
Ø319:    DW Ø           BLOCK NUMBER FILLED IN BY BASIC PGM
```

MAP then calls the subroutine (see lines 1000-1020) to read the Volume Directory key block (BN = 2). It obtains the length of the volume name from +4 (eliminating the $FØ entry type), and peeks the volume name and prints it on the screen from +5 in the buffer. If the total number of blocks on the

volume (+41/42) is not 280, then the message "`NOT A PRODOS DISKETTE`" is printed. Otherwise, the first block of the Volume Bit Map is read. A loop is then entered (lines 365-440) Where each binary bit which is one in the bit map is counted and printed as a "`.`" (free) and each that is zero is counted and printed as a "`U`" (in use). The totals for used and free blocks are then printed and the program exits. If an error occurs, it is printed in decimal and the program aborts execution. Possible errors are 39 (I/O error) and 40 (no device connected).

`MAP` will not currently work for volumes with more or less than 280 blocks but this can be easily changed by the reader.

```
1Ø   REM
2Ø   REM  THIS PROGRAM PRINTS A MAP OF
3Ø   REM  A PRODOS DISKETTE VOLUME.
4Ø   REM
5Ø   REM  PROGRAMMER: DON D WORTH 2/22/84
6Ø   REM
7Ø   DATA   72,152,72,138,72,32,Ø,191,128,21,3,141,2Ø,3,1Ø4,17Ø,1Ø4,168
75   DATA  1Ø4,96,Ø,3,96
8Ø   REM
9Ø   REM  POKE BLOCK READ SUBROUTINE INTO MEMORY
95   REM
1ØØ SB = 768: REM  SB=ADDR OF SUBROUTINE
1Ø5 BF = 16384: REM  BUFFER IS AT $4ØØØ
11Ø  FOR I = SB TO SB + 22
12Ø  READ X: POKE I,X
13Ø  NEXT I
14Ø  POKE I,Ø: POKE I + 1,BF / 256
15Ø BN = SB + 25:RC = SB + 2Ø
16Ø  REM
17Ø  REM   READ THE VOLUME DIRECTORY KEY BLOCK TO FIND THE BIT MAP
19Ø  REM
2ØØ  POKE BN,2
21Ø  GOSUB 1ØØØ
23Ø  REM
24Ø  REM  PRINT THE VOLUME NAME
25Ø  REM
26Ø L =  PEEK (BF + 4) - 24Ø
265  HOME : PRINT "FREESPACE MAP FOR VOLUME /";
27Ø  FOR I = 1 TO L
28Ø  PRINT  CHR$ ( PEEK (BF + I + 4));
29Ø  NEXT I
3ØØ  PRINT "/": PRINT
31Ø  REM
32Ø  REM  LOCATE AND READ BIT MAP BLOCK
33Ø  REM
34Ø  IF  PEEK (BF + 41) +  PEEK (BF + 42) * 256 <  > 28Ø THEN  PRINT  CHR$
     (7);"NOT A PRODOS DISKETTE": END
35Ø  POKE BN, PEEK (BF + 39): POKE BN + 1, PEEK (BF + 4Ø)
36Ø  GOSUB 1ØØØ
362  REM
363  REM  PRINT BIT MAP
364  REM
365 U = Ø:F = Ø
37Ø  FOR B = Ø TO 34
38Ø X =  PEEK (B + BF)
39Ø  FOR I = 1 TO 8
4ØØ  IF X >  = 128 THEN X = X - 128: PRINT ".";:F = F + 1: GOTO 42Ø
41Ø  PRINT "U";:U = U + 1
```

```
420 X = X * 2
430  NEXT I
440  NEXT B
442  REM
443  REM  FINISH UP
444  REM
450  PRINT : PRINT : PRINT "U=USED BLOCK        .=FREE BLOCK"
455  PRINT : PRINT "BLOCKS USED: ";U;"    BLOCKS FREE: ";F
460  END
1000  REM
1001  REM  READ A BLOCK FROM DISK
1002  REM
1003  CALL SB
1010  IF  PEEK (RC) = 0 THEN  RETURN
1020  PRINT "I/O ERROR = "; PEEK (RC); CHR$ (7): END
```

FIB—FIND INDEX BLOCK UTILITY

From time to time one of your diskettes will develop an I/O error smack in the middle of a directory. When this occurs, any attempt to use the files described by that directory will result in an `I/O ERROR` message from ProDOS. Generally, when this happens, the data stored in the files on the diskette is still intact, only the pointers to the files are gone. If the data absolutely must be recovered, a knowledgeable Apple user can reconstruct the directory from scratch. Doing this involves finding the index blocks for each file, and then using `ZAP` to patch a directory entry into the Volume Directory for each file which is found. `FIB` is a utility which will scan a disk volume for index blocks. Although it may flag some blocks which are not index blocks as being such it will never miss a valid index block. Therefore, after running `FIB` the programmer must use `ZAP` to examine each block printed by `FIB` to see if it is really an index block. Additionally, `FIB` will find every index block image on the volume, even some which were for files which have since been deleted. Since it is difficult to determine which files are valid and which are old deleted files, it is usually necessary to restore all the files and copy them to another diskette, and later delete the duplicate or unwanted ones.

To run `FIB`, simply load the program and start execution at `$2ØØØ`. `FIB` will print the block number of each block it finds which bears a resemblance to an index block. For example

```
CALL -151 (Get into the monitor)
BLOAD FIB (Load the FIB program)
```

(Now insert the disk to be scanned into Slot 6 Drive 1)

```
2ØØØG (Run the FIB program on this diskette)
```

The output might look like this...

```
BLK=ØØØ8     BLK=ØØ99
BLK=ØØ27     BLK=ØØAF
BLK=ØØ28     BLK=ØØB1
BLK=ØØ3C     BLK=ØØB4
BLK=ØØ6F     BLK=ØØB7
BLK=ØØ97
```

Here 11 possible files were found. Of course, if some of the lost files were seedlings, they will not be represented here (seedlings are very difficult to locate once their directory entry is gone). And if some files were tree files, then three or more of the above block numbers could refer to index blocks for a single file. Also, if only one of several directories for a volume is damaged,

some of the block numbers given may refer to files whose directory entries are still intact. If, after running `FIB`, you get an error message (`RC = xx`, see `ZAP` errors), you may need to reformat the offending track. Divide the block number by eight to determine which track has the error. An alternative is to use `ZAP` to copy all blocks without errors to another formatted disk and write zeroes on the blocks corresponding to I/O errors. In this way you can preserve undamaged blocks which are on the same track with damaged ones.

In the example above, `ZAP` should now be used to read block 8. At +$00 and +$100 are the LSB and MSB of the block number for the first data block of the file (assuming this is not the master index block for a tree file). This block can be read and examined to try to identify the file and its type. Usually a BASIC program can be identified (even though it is stored in tokenized form) from the text strings contained in the `PRINT` statements. An ASCII conversion chart (see page 16 in the *Apple II Reference Manual for IIe Only*) can be used to decode these character strings. Straight `TXT` type files will also contain ASCII text, with each line separated from the others with `$ØDs` (carriage returns). `BIN` type files are the hardest to identify and recover since their original address and length attributes were lost along with the directory entry. If you cannot identify a file, assume it is `BAS` (Applesoft BASIC). If this assumption turns out to be incorrect, you can always go back and `ZAP` the file type in the directory to try something else. Given below is an example `ZAP` to the Volume Directory to create an entry for the file whose index block is `BLK = ØØØ8`. This `ZAP` assumes that the Volume Directory itself was lost and that you are starting the entire volume from scratch. Do not perform this patch to a diskette which is only partially damaged as you will wipe out the remainder of the valid directory entries in the process.

```
CALL -151
BLOAD ZAP
```

(insert disk to be `ZAP`ped)

```
1ØØØ:ØØ N 1ØØ1<1ØØØ.11FEM     (Zero entire block of memory)
1ØØØ:ØØ ØØ Ø3 ØØ F5 46 49 58  (Store a dummy Volume
                              Directory
1ØØ8:55 5Ø                    header for volume /FIXUP)
1Ø2Ø:ØØ ØØ C3 27 ØD ØØ ØØ Ø6
1Ø28:ØØ 18 Ø1
1Ø2B:24 46 49 4C 45           (Make sapling entry for
                              "FILE")
1Ø3B:FC                       (file is type BAS)
1Ø3C:Ø8 ØØ                    (key block is 8)
1Ø4Ø:ØØ xx ØØ                 (EOF mark, see below)
```

```
1049:E3                          (full access "unlocked")
104A:01 08                       (AUX_TYPE = $801 for BAS
                                 file)
1050:02 00                       (header pointer)
2007:02 00 81 N 2000G            (write new block image out as
                                 first Volume Directory block)
```

The "xx" above should be set to the number of non-zero block numbers found in the index block as a first cut at the end of file mark. If garbage is loaded at the end of the program, try a smaller number. You may be able to deduce the true EOF by examining the program image on disk. Remember that AUX_TYPE will be different for different file types. See Chapter 4 for more information.

As soon as the entry is created using the above procedure, the file should be immediately copied to another diskette. Do not attempt to use the file in place because the Volume Bit Map has not been updated and several other fields in the directory entry have been omitted. Also, you do not want to risk damaging other "lost" files on the disk. Repeat the above process for each index block found by FIB. As each file is recovered, it may be RENAMEd to its original name on the new diskette. Once all the files have been copied to another disk, and successfully tested, the damaged disk may be re-initialized.

```
                1           ORG   $2000

                3     *********************************************************
                4     *                                                       *
                5     * FIB:  THIS PROGRAM SCANS AN ENTIRE VOLUME, SEARCHING  *
                6     *       FOR WHAT APPEAR TO BE INDEX BLOCKS AND PRINTS   *
                7     *       THE BLOCK NUMBER OF EACH ONE IT FINDS. FIB WILL *
                8     *       WORK FOR ANY SIZED VOLUME. SLOT 6, DRIVE 1 IS   *
                9     *       ASSUMED.                                        *
                10    *                                                       *
                11    * INPUT: NONE                                           *
                12    *                                                       *
                13    * ENTRY POINT: $2000                                    *
                14    *                                                       *
                15    * PROGRAMMER: DON D WORTH - 2/25/84                     *
                16    *                                                       *
                17    *********************************************************

                19    *       FIXED LOCATIONS WE NEED

                21    PTR      EQU   $48         WORK POINTER
                22    BUFFER   EQU   $1000       BLOCK BUFFER
                23    MLI      EQU   $BF00       MACHINE LANGUAGE INTERFACE
                24    COUT     EQU   $FDED       MONITOR PRINT VECTOR
                25    PRBYTE   EQU   $FDDA       MONITOR PRINT HEX BYTE
                26    XAM      EQU   $FDB3       MONITOR HEX DUMP SUBRTN

                28    *       OFFSETS INTO VOL DIR HEADER

                30    TOTBLK   EQU   $1029
                31    BITMAP   EQU   $1027

                33    *       ENTRY POINT, READ VOLUME DIRECTORY HEADER

2000: A9 02     35    FIB      LDA   #2          BLOCK = 2
```

```
2ØØ2: 8D E9 2Ø  36             STA   BLOCK
2ØØ5: A9 ØØ     37             LDA   #Ø
2ØØ7: 8D EA 2Ø  38             STA   BLOCK+1
2ØØA: 2Ø 97 2Ø  39             JSR   READ        READ VOL DIR BLOCK
2ØØD: BØ 35     4Ø             BCS   EXIT        IF ERROR, GIVE UP RIGHT NOW
2ØØF: AD 29 1Ø  41             LDA   TOTBLK
2Ø12: 8D ED 2Ø  42             STA   LAST
2Ø15: AD 2A 1Ø  43             LDA   TOTBLK+1
2Ø18: 8D EE 2Ø  44             STA   LAST+1
2Ø1B: 4A        45             LSR   A           COMPUTE TOTBLK/4Ø96
2Ø1C: 4A        46             LSR   A
2Ø1D: 4A        47             LSR   A
2Ø1E: 4A        48             LSR   A           FOR NUM BIT MAP BLOCKS
2Ø1F: 38        49             SEC               ; AND ADD ONE TO THAT
2Ø2Ø: 6D 27 1Ø  5Ø             ADC   BITMAP
2Ø23: 8D E9 2Ø  51             STA   BLOCK
2Ø26: 8D EB 2Ø  52             STA   FIRST
2Ø29: A9 ØØ     53             LDA   #Ø
2Ø2B: 6D 28 1Ø  54             ADC   BITMAP+1
2Ø2E: 8D EA 2Ø  55             STA   BLOCK+1     POINT PAST THE BITMAP
2Ø31: 8D EC 2Ø  56             STA   FIRST+1

                58    *       SEE IF WE ARE AT END OF VOLUME

2Ø34: AD EA 2Ø  6Ø    NEWBLK   LDA   BLOCK+1     WHEN WE REACH LAST BLOCK +1
2Ø37: CD EE 2Ø  61             CMP   LAST+1
2Ø3A: DØ 13     62             BNE   READIT
2Ø3C: AD E9 2Ø  63             LDA   BLOCK
2Ø3F: CD ED 2Ø  64             CMP   LAST
2Ø42: DØ ØB     65             BNE   READIT
2Ø44: 6Ø        66    EXIT     RTS   EXIT        TO SYSTEM
2Ø45: EE E9 2Ø  67    NXTBLK   INC   BLOCK       INCREMENT BLOCK COUNT
2Ø48: DØ EA     68             BNE   NEWBLK
2Ø4A: EE EA 2Ø  69             INC   BLOCK+1
2Ø4D: DØ E5     7Ø             BNE   NEWBLK      AND CONTINUE LOOP

                72    *       READ BLOCK AND CHECK FOR VALIDITY AS AN INDEX BLOCK

2Ø4F: 2Ø 97 2Ø  74    READIT   JSR   READ        READ THIS BLOCK
2Ø52: BØ F1     75             BCS   NXTBLK      ERROR?

2Ø54: AØ ØØ     77             LDY   #Ø
2Ø56: B9 ØØ 1Ø  78    CHKNG1   LDA   BUFFER,Y    MAKE SURE ITS NOT ALL ZERO
2Ø59: 19 ØØ 11  79             ORA   BUFFER+$1ØØ,Y
2Ø5C: DØ Ø5     8Ø             BNE   CHKNG2
2Ø5E: C8        81             INY
2Ø5F: DØ F5     82             BNE   CHKNG1
2Ø61: FØ E2     83             BEQ   NXTBLK      IF SO, SKIP IT

2Ø63: AØ ØØ     85    CHKNG2   LDY   #Ø
2Ø65: B9 ØØ 1Ø  86    CHKING   LDA   BUFFER,Y    ALLOW ZERO ENTRIES
2Ø68: DØ Ø5     87             BNE   COMPR
2Ø6A: B9 ØØ 11  88             LDA   BUFFER+$1ØØ,Y
2Ø6D: FØ ØE     89             BEQ   BLKOK
2Ø6F: A2 ØØ     9Ø    COMPR    LDX   #Ø          CHECK AGAINST FIRST
2Ø71: 2Ø 86 2Ø  91             JSR   CMP         DO 16 BIT COMPARE
2Ø74: 9Ø CF     92             BCC   NXTBLK      TOO SMALL FOR BLOCK NUMBER
2Ø76: A2 Ø2     93             LDX   #2          CHECK AGAINST LAST
2Ø78: 2Ø 86 2Ø  94             JSR   CMP
2Ø7B: BØ C8     95             BCS   NXTBLK      TOO LARGE FOR BLOCK NUMBER
2Ø7D: C8        96    BLKOK    INY
2Ø7E: DØ E5     97             BNE   CHKING

2Ø8Ø: 2Ø BE 2Ø  99             JSR   PBLOCK      FOUND A CANDIDATE, PRINT BLOCK NO.
2Ø83: 4C 45 2Ø  1ØØ            JMP   NXTBLK      THEN CONTINUE
```

```
                  1Ø2  *        CMP: 16 BIT COMPARE

2Ø86: B9 ØØ 11   1Ø4  CMP      LDA   BUFFER+$1ØØ,Y CHECK MSB
2Ø89: DD EC 2Ø   1Ø5           CMP   FIRST+1,X
2Ø8C: 9Ø Ø8      1Ø6           BCC   RTS         ITS SMALLER
2Ø8E: DØ Ø6      1Ø7           BNE   RTS         ITS BIGGER
2Ø9Ø: B9 ØØ 1Ø   1Ø8           LDA   BUFFER,Y    CHECK LSB
2Ø93: DD EB 2Ø   1Ø9           CMP   FIRST,X
2Ø96: 6Ø         11Ø  RTS      RTS

                  112  *        READ A BLOCK FROM DISK TO $1ØØØ

2Ø97: 2Ø ØØ BF   114  READ     JSR   MLI         CALL MLI
2Ø9A: 8Ø         115           DFB   $8Ø         READ CALL
2Ø9B: E5 2Ø      116           DW    RWBLP
2Ø9D: BØ Ø1      117           BCS   ERROR       ERROR?
2Ø9F: 6Ø         118           RTS

                  12Ø  *        IF ERROR OCCURS, PRINT MESSAGE

2ØAØ: 48         122  ERROR    PHA               ; SAVE ERROR CODE
2ØA1: A9 87      123           LDA   #$87        BEEP THE SPEAKER
2ØA3: 2Ø ED FD   124           JSR   COUT
2ØA6: A9 D2      125           LDA   #"R"        PRINT THE "RC="
2ØA8: 2Ø ED FD   126           JSR   COUT
2ØAB: A9 C3      127           LDA   #"C"
2ØAD: 2Ø ED FD   128           JSR   COUT
2ØBØ: A9 BD      129           LDA   #"="
2ØB2: 2Ø ED FD   13Ø           JSR   COUT
2ØB5: 68         131           PLA
2ØB6: 2Ø DA FD   132           JSR   PRBYTE      PRINT THE HEX VALUE
2ØB9: A9 AØ      133           LDA   #$AØ        PRINT A BLANK
2ØBB: 2Ø ED FD   134           JSR   COUT        AND FALL THRU TO PRINT BLOCK

                  136  *        PRINT CURRENT BLOCK NUMBER

2ØBE: A9 C2      138  PBLOCK   LDA   #"B"        PRINT "BLK="
2ØCØ: 2Ø ED FD   139           JSR   COUT
2ØC3: A9 CC      14Ø           LDA   #"L"
2ØC5: 2Ø ED FD   141           JSR   COUT
2ØC8: A9 CB      142           LDA   #"K"
2ØCA: 2Ø ED FD   143           JSR   COUT
2ØCD: A9 BD      144           LDA   #"="
2ØCF: 2Ø ED FD   145           JSR   COUT
2ØD2: AD EA 2Ø   146           LDA   BLOCK+1     PRINT BLOCK NUMBER IN HEX
2ØD5: 2Ø DA FD   147           JSR   PRBYTE      PRINT MSB
2ØD8: AD E9 2Ø   148           LDA   BLOCK
2ØDB: 2Ø DA FD   149           JSR   PRBYTE      AND LSB
2ØDE: A9 8D      15Ø           LDA   #$8D
2ØEØ: 2Ø ED FD   151           JSR   COUT        NEW LINE
2ØE3: 38         152           SEC
2ØE4: 6Ø         153           RTS

                  155  *        MLI READ/WRITE BLOCK PARAMETER LIST

2ØE5: Ø3         157  RWBLP    DFB   $Ø3         PARM COUNT = 3
2ØE6: 6Ø         158  UNIT     DFB   $6Ø         UNIT NUMBER
2ØE7: ØØ 1Ø      159  BUFF     DW    BUFFER      BUFFER ADDRESS
2ØE9: ØØ ØØ      16Ø  BLOCK    DW    $ØØØØ       BLOCK NUMBER

2ØEB: ØØ ØØ      162  FIRST    DW    $ØØØØ       FIRST BLOCK AFTER BIT MAP
2ØED: ØØ ØØ      163  LAST     DW    $ØØØØ       LAST BLOCK ON DISK +1

--End assembly, 239 bytes, Errors: Ø
```

TYPE—TYPE COMMAND

The TYPE program is an example of how to add commands to the ProDOS BASIC Interpreter. TYPE may be installed as a command by BRUNning TYPE or using the "-" smart RUN command. Once installed, the user may enter:

TYPE filename[,Sslot][,Ddrive]

The BI will not recognize "TYPE" as one of its commands and will pass control to the installed external command handler. The handler will locate and open the file, read its contents and print them on the screen or output device. The user may suspend the listing with any keypress and resume it with any other. A control-C will abort the listing.

TYPE's operation begins when it is BRUN. Its first task is to allocate a page of memory between the BI and its buffers. It will copy the resident part of its program into this page. TYPE next stores the address of the newly allocated page in the BI's EXTERNCMD vector in the BI Global Page. Each time the BI sees a command it doesn't recognize, it will call the address in the EXTERNCMD vector before treating it as an invalid command. The transient portion of TYPE finishes up by copying and relocating the fixed addresses in the resident portion up to its new home in the newly allocated BI buffer. The transient portion then exits to ProDOS.

When an unknown command line is encountered, control passes to the resident code at TYPENT. TYPENT compares the command to the string "TYPE", and if there is a match, it claims the command and returns to the BI to allow it to parse the filename and any other keywords given. If no SYNTAX ERROR occurs, control returns from the BI at the label TYPBAK. (If TYPENT does not recognize the command, it passes control on to the original contents of EXTERNCMD, in case there are other external command handlers installed.) When control returns to TYPBAK, the MLI is called to open the file, using the BI's General Purpose buffer at HIMEM for an I/O buffer. The file is then read, 256 bytes at a time using $2ØØ for a data buffer, and its contents are copied to the COUT screen output vector. At End of File, TYPENT exits to the BI through the MLI CLOSE function call.

TYPE may be used as a model for small command handlers. It is written in such a way that it may coexist with numerous other external command handlers by preserving the original value it finds in the EXTERNCMD vector. Suggestions for additional external commands might include a file COPY command or a file hex/ASCII DUMP command. Note that if the installed, resident portion

is longer than 256 bytes, the relocation code will have to be rewritten and will be a bit more complex.

```
                   1     ***************************************************
                   2     *                                                 *
                   3     * TYPE: WHEN BRUN, THIS PROGRAM INSTALLS AN EXTERNAL *
                   4     *       PRODOS BASIC INTERPRETER COMMAND BETWEEN THE *
                   5     *       BI AND ITS BUFFERS. THE NEW COMMAND IS      *
                   6     *       INVOKED AS FOLLOWS:                         *
                   7     *                                                 *
                   8     *       TYPE <PATHNAME>[,S#][,D#]                   *
                   9     *                                                 *
                   1Ø    *       THE TYPE COMMAND COPYS THE CONTENTS OF THE  *
                   11    *       INDICATED FILE TO THE SCREEN.               *
                   12    *                                                 *
                   13    *       THE RESIDENT PORTION OF THE TYPE COMMAND    *
                   14    *       REQUIRES ONLY 256 BYTES OF RAM.             *
                   15    *                                                 *
                   16    * PROGRAMMER: DON D WORTH - 2/21/84               *
                   17    *                                                 *
                   18    ***************************************************

                   2Ø    *       FIXED LOCATIONS WE NEED

                   22    PTR      EQU   $48        WORK ZPAGE POINTER
                   23    HIMEM    EQU   $73        APPLESOFT HIMEM (START OF GP BUFFER)
                   24    IN       EQU   $2ØØ       INPUT LINE BUFFER
                   25    MLI      EQU   $BFØØ      MACHINE LANGUAGE INTERFACE
                   26    KBD      EQU   $CØØØ      KEYBOARD LATCH
                   27    KBDSTB   EQU   $CØ1Ø      KEYBOARD CLEAR STROBE
                   28    COUT     EQU   $FDED      MONITOR PRINT VECTOR

                   3Ø    *       SELECTED THINGS FROM THE BI GLOBAL PAGE

                   32             ORG   $BEØØ      START OF BI GLOBAL PAGE

BEØØ: 4C ØØ ØØ     34    BIENTRY  JMP   $ØØØØ      WARM ENTRY INTO PRODOS BI
BEØ3: 4C ØØ ØØ     35    DOSCMD   JMP   $ØØØØ      COMMAND EXECUTER
BEØ6: 4C ØØ ØØ     36    EXTCMD   JMP   $ØØØØ      EXTERNAL COMMAND VECTOR
BEØ9: 4C ØØ ØØ     37    ERROUT   JMP   $ØØØØ      EXIT WITH ERROR
BEØC: 4C ØØ ØØ     38    PRNTERR  JMP   $ØØØØ      PRINT ERROR MESSAGE
BEØF: ØØ           39    ERRCODE  DFB   Ø          ERROR CODE

                   41             ORG   $BE5Ø
BE5Ø: ØØ ØØ        42    XTADDR   DW    $ØØØØ      EXECUTION ADDRESS OF EXTERNAL CMD
BE52: ØØ           43    XLEN     DFB   Ø          LENGTH OF CMD STRING -1
BE53: ØØ           44    XCNUM    DFB   Ø          BI COMMAND NUMBER

                   46    FN1      EQU   $Ø1        FILE NAME EXPECTED
                   47    SD       EQU   $Ø4        SLOT/DRIVE PERMITTED
BE54: ØØ ØØ        48    PBITS    DW    Ø          ALLOWED PARAMETERS BITS
BE56: ØØ ØØ        49    FBITS    DW    Ø          PARAMETERS FOUND BITS

                   51             ORG   $BE6C
BE6C: ØØ ØØ        52    VPATH1   DW    $ØØØØ      ADDR OF PATHNAME 1 BUFFER
BE6E: ØØ ØØ        53    VPATH2   DW    $ØØØØ      ADDR OF PATHNAME 2 BUFFER

                   55    GOSYS    EQU   *          MLI CALL HANDLER

                   57             ORG   $BECB
BECB: Ø3           58    SOPEN    DFB   $Ø3        OPEN PARAMETER LIST
BECC: ØØ ØØ        59             DW    $ØØØØ
BECE: ØØ ØØ        6Ø    OSYSBUF  DW    $ØØØØ      BUFFER ADDR
BEDØ: ØØ           61    OREFNUM  DFB   $ØØ        REF NUM RETURNED
```

```
                63            ORG   $BED5
                64   SREAD    EQU   *          READ/WRITE PARM LIST
                65   SWRITE   EQU   *
BED5: Ø4        66            DFB   $Ø4        PARM COUNT = 4
BED6: ØØ        67   RWRFNUM  DFB   $ØØ        REFNUM
BED7: ØØ ØØ     68   RWDATA   DW    $ØØØØ      BUFFER ADDR
BED9: ØØ ØØ     69   RWCOUNT  DW    $ØØØØ      REQUEST LENGTH
BEDB: ØØ ØØ     7Ø   RWTRANS  DW    $ØØØØ      TRUE LENGTH

                72   SCLOSE   EQU   *          CLOSE/FLUSH PARM LIST
                73   SFLUSH   EQU   *
BEDD: Ø1        74            DFB   $Ø1        PARM COUNT = 1
BEDE: ØØ        75   CFRFNUM  DFB   $ØØ        REFNUM

                77            ORG   $BEF5
BEF5: 4C ØØ ØØ  78   GETBUFR  JMP   $ØØØØ      ALLOCATE BI BUFFER

                8Ø            ORG   $2ØØØ

                82   *       THIS PART OF THE PROGRAM GETS CONTROL WHEN THE
                83   *       BRUN COMMAND IS ISSUED. IT RELOCATES THE RESIDENT
                84   *       PART OF THE CODE TO THE TOP OF MEMORY

2ØØØ: A9 Ø1     86   TYPE     LDA   #1         WE NEED 1 PAGE
2ØØ2: 2Ø F5 BE  87            JSR   GETBUFR    BUY MEMORY FOR RESIDENT CODE
2ØØ5: 9Ø 1Ø     88            BCC   GOTBUF     GOT IT

2ØØ7: AØ ØØ     9Ø            LDY   #Ø
2ØØ9: B9 76 2Ø  91   ERLP     LDA   MSG,Y      NO BUFFER, PRINT MESSAGE
2ØØC: 2Ø ED FD  92            JSR   COUT
2ØØF: C8        93            INY
2Ø1Ø: C9 8D     94            CMP   #$8D
2Ø12: DØ F5     95            BNE   ERLP
2Ø14: 4C ØØ BE  96   EXIT     JMP   BIENTRY    AND LEAVE

2Ø17: 85 49     98   GOTBUF   STA   PTR+1      PTR --> BUFFER
2Ø19: AE Ø8 BE  99            LDX   EXTCMD+2
2Ø1C: 8D Ø8 BE  1ØØ           STA   EXTCMD+2   EXTCMD --> BUFFER
2Ø1F: 8E 4Ø 21  1Ø1           STX   NXTCOM+2
2Ø22: AE Ø7 BE  1Ø2           LDX   EXTCMD+1   BUT SAVE OLD EXTRNCMD VECTOR
2Ø25: 8E 3F 21  1Ø3           STX   NXTCOM+1
2Ø28: AØ ØØ     1Ø4           LDY   #Ø
2Ø2A: 84 48     1Ø5           STY   PTR
2Ø2C: 8C Ø7 BE  1Ø6           STY   EXTCMD+1

2Ø2F: B9 ØØ 21  1Ø8  COPY     LDA   TYPENT,Y   COPY RESIDENT CODE TO BUFFER
2Ø32: 91 48     1Ø9           STA   (PTR),Y
2Ø34: 8C 8F 2Ø  11Ø           STY   SAVE       SAVE YREG
2Ø37: 48        111           PHA
2Ø38: 29 Ø3     112           AND   #$Ø3       ISOLATE BIT OFFSET OF OPCODE
2Ø3A: A8        113           TAY
2Ø3B: 68        114           PLA
2Ø3C: 4A        115           LSR   A
2Ø3D: 4A        116           LSR   A          DIVIDE BY 4 (BYTE OFFSET)
2Ø3E: AA        117           TAX
2Ø3F: BD 9Ø 2Ø  118           LDA   OPTAB,X    GET 4 OPCODES' LENGTHS
2Ø42: 88        119  OPLOOP   DEY
2Ø43: 3Ø Ø4     12Ø           BMI   OPDONE     SHIFT OUT THE PROPER 2 BIT LEN
2Ø45: 4A        121           LSR   A
2Ø46: 4A        122           LSR   A
2Ø47: DØ F9     123           BNE   OPLOOP
2Ø49: 29 Ø3     124  OPDONE   AND   #$Ø3       ISOLATE LENGTH
2Ø4B: FØ 26     125           BEQ   COPIED     ZERO LENGTH OP ($FF) SIGNALS END
2Ø4D: AA        126           TAX
2Ø4E: AC 8F 2Ø  127           LDY   SAVE       RESTORE YREG
```

```
2Ø51: EØ Ø3     128             CPX   #3
2Ø53: FØ ØC     129             BEQ   RELOC      RELOCATE ALL 3 BYTE OPS
2Ø55: C8        13Ø             INY
2Ø56: CA        131             DEX
2Ø57: FØ D6     132             BEQ   COPY       1 BYTE OP?
2Ø59: B9 ØØ 21  133             LDA   TYPENT,Y   NO, THAT LEAVES 2 BYTE OPS
2Ø5C: 91 48     134  STMSB      STA   (PTR),Y
2Ø5E: C8        135             INY
2Ø5F: DØ CE     136             BNE   COPY       CONTINUE COPYING

                138  *         RELOCATE ABSOLUTE ADDRESSES IN INSTRUCTIONS

2Ø61: C8        14Ø  RELOC      INY
2Ø62: B9 ØØ 21  141             LDA   TYPENT,Y   COPY LSB OF ADDR
2Ø65: 91 48     142             STA   (PTR),Y
2Ø67: C8        143             INY
2Ø68: B9 ØØ 21  144             LDA   TYPENT,Y
2Ø6B: C9 21     145             CMP   #>TYPENT   THIS ADDR WITHIN TYPENT?
2Ø6D: DØ ED     146             BNE   STMSB      NO
2Ø6F: A5 49     147             LDA   PTR+1      YES, USE MSB OF NEW HOME
2Ø71: DØ E9     148             BNE   STMSB      ALWAYS TAKEN

                15Ø  *         RESIDENT CODE HAS BEEN INSTALLED, WE CAN EXIT

2Ø73: 4C ØØ BE  152  COPIED     JMP   BIENTRY    DONE, EXIT

                154  *         INSTALLATION DATA

2Ø76: CE CF AØ  156  MSG        ASC   "NO ROOM FOR NEW COMMAND"
2Ø79: D2 CF CF CD AØ C6 CF D2
2Ø81: AØ CE C5 D7 AØ C3 CF CD
2Ø89: CD C1 CE C4
2Ø8D: 87 8D     157             DFB   $87,$8D

2Ø8F: ØØ        159  SAVE       DFB   Ø          SAVE AREA FOR YREG

                161  *         EACH BYTE CONTAINS THE LENGTHS OF 4 65Ø2 OPCODES
                162  *            FOR EXAMPLE, AT +Ø IS A $59.
                163  *            IN BINARY $59 = Ø1Ø11ØØ1 OR
                164  *                            Ø1 Ø1 1Ø Ø1  (1,1,2,1)
                165  *            THESE ARE THE LENGTHS (IN REVERSED ORDER) FOR BRK,
                166  *            ORA (N,X), AND TWO UNDEFINED OPCODES.

2Ø9Ø: 59 69 59  168  OPTAB      DFB   $59,$69,$59,$7D OPCODE LENGTH TABLE
2Ø93: 7D
2Ø94: 5A 69 5D  169             DFB   $5A,$69,$5D,$7D
2Ø97: 7D
2Ø98: 5B 6A 59  17Ø             DFB   $5B,$6A,$59,$7F
2Ø9B: 7F
2Ø9C: 5A 69 5D  171             DFB   $5A,$69,$5D,$7D
2Ø9F: 7D
2ØAØ: 59 69 59  172             DFB   $59,$69,$59,$7F
2ØA3: 7F
2ØA4: 5A 69 5D  173             DFB   $5A,$69,$5D,$7D
2ØA7: 7D
2ØA8: 59 69 59  174             DFB   $59,$69,$59,$7F
2ØAB: 7F
2ØAC: 5A 69 5D  175             DFB   $5A,$69,$5D,$7D
2ØAF: 7D
2ØBØ: 59 6A 55  176             DFB   $59,$6A,$55,$7F
2ØB3: 7F
2ØB4: 5A 6A 5D  177             DFB   $5A,$6A,$5D,$5D
2ØB7: 5D
2ØB8: 6A 6A 59  178             DFB   $6A,$6A,$59,$7F
2ØBB: 7F
```

```
2ØBC: 5A 6A 5D  179          DFB   $5A,$6A,$5D,$7F
2ØBF: 7F
2ØCØ: 5A 6A 59  18Ø          DFB   $5A,$6A,$59,$7F
2ØC3: 7F
2ØC4: 5A 69 5D  181          DFB   $5A,$69,$5D,$7D
2ØC7: 7D
2ØC8: 5A 6A 59  182          DFB   $5A,$6A,$59,$7F
2ØCB: 7F
2ØCC: 5A 69 5D  183          DFB   $5A,$69,$5D,$3D ONLY $FF GIVES Ø LEN
2ØCF: 3D

                185  *       NOW STARTS THE RESIDENT CODE WHICH IS MOVED TO HIGH
                186  *       MEMORY.

                188  *       TYPENT IS CALLED BY THE BI WHENEVER IT DOESN'T
                189  *       RECOGNIZE A COMMAND. WE TAKE A LOOK AT IT SO SEE IF
                19Ø  *       IT MIGHT BE THE "TYPE" COMMAND.

                192          ORG   TYPE+256   MUST BE PAGE ALIGNED
21ØØ: D8        193  TYPENT  CLD              ; IDENTIFY TO BI
21Ø1: AD 6C BE  194          LDA   VPATH1     COPY COMMAND LINE PTR
21Ø4: 85 48     195          STA   PTR
21Ø6: AD 6D BE  196          LDA   VPATH1+1
21Ø9: 85 49     197          STA   PTR+1
21ØB: AØ Ø1     198          LDY   #1
21ØD: B1 48     199  COMPR   LDA   (PTR),Y    COMPARE COMMAND STRING
21ØF: D9 AC 21  2ØØ          CMP   NAME-1,Y   TO "TYPE"
2112: DØ 29     2Ø1          BNE   NOTIT      NOT IT, VECTOR ON
2114: C8        2Ø2          INY
2115: CØ Ø5     2Ø3          CPY   #5
2117: 9Ø F4     2Ø4          BCC   COMPR      CHECK ALL 4 CHARS

                2Ø6  *       IT HAS BEEN DETERMINED THAT THIS COMMAND IS MINE.
                2Ø7  *       RETURN TO THE BI TO PARSE THE PATHNAME OPERAND

2119: 88        2Ø9          DEY
211A: 8C 52 BE  21Ø          STY   XLEN       STORE COMMAND LINE INDEX
211D: A9 ØØ     211          LDA   #Ø
211F: 8D 53 BE  212          STA   XCNUM      COMMAND NUMBER Ø (EXTERN)
2122: A9 Ø4     213          LDA   #SD        SLOT/DRIVE PERMITTED
2124: 8D 55 BE  214          STA   PBITS+1
2127: A9 Ø1     215          LDA   #FN1       WE NEED PATHNAME1
2129: 8D 54 BE  216          STA   PBITS
212C: AD 44 21  217  WHERE   LDA   TYPBAK     <USED TO FIND TYPBAK>
212F: AD 2D 21  218          LDA   WHERE+1    TELL BI WHERE TO RETURN
2132: 8D 5Ø BE  219          STA   XTADDR
2135: AD 2E 21  22Ø          LDA   WHERE+2
2138: 8D 51 BE  221          STA   XTADDR+1
213B: 18        222          CLC              ; INDICATE COMMAND WAS FOUND
213C: 6Ø        223          RTS              ; BACK TO BI FOR MORE PARSING

                225  *       IF WE DON'T CLAIM A COMMAND, PASS IT THROUGH TO ANY
                226  *       OTHER EXTERNAL COMMAND HANDLERS

213D: 38        228  NOTIT   SEC              ; INDICATE COMMAND NOT FOUND
213E: 4C ØØ ØØ  229  NXTCOM  JMP   $ØØØØ      <OLD EXTCMD VECTOR>

2141: 4C Ø9 BE  231  TYPERR  JMP   ERROUT     VECTOR TO BI ERROR EXIT

                233  *       ONCE THE COMMAND HAS BEEN PARSED, THE BI CALLS THE
                234  *       FOLLOWING CODE TO FINISH HANDLING THE COMMAND

2144: A4 74     236  TYPBAK  LDY   HIMEM+1    MSB OF BUFFER AREA
2146: 8C CF BE  237          STY   OSYSBUF+1  COPY TO OPEN LIST
2149: A9 ØØ     238          LDA   #Ø
```

```
214B: 8D CE BE  239           STA   OSYSBUF
214E: A9 C8     24Ø           LDA   #$C8        MLI: OPEN
215Ø: 2Ø 7Ø BE  241           JSR   GOSYS       OPEN THE FILE
2153: BØ EC     242           BCS   TYPERR      ERROR?
2155: AD DØ BE  243           LDA   OREFNUM     COPY REF NUM
2158: 8D D6 BE  244           STA   RWRFNUM     TO READ LIST
215B: 8D DE BE  245           STA   CFRFNUM     AND CLOSE LIST

                247  *       FILE IS OPEN, READ 256 BYTES AT A TIME

215E: AØ ØØ     249  TYPLP    LDY   #Ø
216Ø: 8C D7 BE  25Ø           STY   RWDATA
2163: 8C D9 BE  251           STY   RWCOUNT
2166: C8        252           INY
2167: 8C DA BE  253           STY   RWCOUNT+1  256 BYTES AT A TIME
216A: C8        254           INY
216B: 8C D8 BE  255           STY   RWDATA+1    TO $2ØØ
216E: A9 CA     256           LDA   #$CA        MLI: READ
217Ø: 2Ø 7Ø BE  257           JSR   GOSYS       READ 256 BYTES TO $2ØØ
2173: 9Ø ØE     258           BCC   TYPPRT      ALL WENT WELL
2175: C9 Ø5     259           CMP   #5          EOF ERROR?
2177: DØ C8     26Ø           BNE   TYPERR      NO, REAL ERROR
2179: A9 8D     261  TYQUIT   LDA   #$8D
217B: 2Ø ED FD  262           JSR   COUT        PRINT A FINAL NEWLINE
217E: A9 CC     263           LDA   #$CC        MLI: CLOSE
218Ø: 4C 7Ø BE  264           JMP   GOSYS       EXIT THROUGH CLOSE
                266  *       COPY READ BUFFER TO SCREEN

2183: AØ ØØ     268  TYPPRT   LDY   #Ø
2185: B9 ØØ Ø2  269  TYPPLP   LDA   $2ØØ,Y      COPY BYTE BY BYTE
2188: Ø9 8Ø     27Ø           ORA   #$8Ø        MSB ON FOR COUT
218A: 2Ø ED FD  271           JSR   COUT        TO OUTPUT VECTOR
218D: AD ØØ CØ  272           LDA   KBD         CHECK FOR INTERVENTION
219Ø: 1Ø 13     273           BPL   TYPON       NOTHING, CONTINUE
2192: 8D 1Ø CØ  274           STA   KBDSTB      CLEAR STROBE
2195: C9 83     275           CMP   #$83        CONTROL-C?
2197: FØ EØ     276           BEQ   TYQUIT      YES, EXIT NOW
2199: AD ØØ CØ  277  TYPWT    LDA   KBD         WAIT FOR A SECOND KEYPRESS
219C: 1Ø FB     278           BPL   TYPWT
219E: 8D 1Ø CØ  279           STA   KBDSTB      CLEAR STROBE
21A1: C9 83     28Ø           CMP   #$83        CONTROL-C?
21A3: FØ D4     281           BEQ   TYQUIT      YES, ABORT
21A5: C8        282  TYPON    INY               ; ELSE, CONTINUE TYPING
21A6: CC DB BE  283           CPY   RWTRANS     UNTIL BUFFER EMPTY
21A9: DØ DA     284           BNE   TYPPLP
21AB: FØ B1     285           BEQ   TYPLP       THEN GO READ ANOTHER


21AD: 54 59 5Ø  287  NAME     ASC   'TYPE'          COMMAND NAME
21BØ: 45
21B1: FF FF FF  288           DFB   $FF,$FF,$FF END OF PROGRAM FLAGS

 --End assembly, 435 bytes, Errors: Ø
```

DUMBTERM—DUMB TERMINAL PROGRAM

DUMBTERM is an example of how to program under ProDOS using interrupts. DUMBTERM acts as a simple, line-at-a-time terminal emulation program which interfaces to a California Computer Systems CCS 7710 serial card. The same program can be written for an Apple Super Serial card (but interrupts are not as reliable for that card). The main portion of the program merely loops, checking the keyboard and the serial card for incoming data. If a keypress is found, it is sent out over the serial line, if incoming serial data is found, it is displayed on the screen.

The meat of the program lies within the communications subroutines in the last half of the listing. COMINT initializes the CCS card for interrupts after passing the address of its interrupt handler (COMIRQ) to ProDOS via the ALLOC_INTERRUPT MLI call. Each time an interrupt occurs, the COMIRQ handler is called by ProDOS and it examines the CCS status register to determine whether the interrupt was raised by the CCS card. If not, COMIRQ returns to ProDOS with the carry flag set to indicate that it is not claiming the interrupt. This gives other interrupt handlers a chance to service the interrupt. If the interrupt was generated by the CCS card and incoming data is available, a character is read and stored in a 256-byte circular buffer and COMIRQ exits to ProDOS.

The buffer is called circular because a pair of index pointers are used (start of data, end of data) to mark the actual data within the buffer and these pointers may wrap at the end of the buffer back to its beginning. Thus, conceptually the buffer has no beginning or end. This means that the main program may be doing something else but the interrupt routine can buffer up to 256 characters coming in from the serial port before it will lose data. If the main part of the program was ever vigilant and constantly checked for incoming serial data, there would be no need for an **interrupt exit**. However, each time the COUT screen output subroutine is called, there is a potential that control will not return before the next character is available. This is because the Apple scrolls the screen by moving every line up a byte at a time, one by one. The process of scrolling a 40-column screen lasts over one character time at 1200 baud (120 characters per second) on the serial port. Thus, without an interrupt exit, a character would be lost each time the screen is scrolled up one line.

Ideally this should be all there is to it. On an Apple II Plus, DUMBTERM works well under most circumstances and with most 80-column cards. Unfortunately this is not the case on an Apple IIe. Due to an error in programming the Apple IIe ROM, the entire process of scrolling the 40-column screen in PR#Ø mode

is disabled from interrupts! Thus the interrupt exit is useless in this mode. For 80-column scrolls, the ROM also disables interrupts while scrolling the bank switched text page, and the interrupt exit is again useless (at 1200 baud anyway). The only mode where the exit is reliable is the 40-column mode with `PR#3` (`control-Q`). There are ways of avoiding these problems for 1200 baud. One is to change the window size (so that the monitor has less data to scroll). This is done by storing a new bottom line value at `$23`. In `PR#Ø` 40-column mode, this value should be `$15`. In 80-column mode, it must be `$ØE`. Another solution would be to reproduce the scrolling code from the monitor into your own program and "sniff" for interrupts (i.e. enable for interrupts and disable again) more frequently than Apple does. It is also worth noting that some 80-column cards, such as the ALS Smarterm, "scroll" by moving a hardware "top of screen" pointer. No CPU time is required to scroll this way and terminal programs are much easier to write.

`DUMBTERM` is also an example of a simple Interpreter or System Program. It sets up the stack register and ProDOS version fields in the System Global Page upon entry, and it exits upon sensing a `control-C` keypress using the MLI `QUIT` call.

```
 1            ORG   $2ØØØ

 3     ***********************************************************
 4     *                                                         *
 5     * DUMBTERM: THIS PROGRAM ACTS AS A DUMB TERMINAL          *
 6     *           THROUGH A CCS 771Ø SERIAL CARD USING          *
 7     *           THE PRODOS INTERRUPT HANDLER FOR INPUT        *
 8     *           INTERRUPTS.  THIS PROGRAM FOLLOWS THE RULES   *
 9     *           FOR A PRODOS INTERPRETER.                     *
 1Ø    *                                                         *
 11    * ASSUMPTIONS: CCS771Ø CARD IN SLOT 1                     *
 12    *              8 DATA BITS, 1 STOP, NO PARITY             *
 13    *              BAUD RATE SET BY DIP SWITCHES ON CARD      *
 14    *                                                         *
 15    * ENTRY POINT: $2ØØØ                                      *
 16    *                                                         *
 17    * PROGRAMMER: DON D WORTH 3/8/84                          *
 18    *                                                         *
 19    ***********************************************************

 21    BELL      EQU   $87         BELL CHARACTER
 22    CR        EQU   $ØD         RETURN
 23    RETURN    EQU   $8D         PRINTABLE RETURN
 24    SPACE     EQU   $AØ         BLANK

 26    *        ZPAGE DEFINITIONS

 28    A1L       EQU   $3C         MONITOR POINTER
 29    A1H       EQU   $3D
 3Ø    A2L       EQU   $3E         MONITOR POINTER
 31    A2H       EQU   $3F

 33    *        OTHER ADDRESSES

 35    MLI       EQU   $BFØØ       PRODOS ENTRY POINT
```

```
                 36   MACHID   EQU   $BF98      PRODOS MACHINE ID
                 37   IBAKVER  EQU   $BFFC      MLI VERSION WANTED
                 38   IVERS    EQU   $BFFD      MY VERSION
                 39   KBD      EQU   $C000      KEYBOARD STROBE
                 40   KBDRES   EQU   $C010      KEYBOARD RESET
                 41   COUT     EQU   $FDED      MONITOR OUTPUT SUBROUTINE
                 42   RDKEY    EQU   $FD0C      MONITOR INPUT SUBROUTINE
                 43   HOME     EQU   $FC58      MONITOR HOME
                 44   TEXT     EQU   $FB39      SET TEXT MODE

                 46   *       START OF DUMB TERMINAL EMULATOR

2000: A2 FF      48   DTERM    LDX   #$FF       SET UP STACK POINTER
2002: 9A         49            TXS
2003: A9 00      50            LDA   #0
2005: 8D FC BF   51            STA   IBAKVER    VERSION 0 FOR EVERYBODY
2008: 8D FD BF   52            STA   IVERS
200B: 20 62 20   53            JSR   COMINT     INITIALIZE SERIAL PORT
200E: 20 58 FC   54            JSR   HOME       CLEAR SCREEN
2011: A9 02      55            LDA   #$02
2013: 2C 98 BF   56            BIT   MACHID     80 COLUMN CARD PRESENT?
2016: F0 03      57            BEQ   BEGIN      NO
2018: 20 00 C3   58            JSR   $C300      INITIALIZE 80 COLUMN CARD

                 60   *
                 61   * NOTE: NO NEED TO SET BIT MAP HERE SINCE I AM NOT DOING
                 62   *       DYNAMIC MEMORY ALLOCATION. ALSO, WE WILL LEAVE
                 63   *       THE POWERUP BYTE SUCH THAT A RESET FORCES REBOOT.
                 64   *
                 65   BEGIN    EQU   *
                 66   *
                 67   * --- SPECIAL NOTE ---
                 68   *
                 69   * IF AN APPLE IIe IS USED, THERE IS A POSSIBILITY OF DATA
                 70   * ERRORS AT 1200 BAUD WHEN SCROLLING THE SCREEN.  IF THIS
                 71   * OCCURS, INSERT THESE INSTRUCTIONS AT THIS POINT:
                 72   *
                 73   *       LDA     #$0E           SET WINDOW TO 14 LINES HIGH
                 74   *       STA     $23
                 75   *
                 76   * (LDA #$15 IF NO 80 COLUMN CARD IN YOUR IIe.)
                 77   *

                 79   *       MAIN TERMINAL LOOP

201B: AD F4 20   81   LOOP     LDA   ERRORS     HAVE ANY ERRORS OCCURED?
201E: F0 0A      82            BEQ   NOERRS     NO
2020: A9 87      83            LDA   #BELL      YES, BEEP AT HIM
2022: 20 ED FD   84            JSR   COUT
2025: A9 00      85            LDA   #0         AND CLEAR ERROR COUNTER
2027: 8D F4 20   86            STA   ERRORS
202A: AD 00 C0   87   NOERRS   LDA   KBD        FIRST TEST KEYBOARD
202D: 10 0F      88            BPL   NOKEY      NOTHING YET?
202F: 29 7F      89            AND   #$7F
2031: C9 03      90            CMP   #3         CONTROL-C?
2033: F0 1D      91            BEQ   EXIT
2035: AD 00 C0   92            LDA   KBD        RELOAD CHARACTER
2038: 8D 10 C0   93            STA   KBDRES     CLEAR KEYBOARD
203B: 20 CC 20   94            JSR   COMOUT     SEND THE CHARACTER OUT

203E: 20 B1 20   96   NOKEY    JSR   COMTIN     TEST FOR AVAILABLE INPUT
2041: F0 D8      97            BEQ   LOOP       NOTHING, CHECK KEYBOARD AGAIN
2043: 20 BA 20   98            JSR   COMINP     GET NEXT INPUT CHARACTER
2046: 09 80      99            ORA   #$80       MSB ON FOR OUTPUT
2048: C9 8A      100           CMP   #$8A       LINE FEED?
```

```
2Ø4A: FØ CF     1Ø1          BEQ   LOOP       YES, SKIP IT
2Ø4C: 2Ø ED FD  1Ø2          JSR   COUT       ELSE, PRINT IT
2Ø4F: 4C 1B 2Ø  1Ø3          JMP   LOOP       AND CONTINUE LOOP

                1Ø5  *       IF CONTROL-C IS TYPED, EXIT

2Ø52: 2Ø D9 2Ø  1Ø7  EXIT    JSR   COMCLS     CLOSE DOWN COMM LINE
2Ø55: 2Ø ØØ BF  1Ø8          JSR   MLI        MLI: QUIT CALL
2Ø58: 65        1Ø9          DFB   $65
2Ø59: 5B 2Ø     11Ø          DW    QPARMS
2Ø5B: Ø4        111  QPARMS  DFB   4          QUIT PARMLIST
2Ø5C: ØØ        112          DFB   Ø
2Ø5D: ØØ ØØ     113          DW    Ø
2Ø5F: ØØ        114          DFB   Ø
2Ø6Ø: ØØ ØØ     115          DW    Ø

                117  *************************************************
                118  *                                               *
                119  *       CCS 771Ø COMMUNICATIONS SUBROUTINES     *
                12Ø  *                                               *
                121  *************************************************

                123  CCCOM   EQU   $CØ9E      CCS COMMAND REG
                124  CCSTS   EQU   $CØ9E      CCS STATUS REG
                125  CCDTA   EQU   $CØ9F      CCS DATA REG

                127  *       COMINT: INITIALIZE THE COMM LINE

2Ø62: A9 ØØ     129  COMINT  LDA   #Ø
2Ø64: 8D F5 2Ø  13Ø          STA   CIRCS      START CIRCULAR BUFFER PTRS
2Ø67: 8D F6 2Ø  131          STA   CIRCE
2Ø6A: 2Ø ØØ BF  132          JSR   MLI        ALLOCATE INTERRUPT EXIT
2Ø6D: 4Ø        133          DFB   $4Ø
2Ø6E: FØ 2Ø     134          DW    APARMS
2Ø7Ø: A9 23     135          LDA   #$23       RESET ACIA
2Ø72: 8D 9E CØ  136          STA   CCCOM
2Ø75: A9 15     137          LDA   #$15       8 BITS/1 STOP/NO PARITY/NO INTS
2Ø77: 8D 9E CØ  138          STA   CCCOM
2Ø7A: AE 9F CØ  139          LDX   CCDTA      THROW AWAY ANY GARBAGE
2Ø7D: Ø9 8Ø     14Ø          ORA   #$8Ø       ENABLE INTERRUPTS
2Ø7F: 8D 9E CØ  141          STA   CCCOM
2Ø82: 6Ø        142          RTS

                144  *       COMIRQ: INTERRUPT EXIT

2Ø83: AD 9E CØ  146  COMIRQ  LDA   CCSTS      CHECK STATUS
2Ø86: 3Ø Ø2     147          BMI   COMME      INTERRUPT WAS FOR ME?
2Ø88: 38        148          SEC              ; INDICATE NOT MY INTERRUPT
2Ø89: 6Ø        149          RTS
2Ø8A: 48        15Ø  COMME   PHA
2Ø8B: 29 7Ø     151          AND   #$7Ø       ANY ERRORS OCCURED?
2Ø8D: FØ Ø3     152          BEQ   COMNE      NO
2Ø8F: EE F4 2Ø  153          INC   ERRORS     YES, BUMP ERROR COUNT
2Ø92: 68        154  COMNE   PLA
2Ø93: 29 Ø1     155          AND   #$Ø1       CHECK FOR INCOMMING DATA
2Ø95: FØ 18     156          BEQ   CLAIM      NOTHING, IGNORE OTHER INTERRUPTS
2Ø97: AD 9F CØ  157          LDA   CCDTA      GET INCOMMING BYTE
2Ø9A: AE F6 2Ø  158          LDX   CIRCE
2Ø9D: 9D F7 2Ø  159          STA   CIRC,X     STORE IT AT END OF BUFFER
2ØAØ: E8        16Ø          INX
2ØA1: 8E F6 2Ø  161          STX   CIRCE      UPDATE END POINT
2ØA4: EC F5 2Ø  162          CPX   CIRCS      WRAPPED BACK TO START?
2ØA7: DØ Ø6     163          BNE   CLAIM      NO, DID NOT OVERRUN
2ØA9: EE F4 2Ø  164          INC   ERRORS     OVERRUN ERROR, BUMP COUNT
2ØAC: CE F6 2Ø  165          DEC   CIRCE      BACK UP END POINT
```

```
20AF: 18        166  CLAIM    CLC                ; CLAIM THE INTERRUPT
20B0: 60        167           RTS

                169  *       COMTIN: TEST FOR AVAILABLE INPUT
                170  *               IF NEQ, DATA IS AVAILABLE

20B1: 78        172  COMTIN   SEI                ; DISABLE FROM INTERRUPTS
20B2: AE F5 20  173           LDX   CIRCS        CHECK CIRCULAR BUFFER
20B5: EC F6 20  174           CPX   CIRCE        SEE IF ITS EMPTY
20B8: 58        175           CLI                ; REENABLE
20B9: 60        176           RTS

                178  *       COMINP: WAIT FOR NEXT INPUT, RETURN IN AREG

20BA: 20 B1 20  180  COMINP   JSR   COMTIN       TEST STATUS
20BD: F0 FB     181           BEQ   COMINP       NOTHING YET?
20BF: 78        182           SEI                ; DISABLE TO MESS WITH CIRC
20C0: AE F5 20  183           LDX   CIRCS
20C3: BD F7 20  184           LDA   CIRC,X       GET INPUT CHARACTER
20C6: E8        185           INX
20C7: 8E F5 20  186           STX   CIRCS        BUMP START POINTER FORWARD
20CA: 58        187           CLI                ; REENABLE FOR INTERRUPTS
20CB: 60        188           RTS

                190  *       COMOUT: OUTPUT A BYTE FROM AREG

20CC: 48        192  COMOUT   PHA
20CD: AD 9E C0  193  COMOL    LDA   CCSTS        CHECK STATUS
20D0: 29 02     194           AND   #$02         ISOLATE TX BUFFER BIT
20D2: F0 F9     195           BEQ   COMOL        NOT READY YET
20D4: 68        196  COMOIT   PLA
20D5: 8D 9F C0  197           STA   CCDTA        SEND BYTE
20D8: 60        198           RTS

                200  *       COMCLS: CLOSE COMM PORT

20D9: A9 23     202  COMCLS   LDA   #$23         STOP INTERRUPTS/DTR OFF
20DB: 8D 9E C0  203           STA   CCCOM
20DE: 58        204           CLI                ; JUST FOR SAFETY SAKE
20DF: A9 01     205           LDA   #1
20E1: 8D F0 20  206           STA   APARMS       CHANGE PARMLIST
20E4: 20 00 BF  207           JSR   MLI
20E7: 41        208           DFB   $41          DEALLOC_INTERRUPT
20E8: F0 20     209           DW    APARMS
20EA: A9 02     210           LDA   #2
20EC: 8D F0 20  211           STA   APARMS       LEAVE THINGS AS I FOUND THEM
20EF: 60        212           RTS

                214  *       DATA

20F0: 02        216  APARMS   DFB   2            ALLOC_INTERRUPT PARMS
20F1: 00        217           DFB   0
20F2: 83 20     218           DW    COMIRQ

20F4: 00        220  ERRORS   DFB   0            ERROR STATISTICS
20F5: 00        221  CIRCS    DFB   0            START OF DATA IN CIRC
20F6: 00        222  CIRCE    DFB   0            END OF DATA IN CIRC
20F7: 00 00 00  223  CIRC     DS    256          CIRCULAR INPUT BUFFER
20FA: 00 00 00 00 00 00 00 00
2102: 00 00 00 00 00 00 00 00
210A: 00 00 00 00 00 00 00 00
2112: 00 00 00 00 00 00 00 00
211A: 00 00 00 00 00 00 00 00
2122: 00 00 00 00 00 00 00 00
212A: 00 00 00 00 00 00 00 00
```

```
2132: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
213A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2142: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
214A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2152: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
215A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2162: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
216A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2172: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
217A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2182: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
218A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
2192: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
219A: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21A2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21AA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21B2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21BA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21C2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21CA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21D2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21DA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21E2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21EA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
21F2: ØØ ØØ ØØ ØØ ØØ

--End assembly, 5Ø3 bytes, Errors: Ø
```

Protecting Against Software Pirates

Appendix B

Diskette Protection Schemes

Protected software, that software which is modified in some way to prevent it from being copied or duplicated, has existed since very early in the history of the Apple II. This was even true of tape based software before disk drives were widely used. It is not known who protected the first piece of Apple software, but it has become a widespread practice. So has the practice of copying or **breaking** protected software. It should be pointed out that the following discussion will not take sides in the sometimes controversial subject of software protection. Rather, it will provide an informative look at the methods used to protect software and how those methods bave been circumvented. This seems appropriate since almost all protection schemes now involve a modified or custom disk operating system.

At this time, ProDOS is still relatively new and it is unclear if it will influence the current practice of protecting software. In that a ProDOS disk is identical to earlier operating systems (DOS 3.3) at a byte level, it is certainly possible and probable that protection will exist. However, since ProDOS can and will support other storage devices (i.e. hard disks etc.), and with the current trend in sharing data between different applications, additional challenges exist for software developers. It is possible that the percentage of protected software may decrease somewhat with the introduction of ProDOS. The following discussion will deal with software protection in general on the Apple II family of computers.

A BRIEF HISTORY OF APPLE SOFTWARE PROTECTION

The first protected software was tape based and appeared in the latter part of 1978, and protected disks followed shortly thereafter. Early protection schemes were often quite effective as there was relatively little technical information available. Almost any modification that rendered the normal means of copying useless was sufficient in most cases—most schemes did in fact consist of relatively minor changes to the normal format of data. Individuals were able to discover and disable these protection methods on a program by program

basis, with little or no thought given to some automated means of reproducing protected software.

It was not until perhaps a year later, in late 1979, that a significant event occurred in disk protection. An extremely popular product was introduced that employed a considerably improved protection method. This marked the beginning of an escalating battle between those protecting software and those trying to copy it. The protection methods used became more and more complex and involved, increasing time and expense for developers to create. The copiers were also increasing their efforts. Programs appeared that were designed to copy particular software products — a major development in that it defeated a great number of different schemes with a single basic technique. These programs are referred to as **nibble copiers** and were introduced in early 1981.

Throughout this process, it is clear that both sides made use of the work of their counterparts. Protection schemes started to reflect a working knowledge of breaking techniques, and were often designed to circumvent a particular method or copier. The people breaking protection methods were also studying the various methods employed to stop them and producing increasingly effective tools. This produced a kind of ebb and flow seen in many competitive areas where each side gains a temporary advantage only to see it lost. Nibble copiers have had numerous revisions to cope with advancements in protection methods.

Another significant milestone was the introduction of a **hardware card** that could copy software from the Apple's memory, thus bypassing most existing protection methods. While it is hard to single out advancements in protection methods, the mere presence of the numerous copy programs, hardware devices, bulletin boards, classes, and magazines aimed at defeating protection methods indicates the constant advancement of protection. Also, the fact that software developers continue to protect software in the face of escalating costs indicates protection is still cost efective.

The cycle will no doubt continue. As new sophisticated schemes are developed, they will be broken by equally sophisticated schemes.

PROTECTION METHODS

It seems reasonable at this time to say that it is impossible to protect a program on disk in such a way that it can't be broken. This is, in large part, due to the nature of the Apple computer and its disk drive. It is an extremely well documented machine, with numerous publications available on both hardware and software functions. It is indeed difficult to hide anything (necessary in

protecting software) from anyone who is willing to invest sufficient time to find it.

Most disk protection methods fall into two different types of schemes. The first involves **format alterations**, altering some portion of the disk from its normal format (Chapter 3 and Appendix C provide descriptions of the normal format). The second involves creating an identifiable mark or signature that can be used to verify the disk.

FORMAT ALTERATION

A great number of ways exist to alter the format of normal data. They range from a single byte change to an entirely different format. A special case is **changing the location** of data, and not necessarily the structure of the data itself. An early example of this was moving the directory information from its normal location to a different track altogether. Later, tracks themselves were moved when "half" tracks becam popular (but data must be a full track apart from other data, a restriction imposed by hardware). Some disks now even use quarter tracks. Although these methods were effective for a while, most nibble copiers are equipped to handle them.

A more elaborate technique used is known as **spiral tracks**. Data is staggered on alternating half tracks producing, as its name indicates, a spiral of sorts. Each half track contains approximately one third of a track of data. If the relationship of the different segments is critical, this method of protection can be quite difficult to deal with. Several copy programs are capable of handling this, but may require parameters and additional time to reproduce a disk protected in this manner.

As with location changes, **format changes** range from simple to complex. Almost all early changes were merely minor modifications to existing operating systems. The most common change was a change to the code that would read and write the Address Field. This was reasonable because the Address Field is never rewritten, and the only special code required was the code to read the modified Address Field.

The Address Field normally starts with the bytes `$D5/$AA/$96`. If any of these bytes were changed, a standard operating system would not be able to locate that particular Address Field, causing an error. After the Address Field comes the address information itself (volume, track, sector, and checksum). Some common techniques include changing the order of this information, doubling the sector numbers, or altering the checksum with some constant. Any of the above would cause an error on a standard operating system. The

Address Field ends with two closing bytes (`$DE/$AA`), which can be changed or switched also. Similar kinds of changes can be made to the Data Field. These techniques worked well until automated programs appeared.

The first automated programs were good but generally made the assumption that the data portions had been modified and that the various gaps between the data portions were normal. This prompted modification of the gaps and eventually a radically different format in an attempt to circumvent the copy programs. These formats generally involved either different numbers of otherwise normal sectors on a track, or special sectors with Address and Data Fields combined. As with other advancements, this worked well for a time, but current nibble copiers make as few assumptions about the data format as possible and can generally deal with such techniques.

Here is the standard DOS 3.3 RWTS code to find the first nibble of the Address Field:

```
B94F- BD 8C CØ     LDA   $CØ8C,X
B952- 1Ø FB        BPL   $B94F
B954- C9 D5        CMP   #$D5        (2 cycles)
B956- DØ FØ        BNE   $B948       (2)
B958- EA           NOP               (2)
```

Now imagine a slight variation of this code:

```
B94F- BD 8C CØ     LDA   $CØ8C,X
B952- 1Ø FB        BPL   $B94F
B954- 4A           LSR               (2)
B955- C9 6A        CMP   #$6A        (2)
B957- DØ EF        BNE   $B948       (2)
```

This variation has three important properties: it fits in the same number of bytes as the original, it runs in the same number of cycles, and it matches the same `$D5` nibble. However, it will also match a `$D4` nibble, because the `LSR + CMP` effectively treats the low bit as a wildcard.

In binary: `$D5` = `11Ø1 Ø1Ø1`
After LSR: `Ø11Ø 1Ø1Ø` = `$6A`

In binary: `$D4` = `11Ø1 Ø1ØØ`
After LSR: `Ø11Ø 1Ø1Ø` = `$6A`

By changing these five bytes of code, we have a unified RWTS that can read protected disks (`$D4/$AA/$96`) and unprotected disks (`$D5/$AA/$96`).

Now consider a second, more complex variation:

```
B94F- BD 8C CØ     LDA   $CØ8C,X
B952- 1Ø FB        BPL   $B94F
B954- C9 D5        CMP   #$D5        (2 cycles)
B956- DØ FØ        BNE   $B948       (2)
B958- EA           NOP               (2)
B959- BD 8C CØ     LDA   $CØ8C,X     (4)
B95C- C9 D5        CMP   #$D5        (2)
B95E- FØ 12        BEQ   $B972
B96Ø- BD 8C CØ     LDA   $CØ8C,X
B963- 1Ø FB        BPL   $B96Ø
B965- C9 AA        CMP   #$AA
B967- DØ DF        BNE   $B948
B969- BD 8C CØ     LDA   $CØ8C,X
B96C- 1Ø FB        BPL   $B969
B96E- C9 96        CMP   #$96
B97Ø- DØ D6        BNE   $B948
```

This RWTS allows for two radically different ways of starting an Address Field. It starts by matching a `$D5` nibble, as usual. But instead of moving on to the second nibble immediately, at `$B959` it checks the data latch exactly once to see if it is still `$D5`. In the absence of a new "`1`" bit, the data latch will hold its value long enough for the `CMP` at `$B95C` to detect a "`Ø`" bit after the `$D5` nibble. If found, it branches to `$B972` to parse the rest of the address field; otherwise it continues with the rest of the standard prologue (`$AA`/`$96`). Once again, we have a unified RWTS that can read protected and unprotected disks.

A third variation targeted the epilogue at the end of the Address Field. This is normally a sequence of two nibbles, `$DE`/`$AA`. Here is the DOS 3.3 code to match them:

```
B98B- BD 8C CØ     LDA   $CØ8C,X
B98E- 1Ø FB        BPL   $B98B
B99Ø- C9 DE        CMP   #$DE
B992- DØ AE        BNE   $B942
B994- EA           NOP
B995- BD 8C CØ     LDA   $CØ8C,X
B998- 1Ø FB        BPL   $B995
B99A- C9 AA        CMP   #$AA
B99C- DØ A4        BNE   $B942
B99E- 18           CLC
B99F- 6Ø           RTS
```

But if we ignored the second `$AA` nibble, it would free up enough space for some fun tricks (not guaranteed, actual fun may vary).

```
B98B- BD 8C CØ     LDA   $CØ8C,X
B98E- 1Ø FB        BPL   $B98B
B99Ø- C9 DE        CMP   #$DE
B992- FØ ØA        BEQ   $B99E
B994- 48           PHA               (3 cycles)
```

```
B995- 68          PLA             (4)
B996- BD 8C CØ    LDA   $CØ8C,X   (4)
B999- C9 Ø8       CMP   #$Ø8      (2)
B99B- BØ A5       BCS   $B942
B99D- EA          NOP
B99E- 18          CLC
B99F- 6Ø          RTS
```

It starts by looking for a `$DE` nibble, as usual. If found, it immediately exits with carry bit clear, indicating success. But if the first epilogue nibble is not `$DE`, we fall through to `$B994` and intentionally burn CPU cycles before checking the data latch exactly once at `$B999`. At this point, enough time has elapsed for the data latch to read some, but not all, of the bits of the next nibble. In other words, its value is dependent on the number of "`Ø`" bits after the first nibble. Once again, we have a unified RWTS that can read protected and unprotected disks.

SIGNATURE

The earliest example of a signature was probably an unused track (track 3 was commonly "un"used). The software verifies the signature by trying to read a sector on the unused track. In an error occurred, the signature was verified. As simple as this seems now, it was reasonably effective. While this is a fairly obvious example of a signature, later methods were much more difficult to detect. In fact, most signatures have been uncovered by finding and examining the code that verified it. Once a method was known, an algorithm could be developed to deal with it.

There are three common signatures used currently in protecting disks. The first to appear involves counting the number of bytes on a given track. This is commonly known as **nibble counting**. The reasoning was that no two drives could spin at precisely the same speed, and therefore would not reproduce a track precisely. While this is in fact true, a number of programs now provide the means to reproduce this type of signature.

Next to arrive was a method that was dependent on the positional relationship between different portions of the disk. This is commonly known as **synchronized tracks**. It generally involves reading a specific sector, then moving the disk arm to another track (often with nonstandard timing), and finding a particular sector first. The angle between the two sectors is arbitrary, but will always provide just enough time to move the arm and allow for any settling time needed. This relationship between tracks would not normally be maintained when copying the disk, and the signature would thus be removed.

This also is provided for in many current copying programs, sometimes requiring parameters for a particular disk.

The final method involves writing extra zero bits at given locations on disk. These can be thought of as **special sync bytes**. When the disk is read, these extra bits are normally discarded.

The most popular such bit pattern involved a sequence of `$E7` nibbles. What would such a sequence look like on disk? The answer is: *It depends.* `$E7` in hexadecimal is `11100111` in binary, so here is the simplest possible answer:

```
|<----$E7---->|<----$E7---->|<----$E7---->|<----$E7---->|
1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1
```

But wait. Every nibble read from disk must have its high bit set. In theory, we could insert one or two "0" bits after any of those nibbles. (Two is the maximum due to hardware limitations.) These extra "0" bits would be swallowed by the standard "wait for data latch to have its high bit set" loop, which we see over and over in any RWTS code:

```
:1    LDA $C08C,X
      BPL :1
```

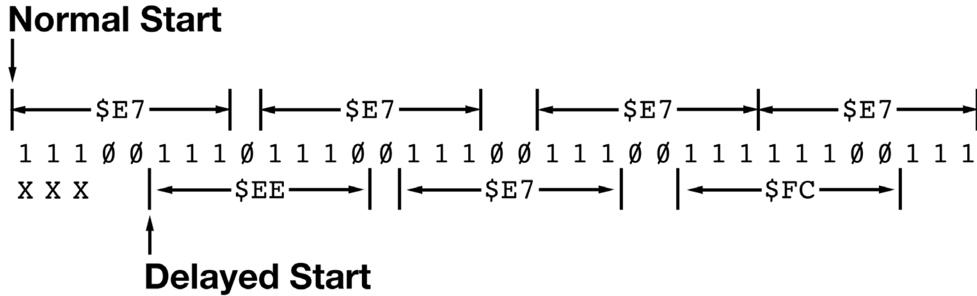
Now consider the following bitstream:

```
|<----$E7---->| |<----$E7---->|   |<----$E7---->|<----$E7---->|
1 1 1 0 0 1 1 1 0 1 1 1 0 0 1 1 1 0 0 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1
                ^                 ^
                 \               /
                      EXTRA
```

The first `$E7` has one extra "0" bit after it, and the second `$E7` has two extra "0" bits after it. Totally legal, works on any Apple II computer and any floppy drive. A `LDA $C08C,X; BPL` loop would still interpret this bitstream as a sequence of four `$E7` nibbles. Each of the extra "0" bits appear after we've just read a nibble and we're waiting for the high bit to be set again. They get "swallowed"—ignored, like they were never there.

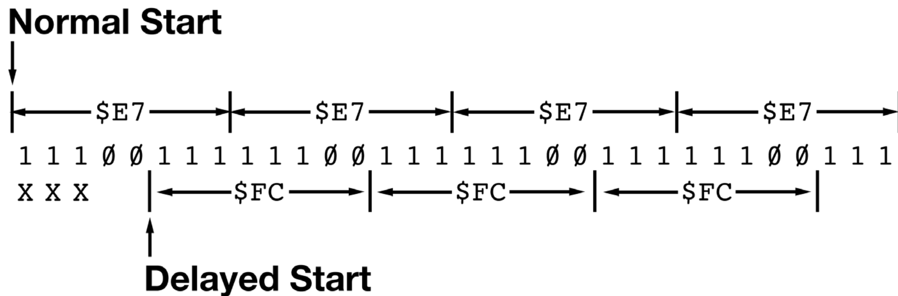
But what if we miss the first few bits of this bitstream, then start looking? The disk is always spinning, whether we're reading from it or not. If we waste too much time doing something other than reading, we'll literally miss some bits as the disk spins. (This is why the timing of low-level RWTS code is so critical.)

Let's say we waste 12 CPU cycles before we start reading this bitstream. Each bit takes 4 CPU cycles to go by, so after 12 cycles, we would have missed the first 3 bits (marked with an X).



Now it's interpreted as a completely different nibble sequence, just by delaying a few CPU cycles before we start reading. Also note that some of those extra bits are no longer being ignored; now they're being interpreted as data, as part of the nibbles that are being returned to the higher level code. Meanwhile, other bits that used to be part of an `$E7` nibble are now being swallowed.

Now, let's go back to the first stream, which had no extra bits between the nibbles, and see what happens when we waste those same 12 CPU cycles.



After skipping the first three bits, the stream is interpreted as a series of `$FC $FC $FC` repeating endlessly—not `$EE $E7 $FC` like the other stream.

Nibble copiers eventually implemented routines to detect hidden "`Ø`" bits after any nibble, not just self-sync `$FF` nibbles. But even the most sophisticated copier couldn't detect whether a nibble was followed by one or two "`Ø`" bits. By wasting just the right number of CPU cycles at just the right time, then interpreting the bits on the disk in mid-stream, developers could determine at runtime whether you had an original disk.

Here is the complete "**$E7** bitstream," annotated to show both the synchronized and desynchronized nibble sequences.

```
|<-$E7->||<-$E7->| |<-$E7->|<-$E7->||<-$E7->| |<-$E7->|<-$E7->||<-$E7->||<-$E7->|<-$E7->|
111ØØ111Ø111ØØ111ØØ111ØØ1111111ØØ111Ø111ØØ111ØØ111ØØ1111111ØØ111Ø111ØØ111Ø111ØØ1111111ØØ111
XXX |<-$EE->||<-$E7->| |<-$FC->|<-$EE->||<-$E7->| |<-$FC->|<-$EE->||<-$EE->||<-$FC->|
```

We have dealt primarily with disk protection schemes and nibble copiers, but several other methods of protection exist. These are protection methods which do not allow a program to be taken out of memory and patched to disable the protection scheme. It is worth mentioning that copies produced by a nibble copier are themselves protected, but software broken in some other way may be copied by normal means.

MEMORY PROTECTION

It has long been realized that software is vulnerable as it is being loaded into memory, and when it resides entirely in memory. This has prompted a number of techniques, the earliest of which involved **reset protection**. When the Reset key was pressed (on early Apples), the software could be interrupted and was then resident in memory. Several memory locations were altered during a reset, and many programs were dependent on the values contained in those locations. The later Apple computers provide some measure of protection in that they make it much harder to interrupt software programs. The hardware boards designed to copy software from memory have made memory protection very difficult. The boards generate a Non-Maskable Interrupt and pass control to on-board software. It is not possible to prevent this interrupt from software. About the only defense is simply to never have the entire program in memory at one time. This is often inconvenient but may be the only effective defense.

CODE PROTECTION

Hiding the code that reads the unusual disk format or checks for a particular signature has become increasingly popular. Early schemes rarely tried to hide anything because there were few people who knew where to look or even what to look for. But it is clear now that most of the advancements in nibble copiers resulted from the examination of the actual code that provided the protection. Signature schemes would have been effective much longer if it had been possible to hide the code that verified them. While it is impossible to prevent the code from being found, it can be made more difficult. The general

method used is some sort of encryption of the code. It is decrypted just before execution, and either encrypted again or destroyed just after execution.

THE IDEAL PROTECTION SCHEME

There are thousands of programs available for the Apple II family of machines, and it is safe to say that they have all been copied despite a vast array of protection schemes. It seems reasonable to assume that this fact will not change. Nevertheless, it may be possible to devise a reasonably effective method. It would have to address the three primary ways that software is broken—nibble copiers, hardware boards that copy memory, and what we call the "front door" method.

NIBBLE COPIERS

Nibble copy programs have an advantage of sorts in that they need only respond to existing protection methods. This clearly requires considerable skill but not necessarily creativity. In fairness though it should be noted that at least one of the nibble copiers has included capabilities that may effectively deal with yet to be created protection schemes. The best that one should hope for is a protection method that requires parameters to be input by the user of the copier. If the method could be varied so that each variation required a different set of parameters, it would be considered a victory.

HARDWARE BOARDS

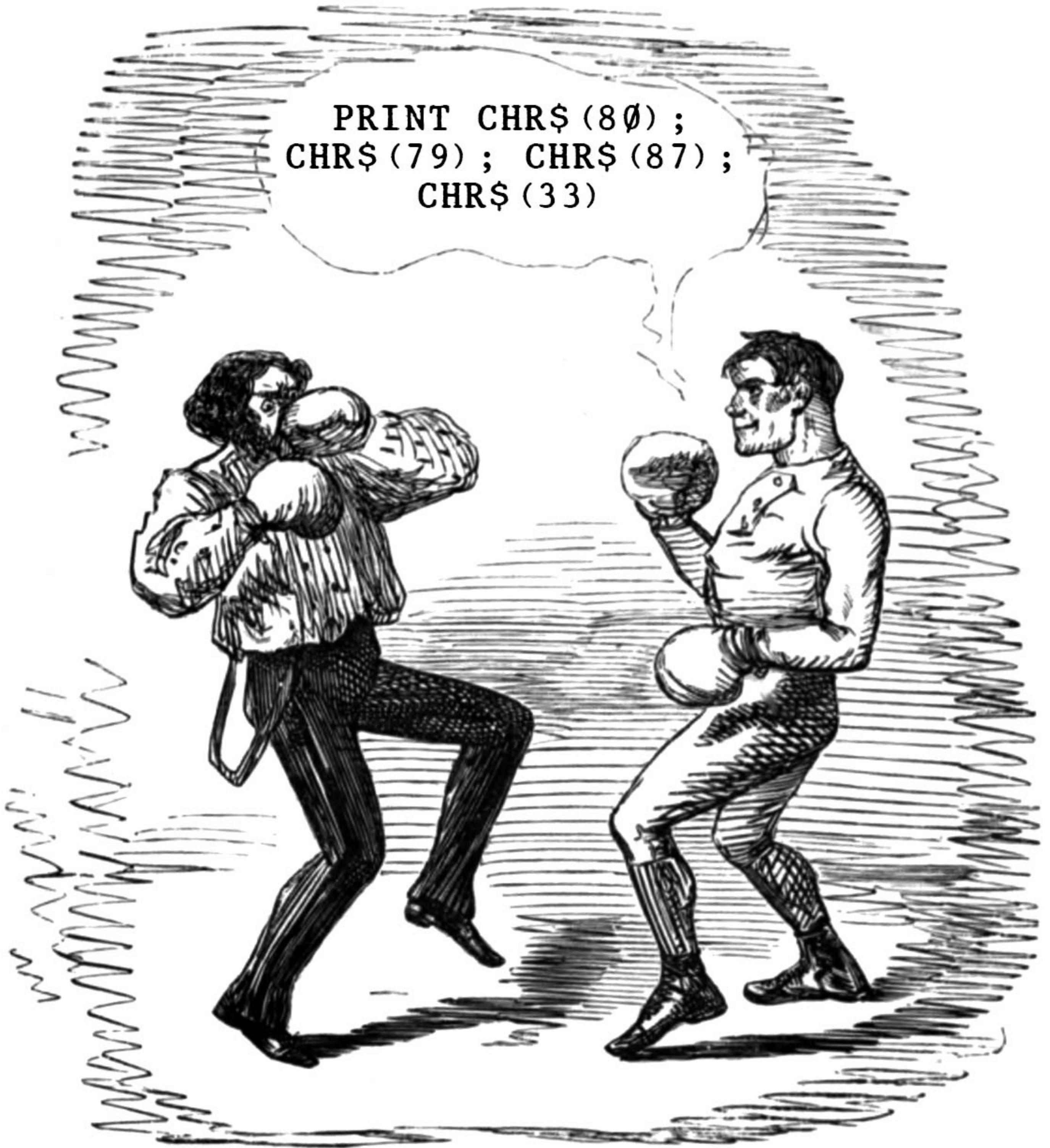
It is not possible through software to detect the presence of these boards, nor prevent them from saving an image of memory onto a disk. For this reason, they are particularly effective with programs that are totally loaded into memory and require no additional disk accesses. The only good defense is to never have the entire program in memory at one time. While this could create some difficulties such as decreased performance for particular programs, it is nevertheless necessary for single program products. Modular software requiring constant disk access may already provide sufficient protection.

FRONT DOOR METHOD

The process by which a disk is loaded into memory is well defined for normal disks. Certain facts remain true of protected disks regardless of the

method employed. First the disk must contain at least one sector (Track 0, Sector 0) which can be read by the program in the PROM on the disk controller card. Second the code that reads the protected disk must be on the disk. This means that it is possible to trace the boot process by disassembling the code involved in each step of that process. While this can be a formidable task, it is nevertheless theoretically possible to break all protection schemes with this method. The main defense against use of this method is to make it require a great deal of time to accomplish. This could primarily be done in several ways.

One way is to write the code in separate modules or layers. Each layer typically decodes the next layer and recodes the previous layer. It is also vital to verify critical layers to ensure thay have not been patched. A second way is to use an interpreted language which introduces an additional level of obscurity and a considerable amount of additional code. Neither of these can be entirely effective, but are important nevertheless.


PRINT CHR$(8Ø);
CHR$(79); CHR$(87);
CHR$(33)

Appendix C
Nibblizing

This appendix covers in great detail the encoding of data (nibblizing) on the Disk II family of drives (Disk II, IIe, and IIc). Some of this discussion may relate in a general way to encoding techniques on other computers made by Apple. But the details relate specifically to ProDOS and its device driver for a Disk II (or equivalent).

Before starting an explanations of encoding, it is fair to ask why data must be encoded at all? It seems reasonable that the data could simply be written to the disk as it is without any encoding. The reason this can't be done involves the hardware itself. Apple's design of the original Disk II was innovative and used a unique method of recording the data. While this allowed Apple to produce an excellent product, it did require some additional work to be done in software. It is not possible to read all 256 possible byte values from a diskette. This was clearly not an insurmountable problem, but it did require that the data stored on disk be restricted to bytes with certain characteristics.

ENCODING TECHNIQUES

Three different techniques have been used. The first one, which is currently used in Address Fields, involves writing a data byte as two **disk bytes**, one containing the odd bits, and the other containing the even bits. This method is often referred to as "**4 and 4**" encoding, depicting the fact that an 8-bit value is split into two 4-bit pieces. It requires two disk bytes for each byte of data, thus 512 disk bytes would be needed for each 256-byte sector of data. Had this technique been used for sector data, no more than 10 sectors would have fit on a track. This amounts to about 88K of data per diskette, typical for 5 ¼ inch single sided, single density drives.

Fortunately, other techniques for writing data to diskettes were devised that allowed more sectors per track. The earliest technique involved 13 sectors per track. This initial method involved a "**5 and 3**" split of the data bits, versus the "4 and 4" mentioned earlier. Each byte written to the disk contains five valid bits rather than four. This required 410 disk bytes to store a 256-byte sector.

Currently, of course, ProDOS features 16 sectors per track and uses a "**6 and 2**" split of data bits thereby requiring 342 disk bytes per 256-byte sector. This allows 140K of data per diskette.

The two different encoding techniques ("4 and 4" and "6 and 2") will now be covered in some detail. The hardware (in order to insure the integrity of the data) imposes a number of restrictions upon how data can be stored and retrieved. It requires that a disk byte have the **high bit set** (the first bit is a "1"), and in addition, it can have **no more than two consecutive zero bits**. Further, each byte can have at most **one pair of consecutive zero bits**.

"4 AND 4" ENCODING

The odd-even "4 and 4" technique meets these requirement—each data byte is represented as two bytes, one containing the even data bits and the other the odd data bits, (shifted one bit right). Figure C.1 illustrates this transformation. It should be noted that the unused bits are all set to "1" to guarantee meeting the two requirements.

Figure C.1: "4 and 4" Encoding Technique

No matter what value the original data byte has, this technique insures that the high bit is set and that there cannot be two consecutive zero bits. The "4 and 4" technique is used to store the information (volume, track, sector, checksum) contained in the Address Field. It is quite easy to decode the data, since the byte with the odd bits is simply shifted left and logically ANDed with the byte containing the even bits. This is illustrated in Figure C.2.

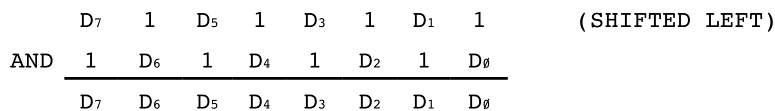


Figure C.2: "4 and 4" Decoding Technique

It is important that the least significant bit is a 1 when the odd-bits byte is left shifted. The entire operation is carried out in the device driver for the Disk II.

"6 AND 2" ENCODING

The major difficulty with the above technique is that it takes up a lot of room on the track. Since each disk byte actually contains only four bits of real data, half the bits are wasted. To overcome this deficiency, the "6 and 2" encoding technique was developed. It is so named because, instead of splitting the bytes in half as in the "4 and 4" technique, they are split "6 and 2". The two bits split off from each byte are grouped together to form additional 6-bit bytes (They are stored in an area called the Auxiliary Data Buffer.) This means that only two bits are lost in each disk byte. The 6-bit bytes used take the form XXXXXX00 and have values from $00 to $FC, each being a multiple of four, for a total of 64 different values. Figure C.3 shows the 6-bit bytes.

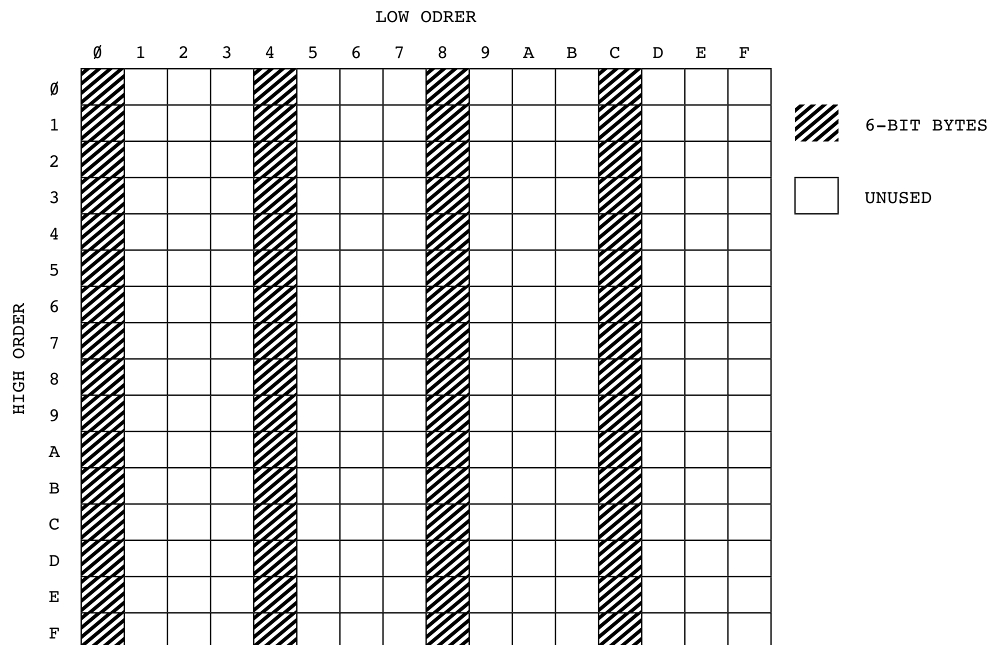


Figure C.3: Valid 6-Bit Bytes

It was necessary to map these 64 6-bit bytes into disk bytes so that they can be stored on the disk. However, there are 72 different bytes ranging in value from $95 up to $FF that meet the requirements for valid disk bytes (i.e. the high bit set and one pair of consecutive zero bits at most). After removing the two reserved bytes, $AA and $D5, 70 disk bytes remain, and only 64 are needed. An additional requirement was introduced to force the mapping to be one to one, namely, that there must be at least two adjacent bits set, excluding

bit 7. This produces exactly 64 valid disk bytes. A table of valid (and invalid) disk bytes is presented in Figure C.4

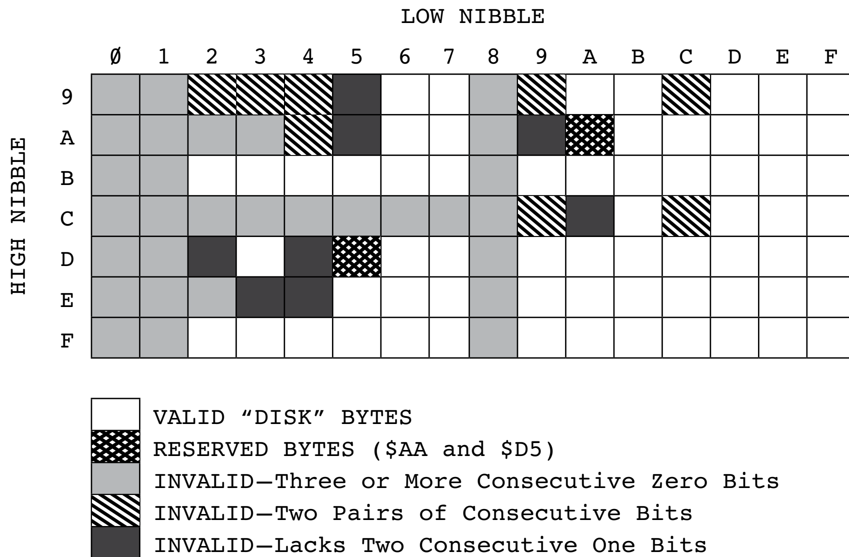


Figure C.4 Valid "Disk Bytes"

The process of converting 8-bit data bytes to disk bytes is a fairly involved process. It has three separate components, two of which we have already mentioned. We will now detail the entire operation required to convert 256 bytes of data into data suitable for diskette storage. An overview of the process is diagrammed in Figure C.5.

THE ENCODING PROCESS

First, the 256 bytes that will make up a sector must be **converted** to 342 6-bit bytes. The number 342 results from finding the total number of bits (256 x 8 = 2048) and dividing by the number of bits per byte (2048/ 6 = 341.33). Four of the bits are not used. This operation is done by the "prenibble" routine in the Disk II device driver. The code that performs this operation is fairly involved, as it requires a good deal of bit rearrangement. The results of the operation can however be easily illustrated. Figure C.6 shows how the Auxiliary Data Buffer is formed. The 256-byte User Data Page (containing 8-bit bytes), is passed to the Disk II device driver by ProDOS. Two bits are taken from each byte and put into the Auxiliary Data Buffer. The bits are rearranged slightly during this process. The two bits from each byte are reversed and the order in which they are stored in the Auxiliary Data Buffer is also reversed. The way in which these bits are rearranged and then stored is arbitrary—it could

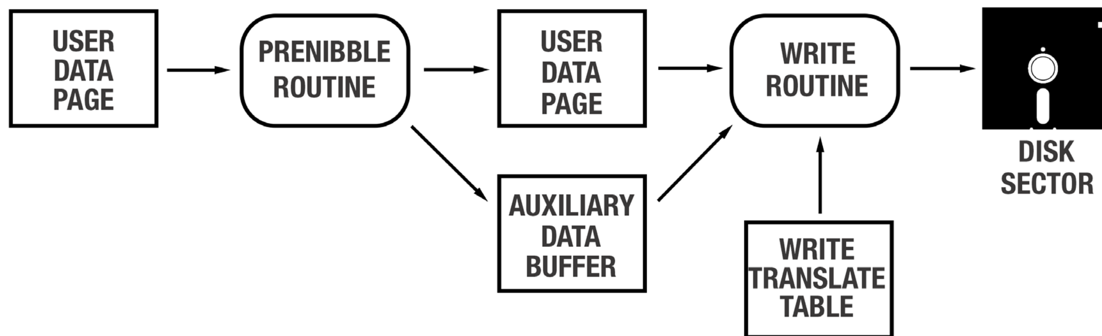


Figure C.5a Writing to the Diskette

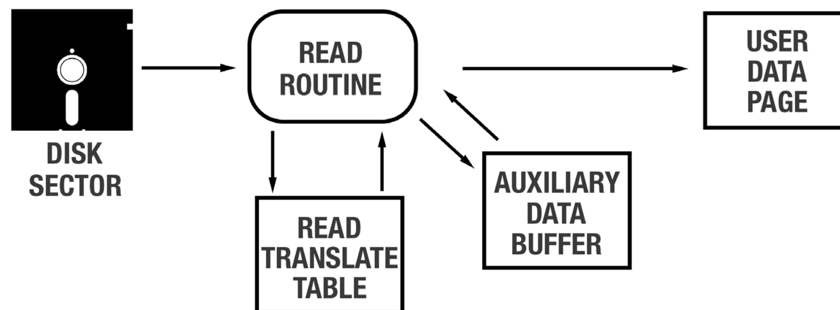


Figure C.5b Reading from the Diskette

have been done differently. The method chosen can be executed rapidly with a small amount of code. The 256-byte User Data Page is in fact unchanged as the bits are copied rather than removed, these bits are ignored (stripped out) when the data is written to the disk. This double usage of the User Data Page eliminates the need for an additional buffer. The Auxiliary Data Buffer contains four areas, one unused and the other three containing segments of the last two bits of the User buffer as is graphically illustrated.

The result of the first step is 342 6-bit bytes. The next step is that of creating a simple checksum that will be used to verify the integrity of the data. Like the Address Field, it also involves **exclusive-ORing** the information, but, due to time constraints during reading bytes, it is implemented differently. The entire block of data is exclusive-ORed with itself offset by one byte. This adds one byte, bringing the block of data to 343 bytes. This process is reversible and, while it cannot aid in recovering damaged data, it does provide a reasonable check on whether the data has been read correctly. The operation of exclusive-ORing the data block with itself is carried out a pair of bytes at a time. This enables the process to be carried out on the fly, that is, while the data is being

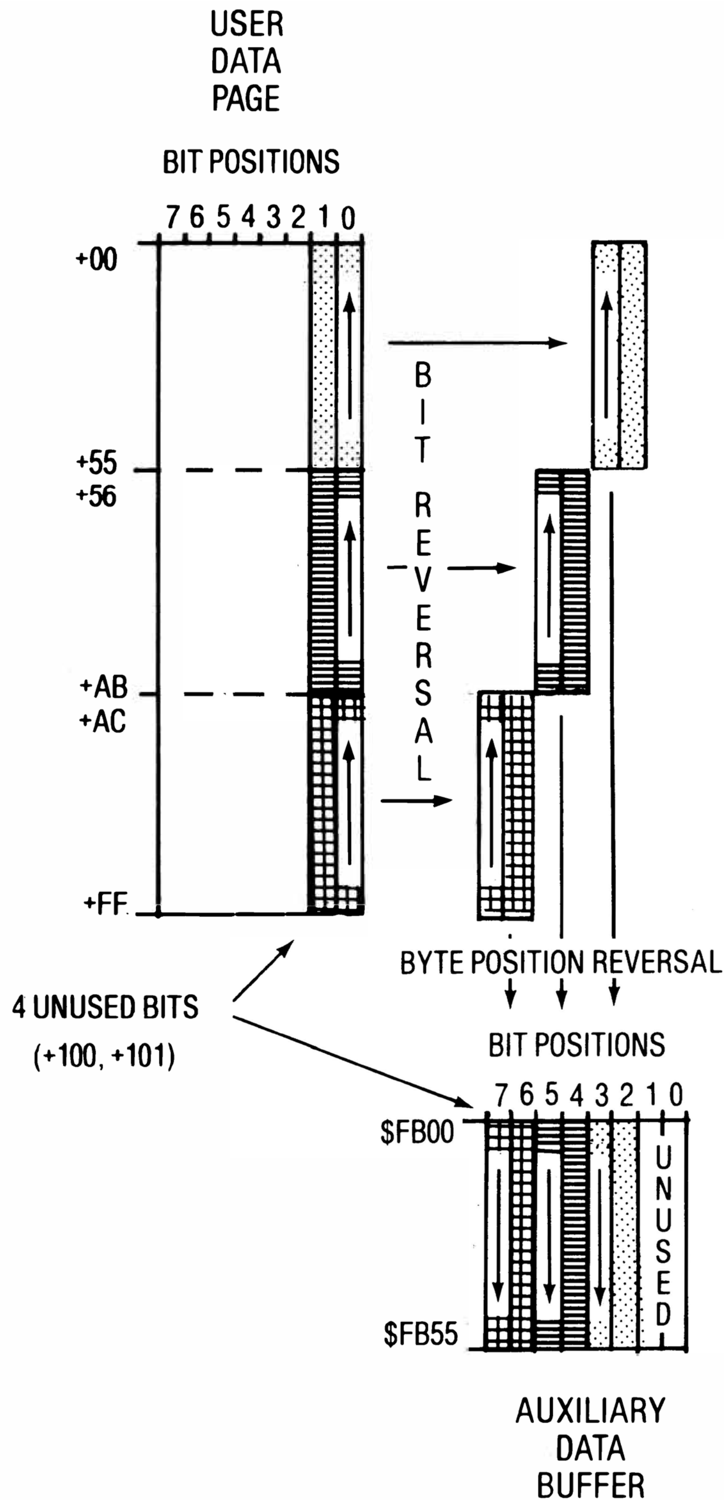


Figure C.6a Forming the Auxiliary Data Buffer

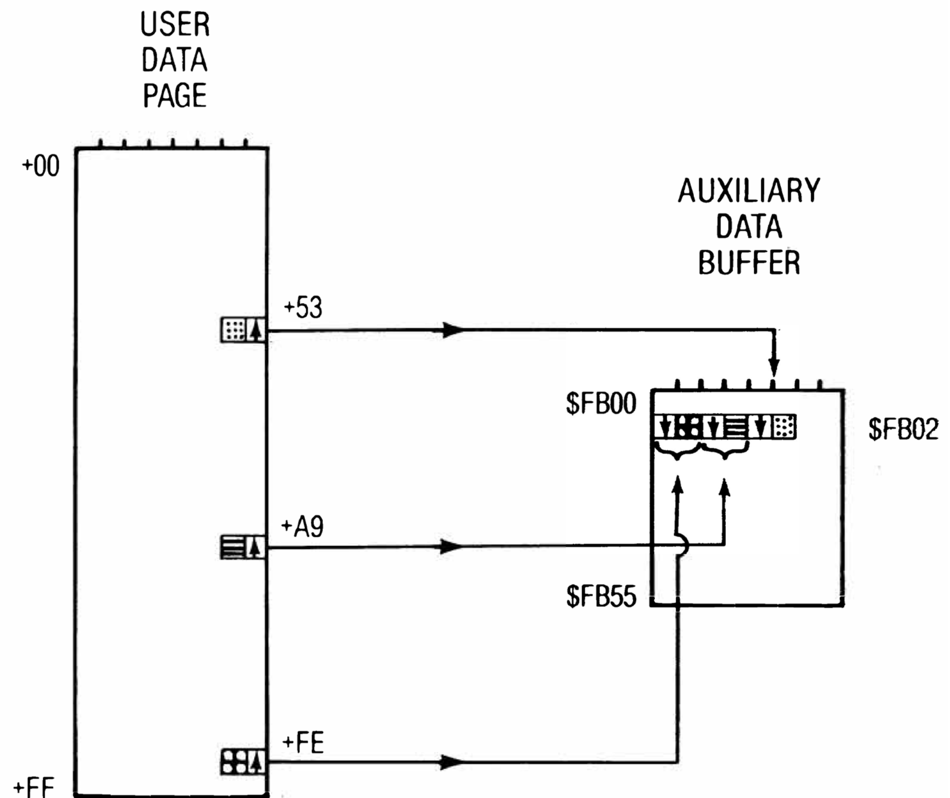


Figure C.6b Forming One Byte of the Auxiliary Data Buffer

read or written. This step and the next are actually done together and are depicted in Figure C.7.

The last step is to **translate** these 343 6-bit bytes to 8-bit disk bytes. This operation is performed using a data table in the Disk II device driver. Figure C.8 shows the mapping of 6-bit bytes to disk bytes in greater detail. Three bytes are highlighted to graphically show how the translation is made. We see for example the $00 becomes $96, $8C becomes $D3, and $FC becomes $FF.

A tabular representation of the same mapping is shown in Figure C.9. It should be noted that this is in fact a **two way mapping**. When bytes are read from the disk they are converted back to 6-bit bytes using this same table.

The reason for this transformation can be better understood by examining how the information is retrieved from the disk. The read routine must read a byte, transform it, and store it—all in under 32 cycles (the time taken to write a byte) or the information will be lost. By using the checksum computation to decode data, the transformation shown in Figure C.10 greatly facilitates the time constraint. As the data is being read from a sector, the accumulator contains the cumulative result of all previous bytes, exclusive-ORed together.

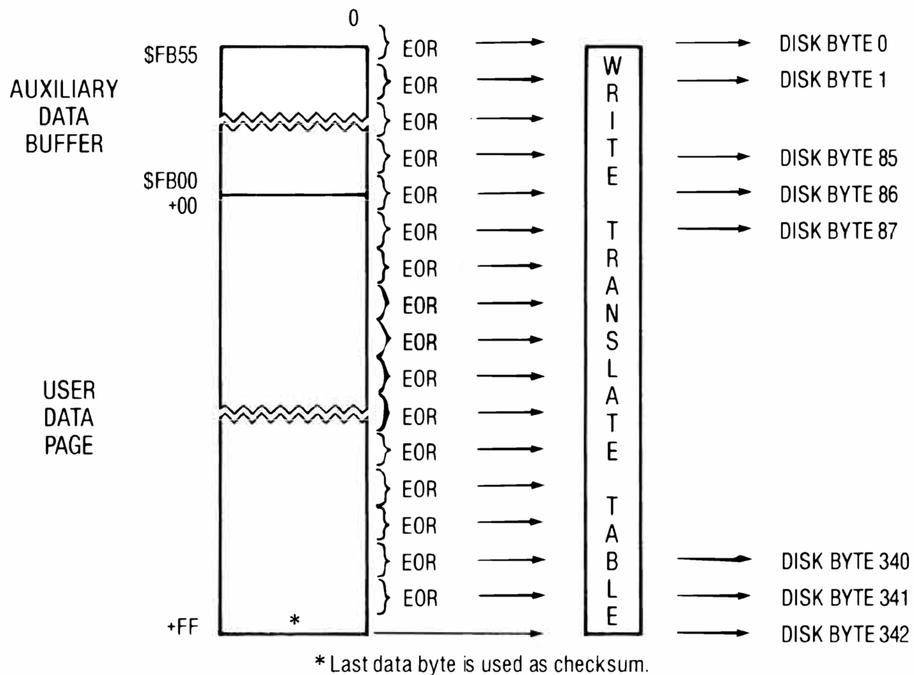


Figure C.7 Writing from Buffers to Disk

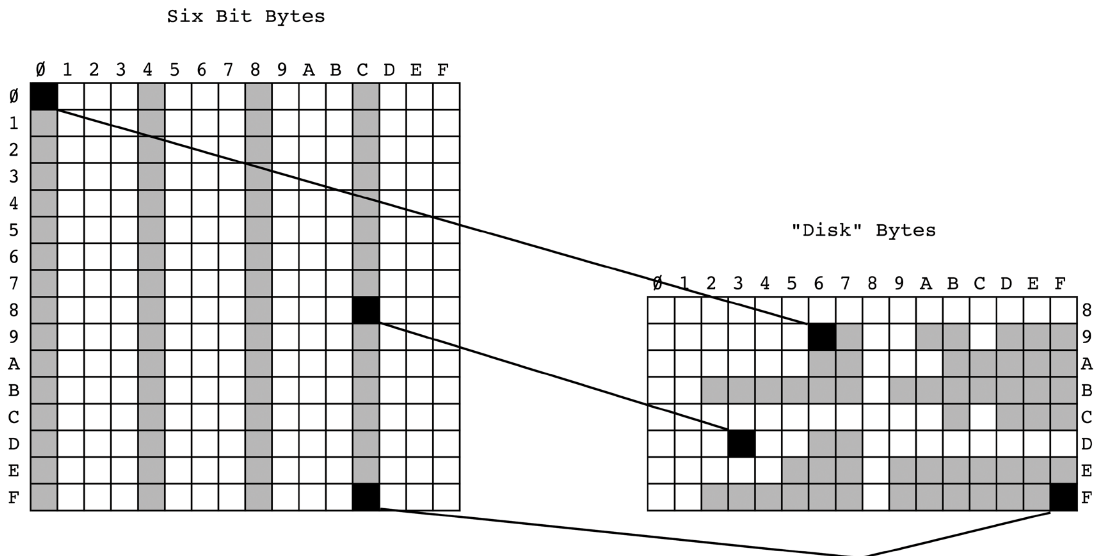


Figure C.8 Relationship of 6-Bit Bytes to Disk Bytes

The value of the accumulator after any exclusive-OR operation is the actual data byte for that point in the series. This process is diagrammed in Figure C.10.

ØØ = 96	1Ø = B4	2Ø = D6	3Ø = ED
Ø1 = 97	11 = B5	21 = D7	31 = EE
Ø2 = 9A	12 = B6	22 = D9	32 = EF
Ø3 = 9B	13 = B7	23 = DA	33 = F2
Ø4 = 9D	14 = B9	24 = DB	34 = F3
Ø5 = 9E	15 = BA	25 = DC	35 = F4
Ø6 = 9F	16 = BB	26 = DD	36 = F5
Ø7 = A6	17 = BC	27 = DE	37 = F6
Ø8 = A7	18 = BD	28 = DF	38 = F7
Ø9 = AB	19 = BE	29 = E5	39 = F9
ØA = AC	1A = BF	2A = E6	3A = FA
ØB = AD	1B = CB	2B = E7	3B = FB
ØC = AE	1C = CD	2C = E9	3C = FC
ØD = AF	1D = CE	2D = EA	3D = FD
ØE = B2	1E = CF	2E = EB	3E = FE
ØF = B3	1F = D3	2F = EC	3F = FF

AA } Reserved Bytes
D5

Figure C.9 "6 and 2" Write Translate Table

While containing specific information, the preceding discussion might still be viewed as somewhat of a theoretical presentation. The follow section will show each stage of the transformation that takes place as 256 bytes of data are prepared prior to being written to disk. The data chosen is real data that exists on the ProDOS System disk which will enable the reader to verify the following transformation.

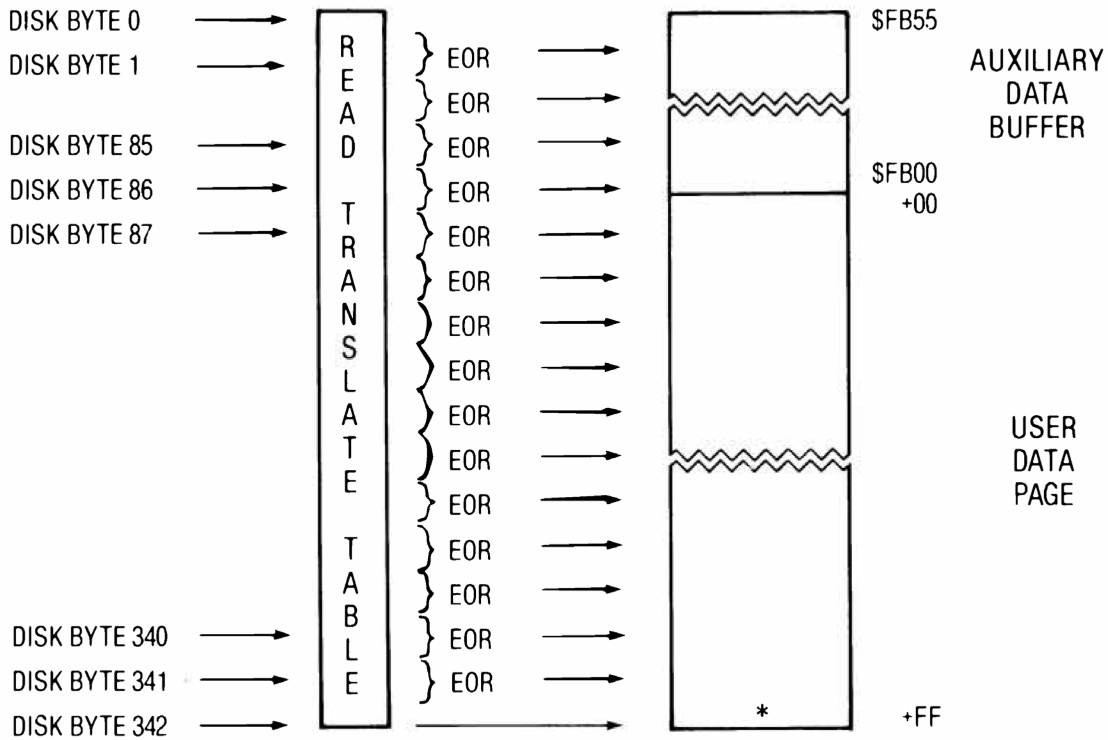


Figure C.10 Reading from Disk into the Buffers

STAGE 1

The first stage consists of creating an auxiliary buffer thereby converting the 256 bytes of data to 342 bytes. Each byte in the auxiliary buffer is made up of bits from three different bytes of the original 256-byte data. Please note that the original 256 bytes are still unchanged. Figure C.11 illustrates the results of stage 1, highlighting several bytes to aid in following this process.

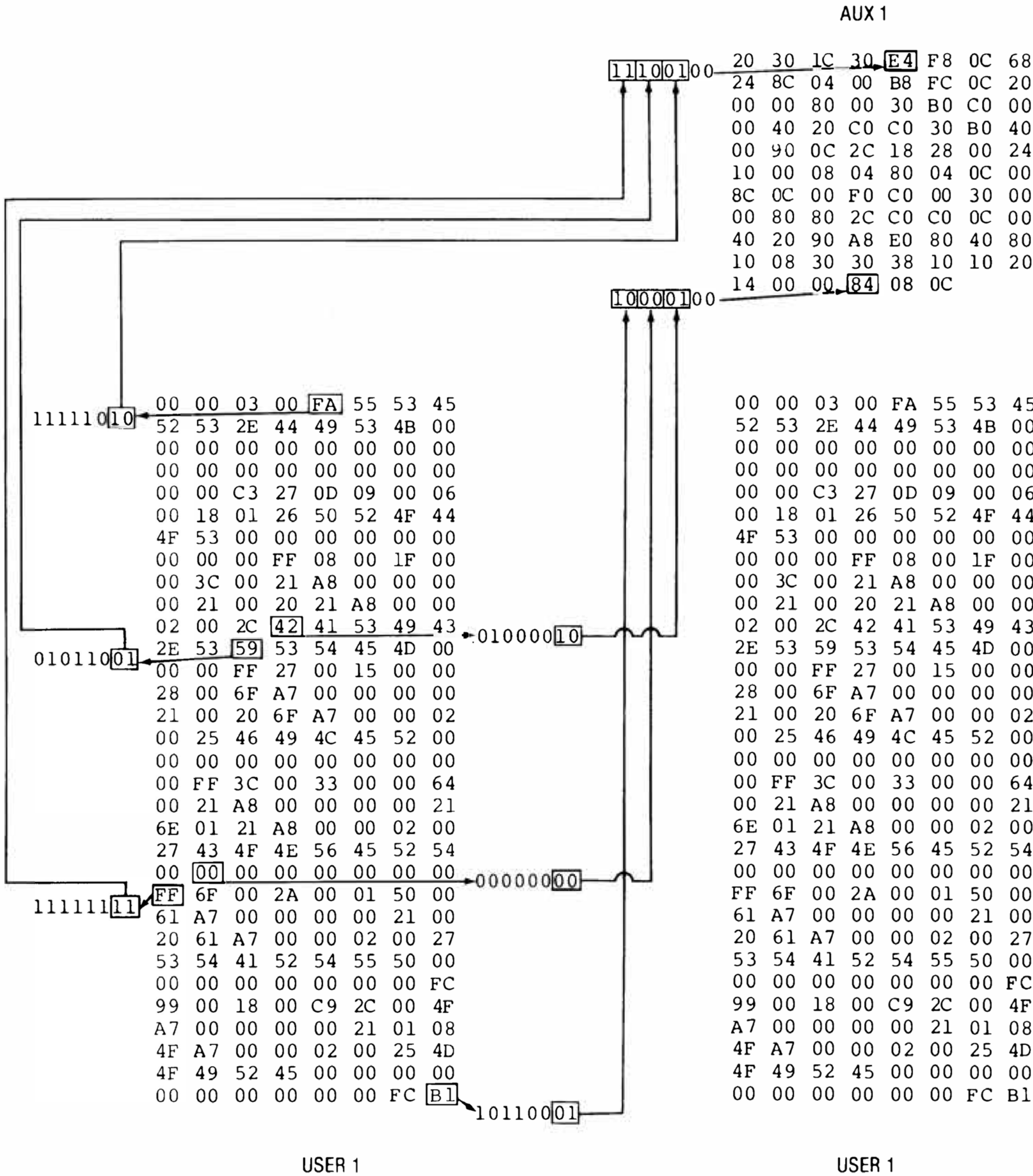


Figure C.11 Example Forming the Auxiliary Data Buffer

STAGE 2

The second stage is to create a checksum by exclusive-ORing the entire 342-byte data block with itself, offset by one byte. If it were not offset, the results would be undesirable (all zeroes). An additional byte is created in this process. While the last byte is in fact unchanged by the process and is independent of the preceding data, it serves as the checksum as seen in Figure C.12.

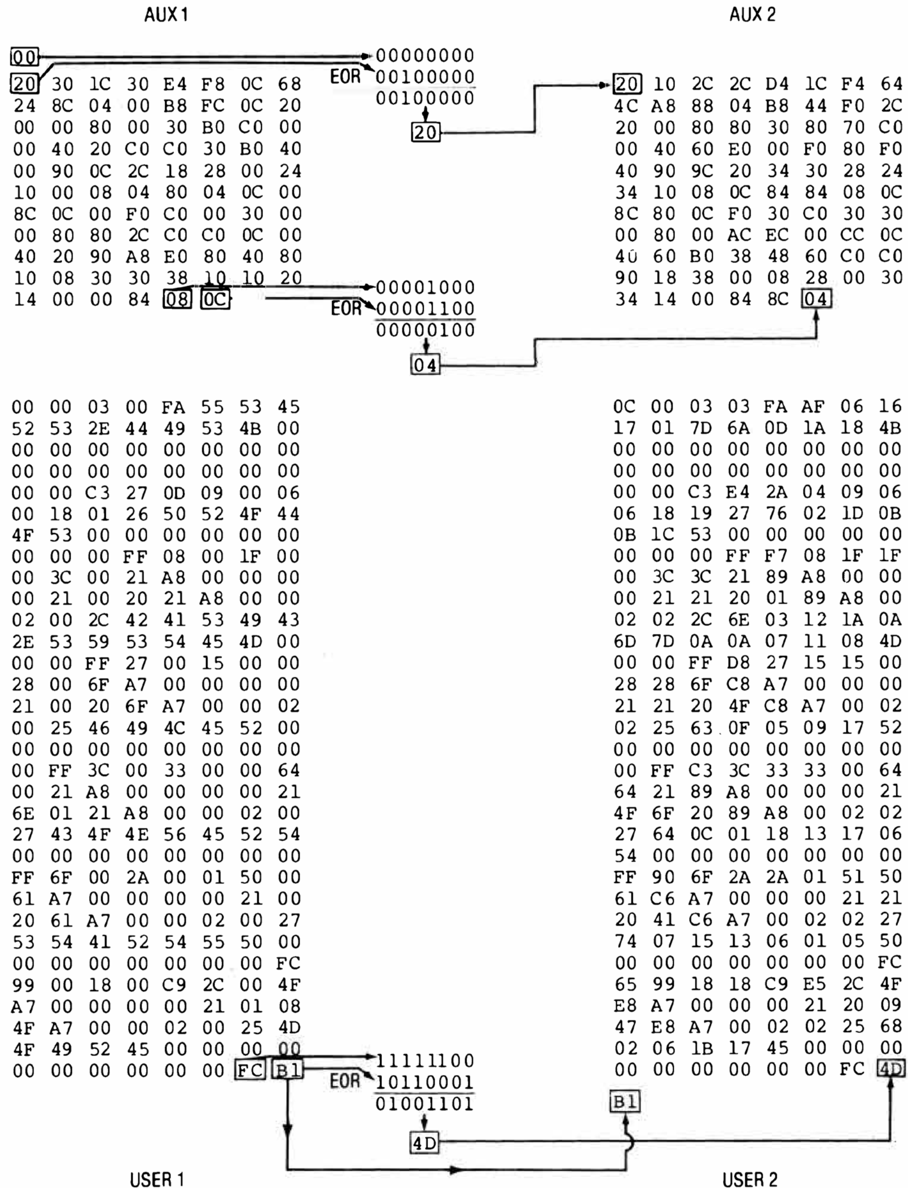


Figure C.12 Example: The Exclusive-ORing Operation

STAGE 3

The third and last stage is to translate the 343 6-bit bytes into disk bytes. This is done with a simple lookup table as shown in Figure C.13. Please note that during this step the last two bits are removed from all bytes before using the table.

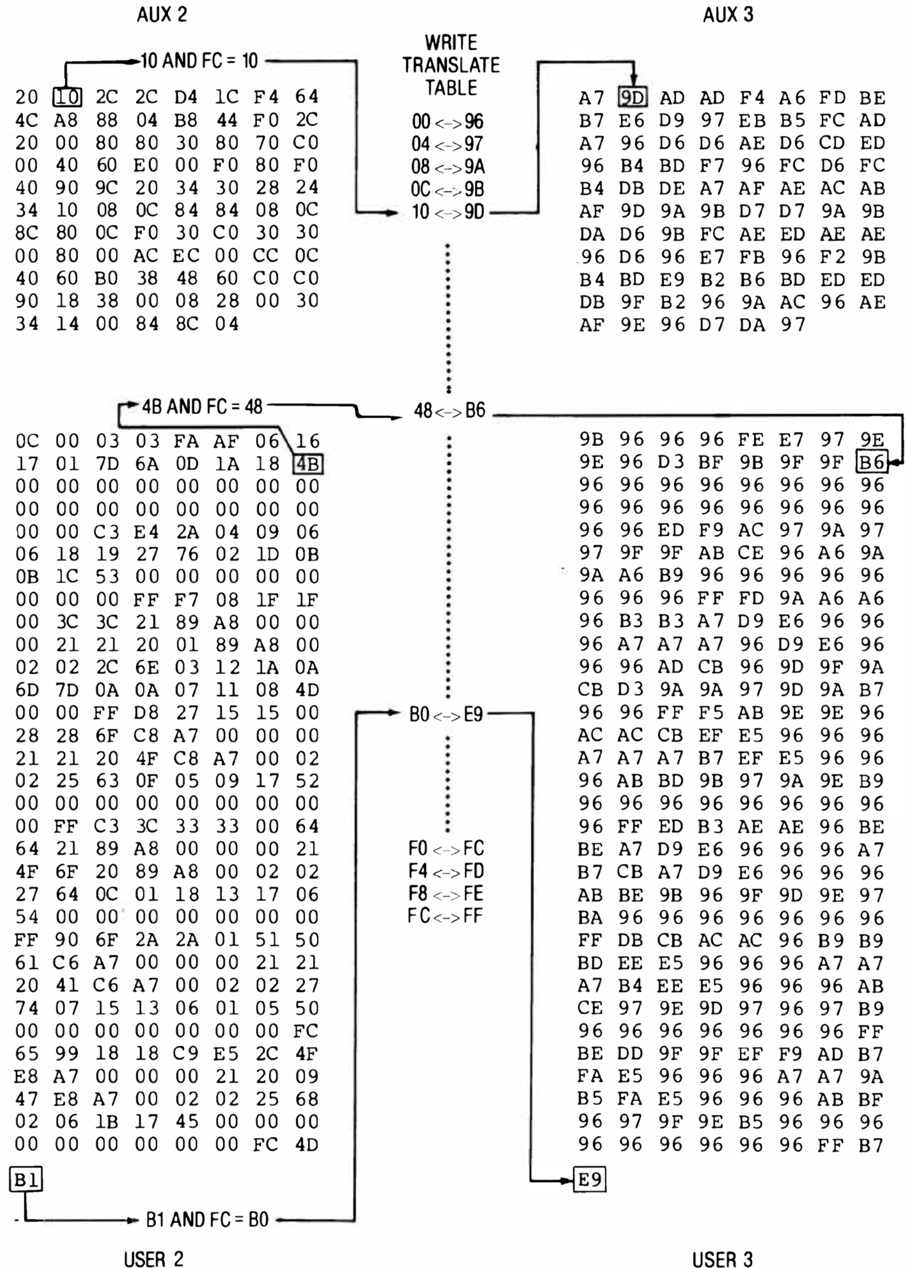


Figure C.13 Example: Translation, the Final Step Before Writing

That's a mighty fine user interface
you got there...

Appendix D

The Logic State Sequencer

Because there is such a close relationship between the disk hardware and the software that controls it, it seems appropriate to examine the firmware that directly responds to the software, that is, the Logic State Sequencer ROM. The code on this ROM actually controls the reading and writing of bits. While the information presented here should enable one to understand the process involved, it is nevertheless intended to be an overview and not a complete analysis.

The Disk II family of drives uses a unique method of storing data on a disk. They use a method named GCR (Group Code Recording) unlike most current disk drives that use FM (Frequency Modulation) or MFM (Modified Frequency Modulation). This enables writing data bytes without the use of clock bits and thereby greatly increases the amount of data that can be stored on a given track. Apple has recently put the Disk Controller Card into a Custom Integrated Circuit. Versions of the Disk Drive Controller Unit (IWM—Integrated Woz/Wendell Machine) are now used on the Apple IIe and the Macintosh. The following discussion is based on the original controller card, but should apply functionally to the new chip as well.

LOGIC STATE SEQUENCER ROM

The Logic State Sequencer is a 256-byte ROM on the disk controller card. The "program" stored there controls the data register, providing the actual means of reading and writing bits. The program on the ROM is unlike traditional software such as BASIC or machine language—it is a simple language with only six different functions or commands available. What makes it different and difficult to follow is how the flow of the program is determined. Traditional languages typically execute instructions in sequence until they encounter a control statement (such as GOTO or GOSUB) that indicates a new location. In the state machine, each byte is both a command (operating on the data register) and a control statement. What is unique is that the location of the next command is only partially determined by the control statement.

The program flow is additionally controlled by four external inputs, two provided by software and two provided by hardware. The software inputs are controlled by four memory locations, $C08C through $C08F. The locations are slot dependent (adding the slot number times 16 to the base address gives the appropriate address). Because of the nature of the state machine (timing), this is normally done with the X-register containing the offset (i.e. the slot number times 16). The two inputs provided by the hardware are the presence or absence of a read pulse and the status of the high bit of the data register.

Each of the 256 bytes in the ROM is an available location that can be accessed with the appropriate control statements. Eight bits are needed to indicate all of the locations. Four of these bits are provided by each byte in the ROM and the remaining four bits are provided by the external inputs described earlier. The four bits in the control statement contained in each byte of the ROM indicate what will be called for the next "sequence," and the four bits from the external inputs indicate what will be called for the next "state." Figure D.1 depicts the ROM as a two dimensional array, with "sequence" and "state" each providing one dimension of the address of a given element.

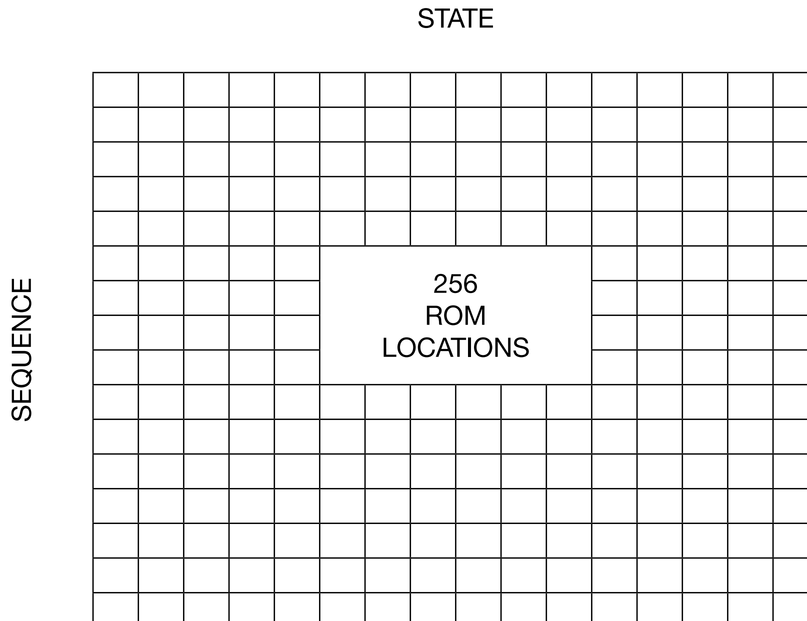


Figure D.1 Sequencer ROM is Addressed by a 4-Bit Input (STATE) and a 4-Bit Control Statement (SEQUENCE)

The 16 sequences are simply the hex numbers 0 through F, and are supplied by the high order nibble of each byte in the ROM. The low order nibble is the command number. For example, the byte "18" would execute command

number 8 (no operation) and proceed to sequence 1. Each byte or instruction takes two cycles to execute, but the state machine is running twice as fast as the 6502, so only one 6502 cycle per state machine instruction is required. The six available commands that control the data register are listed in Table D.1.

CODE	OPERATION	DATA REGISTER BEFORE	DATA REGISTER AFTER
0	Clear	XXXXXXXX*	ØØØØØØØØ
8	No operation	ABCDEFGH	ABCDEFGH
9	Shift left (bringing in a 0)	ABCDEFGH	BCDEFGHØ
A	Shift right (WRITE protected)	ABCDEFGH	11111111
	(not Write Protected)	ABCDEFGH	ØABCDEFG
B	Load	XXXXXXXX*	YYYYYYYY*
D	Shift left (bringing in a 1)	ABCDEFGH	BCDEFGH1

*XXXXXXXX and YYYYYYYY denote valid, but different, bytes.

Table D.1: Commands Which Control The Data Register

The logic of the state machine is difficult to follow even though relatively few operations are carried out on the data register. Figure D.2 graphically illustrates the logic.

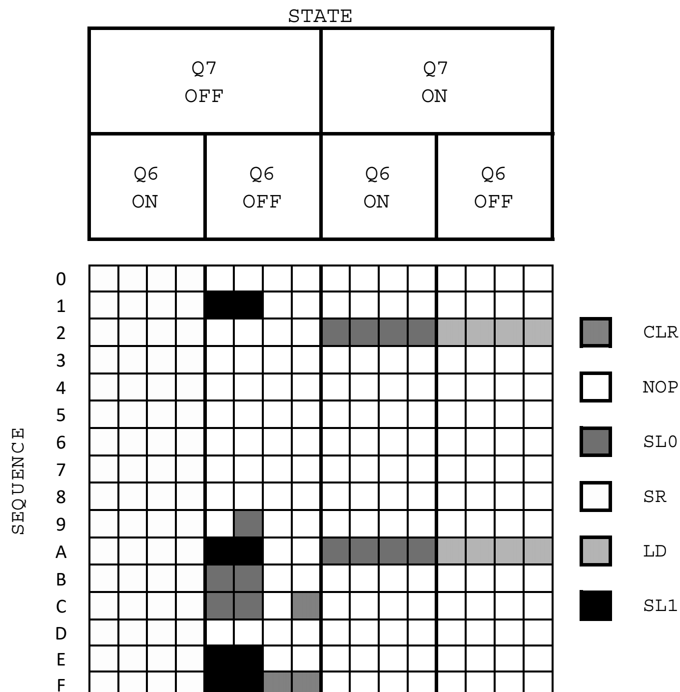


Figure D.2 Sequencer Commands

To make the task easier, the contents of the ROM will be analyzed in four parts, corresponding to the four software states. As mentioned above, the locations $C08C-C08F (plus the slot number times 16) partially controls the state machine. These four locations control two switches, Q6 and Q7. If one of these addresses is accessed, the appropriate switch will be set as indicated in Table D.2.

	SWITCH	
ADDRESS	Q6	Q7
$C08C	OFF	—
$C08D	ON	—
$C08E	—	OFF
$C08F	—	ON

Table D.2: State Switches

The first state examined will be with switch Q6 on and Q7 off. This can be described as checking the write protect switch and initializing the state machine for writing. Table D.3 lists the contents of this portion of the state machine ROM. All the instructions are identical ($0A), each shifting the data register right (command A), brining in the status of the write protect switch, and then going to sequence 0. This readies the hardware for writing since it is necessary to be in sequence 0 in order to write correctly.

	HIGH BIT CLEAR		HIGH BIT SET	
SEQUENCE	PULSE	NO PULSE	PULSE	NO PULSE
Ø	ØA	ØA	ØA	ØA
1	ØA	ØA	ØA	ØA
2	ØA	ØA	ØA	ØA
3	ØA	ØA	ØA	ØA
4	ØA	ØA	ØA	ØA
5	ØA	ØA	ØA	ØA
6	ØA	ØA	ØA	ØA
7	ØA	ØA	ØA	ØA
8	ØA	ØA	ØA	ØA
9	ØA	ØA	ØA	ØA
A	ØA	ØA	ØA	ØA
B	ØA	ØA	ØA	ØA
C	ØA	ØA	ØA	ØA
D	ØA	ØA	ØA	ØA
E	ØA	ØA	ØA	ØA
F	ØA	ØA	ØA	ØA

Table D.3: State: Q6 On And Q7 Off (Check Write Protect)

The next state examined is with switches Q6 and Q7 off (see Table D.4). This reads data from the disk, shifting in the appropriate bits as a “Pulse” or

"No Pulse" is detected by the hardware. Additionally, once the high bit of the data register is set, the data is retained until a read pulse is detected (0 bits or "No Pulses" are ignored).

SEQUENCE	HIGH BIT CLEAR		HIGH BIT SET	
	PULSE	NO PULSE	PULSE	NO PULSE
Ø	18	18	18	18
1	2D	2D	38	38
2	D8	38	Ø8	28
3	D8	48	48	48
4	D8	58	D8	58
5	D8	68	D8	68
6	D8	78	D8	78
7	D8	88	D8	88
8	D8	98	D8	98
9	D8	29	D8	A8
A	CD	BD	D8	B8
B	D9	59	D8	C8
C	D9	D9	D8	AØ
D	D8	Ø8	E8	E8
E	FD	FD	F8	F8
F	DD	4D	EØ	EØ

Table D.4: State: Q6 Off And Q7 Off (Read)

SEQUENCE	Q6 OFF HIGH BIT		Q6 ON HIGH BIT	
	CLEAR	SET	CLEAR	SET
Ø	18	18	18	18
1	28	28	28	28
2	39	39	3B	3B
3	48	48	48	48
4	58	58	58	58
5	68	68	68	68
6	78	78	78	78
7	Ø8	88	Ø8	88
8	98	98	98	98
9	A8	A8	A8	A8
A	B9	B9	BB	BB
B	C8	C8	C8	C8
C	D8	D8	D8	D8
D	E8	E8	E8	E8
E	F8	F8	F8	F8
F	88	Ø8	88	Ø8

Table D.5: State: Q7 On (Write)

When switch Q7 is turned on (write mode), the presence or absence of a read pulse is ignored. For this reason, the ROM contains two identical 64-byte

sections. Therefore, Table D.5 is presented in slightly different format. Only two operations are carried out, loading the data register from the data bus, and shifting the data out one bit at time, so that it can be written to the disk. Note that only sequence 2 and A carry out any action on the data register.

STEP	DATA REGISTER	READ PULSE	REFER TO TABLE D.4			NEXT SEQUENCE	ACTION**
			COLUMN	SEQUENCE	BYTE		
1	11Ø1Ø1Ø1	NO	4	2	28	2	NOP
2	11Ø1Ø1Ø1	YES	3	2	Ø8	Ø	NOP
3	11Ø1Ø1Ø1	NO	4	Ø	18	1	NOP
4	11Ø1Ø1Ø1	NO	4	1	38	3	NOP
5	11Ø1Ø1Ø1	NO	4	3	48	4	NOP
6	11Ø1Ø1Ø1	NO	4	4	58	5	NOP
7	11Ø1Ø1Ø1	NO	4	5	68	6	NOP
8	11Ø1Ø1Ø1	NO	4	6	78	7	NOP
9	11Ø1Ø1Ø1	NO	4	7	88	8	NOP
1Ø	11Ø1Ø1Ø1	NO*	4	8	98	9	NOP
11	11Ø1Ø1Ø1	NO	4	9	A8	A	NOP
12	11Ø1Ø1Ø1	NO	4	A	BB	B	NOP
13	11Ø1Ø1Ø1	NO	4	B	C8	C	NOP
14	11Ø1Ø1Ø1	NO	4	C	AØ	A	CLR
15	ØØØØØØØØ	NO	2	A	BD	B	SL1
16	ØØØØØØØ1	NO	2	B	59	5	SLØ
17	ØØØØØØ1Ø	NO	2	5	68	6	NOP
18	ØØØØØØ1Ø	YES	1	6	D8	D	NOP
19	ØØØØØØ1Ø	NO	2	D	Ø8	Ø	NOP
2Ø	ØØØØØØ1Ø	NO	2	Ø	18	1	NOP
21	ØØØØØØ1Ø	NO	2	1	2D	2	SL1
22	ØØØØØ1Ø1	NO	2	2	38	3	NOP
23	ØØØØØ1Ø1	NO	2	3	48	4	NOP
24	ØØØØØ1Ø1	NO	2	4	58	5	NOP
25	ØØØØØ1Ø1	NO	2	5	68	6	NOP
26	ØØØØØ1Ø1	NO*	2	6	78	7	NOP
27	ØØØØØ1Ø1	NO	2	7	88	8	NOP
28	ØØØØØ1Ø1	NO	2	8	98	9	NOP
29	ØØØØØ1Ø1	NO	2	9	29	2	SLØ
3Ø	ØØØØ1Ø1Ø	NO	2	2	38	3	NOP
31	ØØØØ1Ø1Ø	NO	2	3	38	4	NOP
32	ØØØØ1Ø1Ø	NO	2	4	58	5	NOP
33	ØØØØ1Ø1Ø	NO	2	5	68	6	NOP
34	ØØØØ1Ø1Ø	YES	1	6	D8	D	NOP
35	ØØØØ1Ø1Ø	NO	2	D	Ø8	Ø	NOP
36	ØØØØ1Ø1Ø	NO	2	Ø	18	1	NOP
37	ØØØØ1Ø1Ø	NO	2	1	2D	2	SL1
38	ØØØ1Ø1Ø1	NO	2	2	38	3	NOP

*Normal time to detect a read pulse (if one exists)
**See Table D.1. Notation used here is borrowed from *Understanding the Apple II* by Jim Sather.

Table D.6: A Sequence Example

This discussion should provide a general understanding of the Logic State Sequencer. For a comprehensive look at the disk hardware, an excellent source is *Understanding the Apple II* by Jim Sather, published by Quality Software.

SEQUENCER EXAMPLE

Table D.6 follows the state machine through a number of steps during the read process. It is assumed that a $D5 has just been read and is now in the data register. The state machine is executing the instruction at column 4 and sequence 2 of Table D.4 and will continue to loop until a read pulse is detected. The instruction being executed is a $28 which performs a NOP (8 = No OPeration) and remains at sequence 2. In our example, the next byte to read is an $AA (only the first 5 bits are shown in Table D.6). If the reader can understand this example, it should be possible to construct a similar table for any other read or write example. Note that the column number is controlled by the contents of the MSB of the data register and the presence or absence of a Read Pulse.

That feeling you get when
you try to convert a file
from DOS to ProDOS.

Appendix E
ProDOS, DOS, and SOS

This appendix is intended to assist the reader who is moving programs and data between ProDOS, DOS and SOS operating systems on Apple IIs and Apple IIIs. It is divided into two sections. One deals with the possible problems one might encounter moving from DOS 3.3 or DOS 3.2 to ProDOS with a particular emphasis on the differences in BASIC programming on the two systems. The other section discusses the areas in which ProDOS and SOS are alike, and explains ways in which programs may be written which will run with minimal modifications on either system.

CONVERTING FROM DOS TO PRODOS

The following is a list of potential problem areas when converting programs from DOS 3.3 or DOS 3.2 to ProDOS:

1. Apple DOS allows 30-character file names with embedded special characters and blanks. ProDOS restricts file names to 15 characters. The first must be a letter, and the rest may be letters, numbers or periods. No blanks or other special characters (other than period) may be in a ProDOS file name.
2. The following DOS commands are not supported by ProDOS: `MON`, `NOMON`, `MAXFILES`, `INT`, `FP`, and `INIT`. `MON` and `NOMON` may be entered under ProDOS, but they have no effect.
3. Under ProDOS, the `VERIFY` command does not read through a file to check it for I/O errors. It only verified the file's existence.
4. Although the `V` keyword is syntactically permitted on ProDOS file commands, it is not supported. Programs which depend upon volume numbers must be changed to use volume names instead.
5. When the `APPEND` command is used on a "sparse" random file, it will position at the `EOF` position, not the first "hole."
6. `CHAIN`ing between BASIC programs is now supported with a command rather than by `BRUN`ing a separate file.

7. The most significant bit of each byte is **off** in text files under ProDOS. It is **on** in DOS text files. For example, a blank in DOS was stored as $A0. Under ProDOS, it is stored as `$2Ø`.
8. Under DOS, many programs use statements of the form: `PRINT CHR$(13);CHR$(4);"dos command to be executed"` This will not work under ProDOS. The `CHR$(4)` must be the first item in the `PRINT` list. The `CHR$(4)` need not be the first thing on an output line, just the first thing in a `PRINT` statement.
9. DOS supports up to 16 simultaneously open files. ProDOS allows only 8.
10. Less memory is available to BASIC programmers under ProDOS. With no files open, the amount of memory available is equivalent to that available under DOS with three open files. Each open file uses 1024 bytes under ProDOS. Under DOS, only 595 bytes are used per file.
11. `HIMEM` should always be set to point to an even page boundary under ProDOS (a multiple of 256).
12. ProDOS does not support Integer BASIC programs.
13. The "`HELLO`" file name must be "`STARTUP`" on ProDOS. DOS allows the user to specify any name for the first file run.
14. All low level assembly language interfaces are drastically different under ProDOS. The MLI must be called to perform disk access wherever the DOS File Manager and RWTS were used in a program. There is no exact equivalent to RWTS in ProDOS, so programs which access the disk by track and sector must be converted to use the `READ_BLOCK` and `WRITE_BLOCK` MLI calls.

WRITING PROGRAMS FOR PRODOS AND SOS

When writing programs which are to run on either ProDOS or SOS, consider the following:

1. ProDOS and SOS volumes are identical in format. Either system can read the other's diskettes.
2. Block 1 on a ProDOS volume contains the SOS boot loader. This program is loaded instead of Block Ø when booted on an Apple III. It searches the directory for SOS.KERNEL and loads it instead of ProDOS. This means that a diskette can be constructed which will boot on either an Apple II or an Apple III.
3. SOS allows up to 16 concurrently open files in BASIC. ProDOS only allows 8.

4. SOS uses different file types than ProDOS. A ProDOS **CATALOG** on a SOS diskette will produce hex codes for file type but this is normal. Table E.1 shows all ProDOS and SOS file types currently defined.
5. SOS memory management allows programs to allocate and free segments of memory by making system calls. Under ProDOS, programs must manage memory themselves by marking pages free or in use in the System Global Page.
6. SOS system calls are, for the most part, very similar to ProDOS MLI calls. The areas in which differences occur are: ProDOS filing calls apply only to block devices (disks), but SOS filing calls apply to all devices; **GET_FILE_INFO** under SOS gives the **EOF** position of a file, whereas ProDOS's **GET_FILE_INFO** does not; SOS's **SET_MARK** and **SET_EOF** positions may be given as relative to the current position, but ProDOS requires them to be absolute.
7. SOS interrupts are prioritized and managed by device drivers; however, ProDOS interrupts are polled sequentially and are managed by interrupt handlers installed using MLI calls.

Table E.1: ProDOS and SOS File Types

HEX TYPE	PRODOS NAME	OS	MEANING
$00		both	Typeless file
$01		both	Bad blocks file
$02		SOS	PASCAL code file
$03		SOS	PASCAL text file
$04	TXT	both	ASCII text file
$05		SOS	PASCAL text file
$06	BIN	both	Binary File
$07		SOS	Font file
$08		SOS	Graphics screen file
$09		SOS	Business BASIC program file
$0A		SOS	Business BASIC data file
$0B		SOS	Word processor file
$0C		SOS	SOS System file
$0D-$0E		SOS	SOS reserved for future use
$0F	DIR	both	Directory file
$10		SOS	RPS data file
$11		SOS	RPS index file
$12-$18		SOS	SOS reserved for future use
$19	ADB	ProDOS	AppleWorks data base file
$1A	AWP	ProDOS	AppleWorks word processing file
$1B	ASP	ProDOS	AppleWorks spreadsheet file
$1C-$BF		SOS	SOS reserved for future use
$C0-$EE		ProDOS	ProDOS reserved for future use
$EF	PAS	ProDOS	ProDOS PASCAL file
$F0	CMD	ProDOS	Added command file
$F1-$F8		ProDOS	ProDOS user defined file types
$F9		ProDOS	ProDOS reserved for future use
$FA	INT	ProDOS	Integer BASIC program file
$FB	IVR	ProDOS	Integer BASIC variables file
$FC	BAS	ProDOS	Applesoft BASIC program file
$FD	VAR	ProDOS	Applesoft BASIC variables file
$FE	REL	ProDOS	EDASM relocatable object module file
$FF	SYS	ProDOS	System file

Appendix F

Differences Between the ProDOS 8 Versions

This Appendix identifies the changes to ProDOS 8 that were introduced with the 1.2 and 1.3 versions. Although we believe this to be a fairly thorough list, there may have been a few changes (especially deletions) that we didn't catch. Quite a few changes that were apparently made only to save space are not listed here.

CHANGES INTRODUCED IN THE 1.2 VERSION

The changes that were made to the 1.1.1 version of ProDOS to produce ProDOS 8, Version 1.2, are listed below. Addresses given here are Version 1.2 addresses.

RELOCATOR

1. When running on a IIGS with ProDOS 16 installed, ProDOS 8 is entered from PQUIT (the ProDOS 16 Quit Handler). In this case the Relocator is entered at $2003 instead of $2000 `[2ØØØ-2ØØ5]`.
2. The ProDOS 8 version number is now stored in the MLI subdirectory header data area `[2Ø48-2Ø4C]`.
3. Always checks to see if running on a IIGS, and if so sets a flag (2278). Also sets E100BD=0 if ProDOS 8 is the initial boot on a IIGS. `[2Ø79-268Ø]`.
4. Sets aux stack pointer to $FF `[2ØE6-2ØF5, 2111-2115]`.
5. If operating on a IIGS, skips logic that searches for slot 3 80-column card `[212A-2139]`.
6. If operating on a IIGS, installs the IIGS Clock Code `[21AD-21DØ, 22D3-22DA]`.
7. Now checks for an AppleTalk Initialization File (ATINIT file) before looking for a .SYSTEM file. If the ATINIT file is found, it is loaded and executed, then the search for a .SYSTEM file commences. `[22DB-2381]`.

8. The list of devices is now ordered differently. It recognizes the SmartPort and allows four Slot 5 SmartPort units to be accessed as Slot 5, Drives 1 and 2 and Slot 2, Drives 1 and 2. If Slot 2 is being used by a storage device, however, only the first two devices on the Slot 5 SmartPort can be accessed. It also changes the search order, making sure that Disk II devices are searched last when a device scan takes place (such as during an MLI ON_LINE call). [2668-271B, 275D-2767, 2814-28AA].
9. A bug in the /RAM driver (which we pointed out in the 1.1.1 supplement) that allowed a block read of block 7 (which doesn't exist) has been corrected [2D4B].
10. The /RAM caller, which operates in high RAM, now contains a $60 (RTS instruction) at address $FF58. Peripheral cards sometimes call that address to figure out which slot they are in, and in case they forget to set ROM for reading, the call will still work. [2E56-2E58].
11. A subroutine that sets high RAM for reading/writing was created to save space in the code [2518-251E].

MLI AND MI GLOBAL PAGE

1. The Global Page now pushes the P-Register and disables interrupts before calling the actual MLI [DEØ1-DEØ4, DE1C-DE21, BF4B-BF4F].
2. Setting the MLIACTIVE flag a little differently now allows nested calls to the MLI by interrupt routines [DEBF-DE91].
3. A new MLI command was introduced (command=$82), that allows the user to install a routine to handle unclaimed interrupts [DEFF-DFØB, FD23-FD2C].
4. If there is no unclaimed interrupt handler, ProDOS 8 now counts unclaimed interrupts, and will allow 255 of them to occur before finally issuing a fatal error. This allows a brief time for the unknown interrupt to stop interrupting. [DFB3-DFB7].
5. If operating on a IIGS and system death occurs, the NEWVIDEO softswitch is set to 0, reinitializing the IIGS video [EØ13-EØ16].
6. Processing for the ON_LINE command now frees the VCB entry for a device that was previously on-line but has been taken off-line [E28A-E2A4].
7. A subroutine that reads a block where the block number is in the A and X registers was added to save space in the code [EBC9-EBDØ].

8. The error message that results when a file being opened has an illegal storage type has been changed from "incompatible directory format" to "unsupported storage type" [EEC7-EECA].
9. An error type $C is now indicated when truncating or deleting a file and the file's storage type is illegal [FA4D-FA4E].
10. To save space in the QuitCode Caller, some in-line code was changed to a loop [FCAF-FCB7, FCD8-FCEØ].

QUIT CODE

1. Uses standard character set instead of alternate character set [1ØØ6-1ØØ8].
2. Sets normal 40-co1umn screen in a safer way, such that screen hole values are preserved [1ØØC—1Ø11].
3. Message display routine is modified [1Ø33-1Ø34, 11D6—11D7, etc.].
4. The method of displaying the current prefix is changed so that it is always written to the same screen location [1Ø4D-1Ø5C].
5. User can now backspace with the DELETE key as well as left arrow [1Ø7C-1Ø7F, 1ØFC-1ØFF].
6. The method of inputting the Application name was modified [1ØE7-1ØE9].

CLOCK CODE

1. The code for the ThunderClock includes a lookup table to determine the year based on the day of the year and the day of the week. This table is only good for a span of five or six years. The table released with Version 1.1.1 was good for the years 1982-1987. The table released with Version 1.2 covers the years 1986-1991. [D7B8-D7BE].
2. A completely separate clock routine is provided in Version 1.2 in case ProDOS 8 is operating on a IIGS. If so, the IIGS Clock Code is always enabled. It is written in 65816 and calls the ReadTimeHex tool in the tool kit to read the clock. [D742-D79Ø].

CHANGES INTRODUCED IN THE 1.3 VERSION

The changes that were made to the 1.2 version of ProDOS 8 to produce ProDOS 8, Version 1.3, are listed below. Addresses given here are Version 1.3 addresses.

RELOCATOR

1. The boot message now includes a line that says "ALL RIGHTS RESERVED." Chalk up one for the legal department! `[25F6-2671]`.
2. A ProDOS Status call now immediately precedes the SmartPort Status Call. This is because the SmartPort interface does not set up its device list until it receives a ProDOS Status call. Earler versions of ProDOS 8 may not always find all SmartPort devices. `[286E-2894]`.

MLI

1. When files are deleted, previous versions of ProDOS zero out all but the first block of discarded index blocks. Now such index blocks will not be zeroed, but the pages of these blocks will be flipped. That is, the high byte of the block numbers will be exchanged with the low byte of the block numbers. `[F992-F99A, FBC7-FBDC]`.
2. Previous versions of ProDOS forgot, in certain cases, to rewrite index blocks that were being discarded when shortening or deleting files. Now such blocks will always be rewritten to disk in a zeroed (shortened file) or flipped (deleted file) form. `[FAB8-FABC, FB91-FB93]`.
3. A poorly-written loop in the QuitCode Caller that was added for Version 1.2 was rewritten for Version 1.3. The Version 1.2 code might cause problems on a IIGS. `[FDØ5-FDØC]`.

DISK II DEVICE DRIVER

1. There is a routine in the Disk II Device Driver that clears phases in case the Disk II device is sharing transmission lines with SmartPort devices. This routine was patched in Version 1.3 so that phases are now cleared with LDA instructions instead of STA instructions. This eliminates bus fights that can, in some situations, cause unwanted writing to the floppy disk. `[D6C3-D6CE]`.

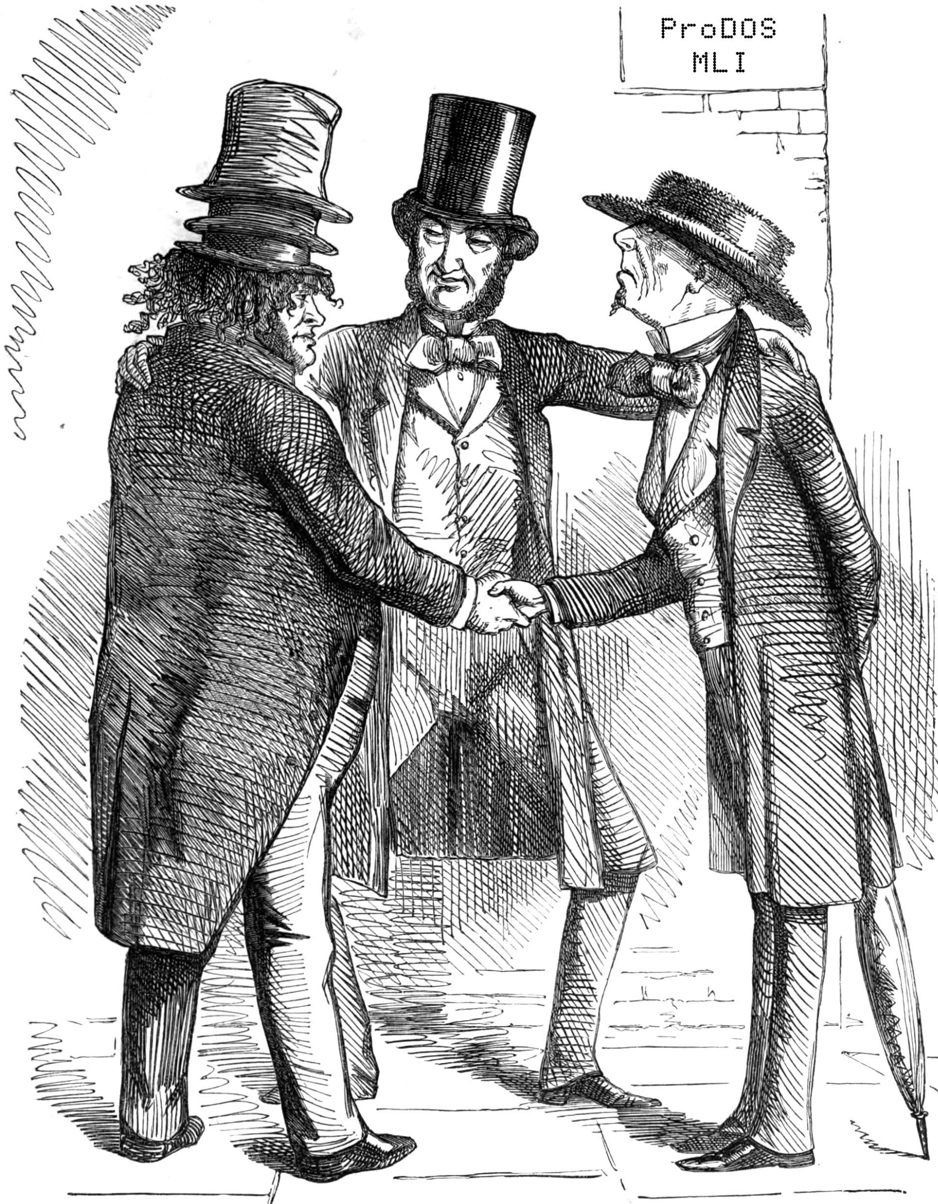
BUGS IN VERSIONS 1.2 AND 1.3

It is fair to say that both Versions 1.2 and 1.3 of ProDOS 8 are relatively bug free. Perhaps a few escaped our notice, but we know of only three minor bugs, which are as follows:

MLI, Versions 1.2 and 1.3, at EC64. This bug has been in ProDOS since day l. Although there is no easy way to correct the problem (because a three-byte instruction is needed where there are only two bytes), any serious problems can be avoided by putting NOP's at EC64 and EC65 (3D64 and 3D65 in load location). This bug can only take effect when a storage type 0 is found (not likely unless disk swapping) and a lot of files are open simultaneously.

MLI, Version 1.2 only, at FCD8. A loop that indexes around the 64K boundary may cause problems on a IIGS. Use version 1.3 or recode the loop so that the boundary crossover is eliminated.

MLI, Version 1.3 only, at FBCD. A 65C02 instruction snuck into the code, which will be disastrous when Version 1.3 is run on a computer with a 6502 processor (Apple II+, unenhanced IIe). It is easily patched: just change the byte at FBCD (4CCD in the load image) from $80 to $F0, because BEQ works fine here.



Adding New Commands to ProDOS

Miscellanea

Could you spare a few bits?

SETUP.SYSTEM

BY SEAN NOLAN

This article describes a program which lets you easily install customizing routines, such as a /RAM disk and clock card drivers, on bootup into ProDOS, It mimics, under ProDOS 8, the SYSTEM.SETUP feature of ProDOS 16. It is emphatically placed in the public domain: I urge software publishers to take it and include it where appropriate with their own programs. I only ask that you take the program unmodified. It would be unfortunate if variants of the program spread around, each subtly incompatible with the others.

One feature which makes ProDOS 16 a more powerful operating system than ProDOS 8 is its flexible and complicated startup sequence. This includes the ability to boot up into either ProDOS 8 or ProDOS 16 based applications, as well as automatically installing, on bootup, patches in /RAM to the Apple IIGS toolbox. ProDOS 16 accomplishes this by loading and executing all files found in the /SYSTEM/SYSTEM.SETUP subdirectory. These files must include TOOLSETUP, but may include other files for installing desk accessories, diagnostic and debugging routines, or anything else you can think of.

SYSTEM.SETUP is a nice idea. It allows you to specify what happens at boot up simply by placing the correct files in a particular subdirectory. Unfortunately, ProDOS 8 does not support SYSTEM.SETUP. If it did, it would be much easier to install IIe /RAM disk drivers, clock card drivers, and IIGS Classic Desk Accessories on ProDOS 8 disks.

Suppose you have a large memory card in your IIe, and you want a /RAM disk driver for it to be installed automatically every time you boot. Here are two ways you could do it:

1. If the /RAM disk driver is a system program, you could place it first on the disk so that it's automatically called on boot up. When the system program quits, you then type in the name of the next system program to run, such as BASIC.SYSTEM.

2. If the /RAM disk driver is a binary file, you can place BASIC.SYSTEM first, and have a BASIC program called STARTUP which BRUNs the /RAM disk driver. This is fine, as long as you plan to boot up into BASIC.

If you have both a large memory card and a clock card, you may need to install drivers for both of them. Now things become more difficult for you, particularly if your memory card came with a SYStem file program to install the /RAM disk, and your clock card came with a BINary file clock driver. SETUP.SYSTEM (note: *not* SYSTEM.SETUP) automates the bootup sequence for you. To use it, assemble it and make it the first SYStem program on your disk whose name ends in ".SYSTEM". This ensures that it will be run on boot up. Then create a subdirectory called SETUPS, containing the files which you want to be executed on boot up. That's all there is to it.

Many existing programs already work with SETUP.SYSTEM. I have used it with the RAM disk drivers for both Applied Engineering's and Checkmate Technologies' IIe memory cards, and with *Diversi-Cache* and *Diversi-Hack* (1165 classic desk accessories by Bill Basham).

When run, SETUP.SYSTEM will search for the SETUPS subdirectory in the volume directory of the last disk accessed. It then loads and executes, in turn, each SYStem or BINary file in the SETUPS subdirectory. These files I refer to as "setup files." When all setup files have been called, SETUP.SYSTEM then looks for the second system file in the main volume directory whose name ends in ".SYSTEM", and runs that. This second ".SYSTEM" program is your main application. It can be BASIC.SYSTEM, your assembler, word processor, *AppleWorks*, or any other ProDOS application.

SETUP.SYSTEM was especially written to be compatible with already existing /RAM disk drivers, clock drivers, and Apple IIGS Classic Desk Accessories under ProDOS 8. The rules for compatibility are simple. Setup files must be either BINary files or SYStem programs in the SETUPS subdirectory. This is in contrast to ProDOS 16's SYSTEM.SETUP, which uses new ProDOS 16 file types exclusively. You may place other file types in the SETUPS subdirectory, such as text files, but SETUP.SYSTEM will ignore them. BINary setup files may load anywhere in either memory range $200-$3EF or $800-$BEFF. They should exit with an RTS. You may also place SYStem programs in the SETUPS subdirectory. These SYStem programs. as usual, must load at $2000, and must not exceed $B8FF when loaded. SYStem setup programs should exit with a ProDOS QUIT call. Most SYStem programs do this, although there are exceptions. A few exit by looking for the next SYStem program to run by name, or look for and run the second ".SYSTEM" file on

the disk. Programs which do this are incompatible with SETUP.SYSTEM. Among the few system programs which are incompatible for this reason is the original version of the ProDOS FILER.

Once the setup program is in memory, it may use any memory except for pages $BD and $BE, which is where SETUP.SYSTEM lives in hibernation, and the ProDOS global page at $BFØØ. Because of this memory restriction, some existing SYStem programs may not be used as setup files. You should not, for instance, place BASIC.SYSTEM in the SETUPS subdirectory. BASIC.SYSTEM relocated itself to $9AØØ-$BEFF, overwriting SETUP.SYSTEM. If you do so, and then type "BYE" from BASIC, your Apple will explode into a reball, destroying all life other than insects within a half mile radius. Really.

A third reason why system programs may not work in the SETUPS subdirectory is that they may require subsidiary files—which must also be placed in the SETUPS subdirectory. If these subsidiary files are BINary or SYStem files, SETUP.SYSTEM will eventually try to call them on their own. This problem happened to me when I moved the Apple IIGS *System Utilities* into my SETUPS directory. (Why? So I could copy some files onto my /RAM disk on bootup.) The problem was solved by changing the file type of the three BINary files which the *System Utilities* needed. I gave them file type codes of $Ø3. which is a defunct Apple III Pascal Text file code. The point was, SETUP.SYSTEM cared what the file types were and the *System Utilities* didn't. I should point out that placing random system programs in your SETUPS subdirectory can be dangerous. By trial and error, I found that the GS *System Utilities* left the crucial pages $BD and $BE undamaged.

If you plan to write new setup programs, I recommend that you make them BINary files rather than SYStem programs. SETUP.SYSTEM temporarily redirects the ProDOS quit vector at $BFØ3 into itself before calling any SYStem programs in the SETUPS subdirectory. Therefore, when the setup program executes a ProDOS QUIT call, control is returned to my code. This is not kosher programming, and may not work with future revisions of ProDOS. I added the feature solely so that many already existing SYStem programs could be used with SETUP.SYSTEM.

TYPING IT IN

Although the program below was assembled using the Merlin Pro assembler, I tried to avoid as many Merlin-specific psuedo ops as possible and present a generic assembly listing. In order to assemble it on other assemblers, some minor changes will be necessary: The psuedo-op ASC in Merlin creates either

high or low ASCII characters, depending on whether the string is surrounded by a single quote (low ASCII) or a double quote (high ASCII), I used both varieties. Users of EDASM, APW, or ORCA/M should add the `MSB` psuedo-op where needed. The last byte of the program is a checksum byte. It is the exclusive-OR of all the previous bytes in the program. If you assemble the program using Merlin, be sure to check that the byte you get is the same as the one in the listing. If it isn't, you made a typo.

If you are not using Merlin, replace the last line with

```
DFB $E8
```

and run the following program to check that the code is correct:

```
1Ø PRINT CHR$(4)"BLOAD SETUP.SYSTEM,A$2ØØØ,TSYS"
2Ø FOR X = 8192 TO 87Ø3
3Ø SUM = SUM + PEEK(X)
4Ø NEXT
5Ø IF SUM = 64732 THEN PRINT "OK" : END
6Ø PRINT "ERROR"
```

PROGRAM LOGIC

`SETUP.SYSTEM` is not a model of good programming practice. In order to get all the functionality I wanted into 512 bytes, I had to resort to self-modifying code. If the program went one byte over, to 513 bytes, the storage space on disk would mushroom from one block to three.

Among the first tasks of `SETUP.SYSTEM` is to relocate itself from it's load address of `$2ØØØ` to it's running address of `$BDØØ`. `SETUP.SYSTEM` must move itself out of the way so that it can load other system programs without overwriting itself. Lines 48 through 54 in the program listing accomplish this. Next, the program (lines 61 through 72) tries to determine the name of the disk from which `SETUP.SYSTEM` was run. This name is saved so that the program can set the ProDOS prefix to the volume directory at a later time. The quit vector in the ProDOS global page is then saved (lines 73-76) so that it can be restored on exit.

Control returns to line 78 at the label `MainLoop`, before and after executing each of the setup files. The reset vector and the quit vector are both redirected to point to `MainLoop`, to increase the chances of `SETUP.SYSTEM` regaining control after calling setup programs. The system bitmap is then cleared so that we can load the setup file (lines 87-95), and the machine is then put into as normal a state as possible (lines 96-103).

Lines 106 through 127 make sure that the original volume is in the disk drive, and sets the ProDOS prefix to that volume. If the disk isn't around, a warning message is displayed, and the user is given a chance to insert the correct disk.

Line 129 calls a subroutine called `NextFile`, which reads the `SETUPS` subdirectory until it finds the name of the next `BIN`ary or `SYS`tem program. It leaves that name in a known place. If it finds no more `SYS`tem or `BIN`ary files it exits with the carry set. This will happen when all of the setup files have been loaded and executed—that case will be dealt with later.

If `NextFile` found the name of a setup file, we proceed to line 132. Here the prefix is set to the `SETUPS` subdirectory, and the setup file is loaded into memory by calling the subroutine `ReadFile`, which starts at line 240. If there was no error when loading it, the setup file in memory is called. One way or another, control returns to `MainLoop`, either through the ProDOS `QUIT` call, the user pressing RESET, or the setup file exiting with an `RTS`.

Once all the setup files have been loaded, we go to line 139 and restore the original ProDOS quit vector. Lines 145 through 149 then modify the `NextFile` routine so that it works rather differently. Where before `NextFile` looked for either `BIN`ary or `SYS`tem files, now it will look for only `SYS`tem ones. `NextFile` used to look in the `SETUPS` subdirectory. It now looks in the volume directory. `NextFile` will now be used to find the name of the system program to exit to.

`NextFile` is called repeatedly, returning the names of all `SYS`tem programs in the volume directory (line 150). Lines 152-159 make sure that the filename ends in "`.SYSTEM`". If it passes this test, we then check that it is not the first "`.SYSTEM`" file on the disk (lines 160-162). The first "`.SYSTEM`" file on the disk is presumably "`SETUP.SYSTEM`", and we don't want to run that, or we'd go into an endless loop. The system file is loaded, and called (line 163) with the same subroutine, `ReadFile`, which was used to load and call setup files. The difference is that this time we never come back, since the ProDOS quit vector has been restored to its original value.

```
                1    *
                2    * SETUP.SYSTEM by Sean Nolan
                3    *
                4    * A Proposed Startup File Standard
                5    *
                6    * Published in Call-APPLE, November, 1987
                7    * This program is in the public domain.
                8    *
                9    * This program mimics the ProDOS 16
                10   * SYSTEM.SETUP convention. It can be used
                11   * to install RAM disk drivers, clock
                12   * drivers, and IIGS Classic Desk
                13   * Accessories on bootup under ProDOS 8.
                14   *
                15   * This program loads and calls all BINary
                16   * and SYStem files in a subdirectory named
                17   * SETUPS. It then looks for the second
                18   * system program in the volume directory
                19   * whose name ends in ".SYSTEM", and runs
                20   * that.
                21   *
                22   *
                23            TYP   $FF        ;save as a system file
                24            ORG   $BD00      ;load at $2000, but run at $BD00
                25   ****************** equates
                26   CH       =     $24
                27   IN2      =     $280
                28   FILETYPE =     IN2+16
                29   AUXCODE  =     IN2+31
                30   RESET    =     $3F2
                31   IOBUFFER =     $B900
                32   PRODOS   =     $BF00
                33   QUITVECT =     $BF03
                34   DEVNUM   =     $BF30
                35   BITMAP   =     $BF58
                36   INIT     =     $FB2F
                37   VTABZ    =     $FC24
                38   HOME     =     $FC58
                39   RDKEY    =     $FD0C
                40   SETVID   =     $FE93
                41   SETKBD   =     $FE89
                42   SETNORM  =     $FE84
                43   ****************** boot code
                44   VOLNAME  =     *          ;The first 17 bytes are overwritten with the
                45                             ;name of the volume from which this was run.
BD00: A2 01     46            LDX   #1         ;mark page $BD as free in the system bitmap
BD02: 8E 6F BF  47            STX   BITMAP+23  ;so we can put Online result in our code.
BD05: CA        48            DEX              ;relocate this program to $BD00-BEFF
BD06: BD 00 20  49   LOOP1    LDA   $2000,X
BD09: 9D 00 BD  50            STA   $BD00,X
BD0C: BD 00 21  51            LDA   $2100,X
BD0F: 9D 00 BE  52            STA   $BE00,X
BD12: E8        53            INX
BD13: D0 F1     54            BNE   LOOP1
BD15: CA        55            DEX
BD16: 9A        56            TXS              ;init stack pointer
BD17: 4C 21 BD  57            JMP   ENTER      ;jump to relocated code
BD1A: 06        58   DIRNAME  DFB   6          ;DirName and VolName must be in the same page
BD1B: D3 C5 D0  59            ASC   "SETUPS"
BD1E: D5 D0 D3
                60   ******* Get name of boot volume
BD21: AD 30 BF  61   ENTER    LDA   DEVNUM     ;get name of last volume accessed
BD24: 8D BF BE  62            STA   ONLINEN
BD27: 20 00 BF  63            JSR   PRODOS
BD2A: C5        64            DFB   $C5        ;ONLINE
```

```
BD2B: BE BE     65            DA    ONLINEP
BD2D: AD Ø1 BD  66            LDA   VOLNAME+1  ;insert a slash nefore the name
BD3Ø: 29 ØF     67            AND   #$ØF
BD32: AA        68            TAX
BD33: E8        69            INX
BD34: 8E ØØ BD  7Ø            STX   VOLNAME
BD37: A9 2F     71            LDA   #$2F       ;/
BD39: 8D Ø1 BD  72            STA   VOLNAME+1
BD3C: AD Ø4 BF  73            LDA   QUITVECT+1 ;save original quit vector
BD3F: 8D D2 BD  74            STA   QUITMOD1+1
BD42: AD Ø5 BF  75            LDA   QUITVECT+2
BD45: 8D D7 BD  76            STA   QUITMOD2+1
                77   ******* Clean up before & after calling files
BD48: A2 Ø2     78   MAINLOOP LDX   #2         ;point Reset vector and ProDOS
BD4A: BD CB BD  79   LOOP3    LDA   JUMP+1,X   ;Quit vectors to MainLoop
BD4D: 9D F2 Ø3  8Ø            STA   RESET,X
BD5Ø: BD CA BD  81            LDA   JUMP,X
BD53: 9D Ø3 BF  82            STA   QUITVECT,X
BD56: CA        83            DEX
BD57: 1Ø F1     84            BPL   LOOP3
BD59: 9A        85            TXS              ;fix stack pointer (X=$FF)
BD5A: 2Ø 85 BE  86            JSR   CLOSE      ;close all open files
BD5D: A2 17     87            LDX   #23        ;clear system bit map
BD5F: A9 ØØ     88            LDA   #Ø
BD61: 9D 58 BF  89   LOOP2    STA   BITMAP,X
BD64: CA        9Ø            DEX
BD65: 1Ø FA     91            BPL   LOOP2
BD67: A9 CF     92            LDA   #$CF       ;mark pages Ø,1,4-7 as used
BD69: 8D 58 BF  93            STA   BITMAP
BD6C: A9 Ø7     94            LDA   #%111      ;mark pages $BD-$BF as used
BD6E: 8D 6F BF  95            STA   BITMAP+23
BD71: AD 82 CØ  96            LDA   $CØ82      ;Language card off
BD74: 8D ØC CØ  97            STA   $CØØC      ;4Ø-column
BD77: 8D ØE CØ  98            STA   $CØØE      ;normal character set
BD7A: 8D ØØ CØ  99            STA   $CØØØ      ;8ØSTORE off
BD7D: 2Ø 84 FE  1ØØ           JSR   SETNORM    ;normal
BD8Ø: 2Ø 2F FB  1Ø1           JSR   INIT       ;display text page 1
BD83: 2Ø 93 FE  1Ø2           JSR   SETVID     ;PR#Ø
BD86: 2Ø 89 FE  1Ø3           JSR   SETKBD     ;IN#Ø
                1Ø4  *Make sure boot volume is around
                1Ø5  *AND set prefix to the boot volume
BD89: 2Ø 58 FC  1Ø6  VOLMOUNT JSR   HOME
BD8C: 2Ø ØØ BF  1Ø7           JSR   PRODOS     ;set prefix to volume
BD8F: C6        1Ø8           DFB   $C6        ;SET PREFIX
BD9Ø: C5 BE     1Ø9           DA    PFX2P
BD92: 9Ø 28     11Ø           BCC   VOLOK
BD94: A2 ØD     111           LDX   #13
BD96: BD F1 BE  112  LOOP6    LDA   VOLTEXT-1,X ;print message “insert volume”
BD99: 9D AC Ø5  113           STA   $5A8+4,X
BD9C: CA        114           DEX
BD9D: DØ F7     115           BNE   LOOP6
BD9F: BD Ø1 BD  116  LOOP7    LDA   VOLNAME+1,X ;print volume name
BDA2: Ø9 8Ø     117           ORA   #$8Ø
BDA4: 9D BB Ø5  118           STA   $5A8+19,X
BDA7: E8        119           INX
BDA8: EC ØØ BD  12Ø           CPX   VOLNAME
BDAB: 9Ø F2     121           BCC   LOOP7
BDAD: A9 23     122           LDA   #35        ;go to CH=35, CV=11
BDAF: 85 24     123           STA   CH
BDB1: A9 ØB     124           LDA   #11
BDB3: 2Ø 24 FC  125           JSR   VTABZ
BDB6: 2Ø ØC FD  126           JSR   RDKEY      ;wait for keypress
BDB9: 4C 89 BD  127           JMP   VOLMOUNT
                128  ******* Get name of next file at IN2
BDBC: 2Ø 15 BE  129  VOLOK    JSR   NEXTFILE   ;get name of next file at IN2
```

```
BDBF: BØ ØD     13Ø           BCS   EXITLOOP   ;if error, we're done with setup files
                131  ******* Load and call setup file
BDC1: 2Ø ØØ BF  132           JSR   PRODOS     ;set prefix to SETUPS
BDC4: C6        133           DFB   $C6        ;SET PREFIX
BDC5: C2 BE     134           DA    PFX1P
BDC7: 2Ø 8E BE  135           JSR   READFILE   ;read in file whose name is at IN@
                136                            ;and call it if there was no error.
BDCA: 4C 48 BD  137  JUMP     JMP   MAINLOOP   ;3 bytes here copied into ProDOS quit vector
BDCD: 18        138           DFB   #$BD!$A5   ;3 bytes here are copied into reset vector
BDCE: EE F4 Ø3  139  EXITLOOP INC   RESET+2    ;scramble reset vector
BDD1: A9 ØØ     14Ø  QUITMOD1 LDA   #Ø         ;restore original quit vector
BDD3: 8D Ø4 BF  141           STA   QUITVECT+1
BDD6: A9 ØØ     142  QUITMOD2 LDA   #Ø
BDD8: 8D Ø5 BF  143           STA   QUITVECT+2
                144  ******* Look for second system program on disk
BDDB: A9 ØØ     145           LDA   #Ø         ;modify NextFile routine so that it searches
BDDD: 8D 3E BE  146           STA   NUMBER+1   ;the volume directory for system files only.
BDEØ: 8D 82 BE  147           STA   CHEKTYPE+1
BDE3: A9 ØØ     148           LDA   #<VOLNAME  ;NamePtr+1 does not bneed to be changed
BDE5: 8D D2 BE  149           STA   NAMEPTR    ;since VolName and DirName are in the same
page
BDE8: 2Ø 15 BE  15Ø  NEXTSYS  JSR   NEXTFILE
BDEB: BØ 1B     151           BCS   QUIT
BDED: AE 8Ø Ø2  152           LDX   IN2        ;see if file ends with ".SYSTEM"
BDFØ: AØ Ø6     153           LDY   #6
BDF2: BD 8Ø Ø2  154  LOOP4    LDA   IN2,X      ;I expect pathname at IN2 in low ASCII
BDF5: D9 ØE BE  155           CMP   SYSTEXT,Y
BDF8: DØ EE     156           BNE   NEXTSYS
BDFA: CA        157           DEX
BDFB: 88        158           DEY
BDFC: 1Ø F4     159           BPL   LOOP4
BDFE: EE Ø2 BE  16Ø           INC   MOD+1
BEØ1: A9 FF     161  MOD      LDA   #$FF       ;the first .SYSTEM program we find is this
BEØ3: FØ E3     162           BEQ   NEXTSYS    ;one, so skip it and look for next one.
BEØ5: 2Ø 8E BE  163           JSR   READFILE   ;if successful, never come back
BEØ8: 2Ø ØØ BF  164  QUIT     JSR   PRODOS
BEØB: 65        165           DFB   $65        ;QUIT
BEØC: C8 BE     166           DA    QUITP
BEØE: 2E 53 59  167  SYSTEXT  ASC   '.SYSTEM'
BE11: 53 54 45 4D
                168  * Get name of next system file or binary file
                169  *
                17Ø  * This routine is set up to look for both SYSTEM and
                171  * BINary files in the SETUPs subdirectory. It is later
                172  * modified to search for SYSTEM files only in the
                173  * volume directory. The locations which are changed
                174  * are ChekType+1, Number+1, and NamePtr (in the Open
                175  * parametr list)
                176  *
                177  * Returns carry if not found, clear if found.
BE15: 2Ø ØØ BF  178  NEXTFILE JSR   PRODOS
BE18: C8        179           DFB   $C8        ;OPEN
BE19: D1 BE     18Ø           DA    OPENP
BE1B: BØ 68     181           BCS   CLOSE
BE1D: AD D6 BE  182           LDA   OPENN
BE2Ø: 8D D8 BE  183           STA   MARKN
BE23: 8D DD BE  184           STA   READN
BE26: 2Ø ØØ BF  185           JSR   PRODOS     ;Read in first 39 bytes of directory to
BE29: CA        186           DFB   $CA        ;IN2. This gets the number of entries per
BE2A: DC BE     187           DA    READP      ;block and number of bytes per entry.
BE2C: BØ 57     188           BCS   CLOSE
BE2E: AD A3 Ø2  189           LDA   IN2+35     ;save number of bytes per directory entry
BE31: 8D 54 BE  19Ø           STA   ENTSIZE+1
BE34: AD A4 Ø2  191           LDA   IN2+36     ;save number of entries per directory block
BE37: 8D 44 BE  192           STA   ENTRIES+1
```

```
BE3A: EE 3E BE  193  NEXTENT  INC   NUMBER+1
BE3D: A9 ØØ     194  NUMBER   LDA   #Ø          ;self-modified operand
                195  * Retrieve catalog entry #A
BE3F: A2 FE     196           LDX   #$FE        ;build page index in X
BE41: E8        197  LOOP5    INX
BE42: E8        198           INX
BE43: C9 ØD     199  ENTRIES  CMP   #13
BE45: 9Ø Ø5     2ØØ           BCC   OK
BE47: ED 44 BE  2Ø1           SBC   ENTRIES+1
BE4A: BØ F5     2Ø2           BCS   LOOP5       ;always
BE4C: A8        2Ø3  OK       TAY
BE4D: A9 Ø4     2Ø4           LDA   #4          ;1st entry per directory block starts 4 bytes
in
BE4F: 88        2Ø5  LOOP1Ø   DEY
BE5Ø: 3Ø Ø8     2Ø6           BMI   OK2
BE52: 18        2Ø7           CLC
BE53: 69 27     2Ø8  ENTSIZE  ADC   #39         ;add size of directory entry
BE55: 9Ø F8     2Ø9           BCC   LOOP1Ø
BE57: E8        21Ø           INX
BE58: DØ F5     211           BNE   LOOP1Ø      ;always
BE5A: 8D D9 BE  212  OK2      STA   MARK        ;save mark in file
BE5D: 8E DA BE  213           STX   MARK+1
BE6Ø: 2Ø ØØ BF  214           JSR   PRODOS      ;set the mark
BE63: CE        215           DFB   $CE         ;SET_MARK
BE64: D7 BE     216           DA    MARKP
BE66: BØ 1D     217           BCS   CLOSE
BE68: 2Ø ØØ BF  218           JSR   PRODOS      ;read in directory info
BE6B: CA        219           DFB   $CA         ;READ
BE6C: DC BE     22Ø           DA    READP
BE6E: BØ 15     221           BCS   CLOSE
BE7Ø: AD 8Ø Ø2  222           LDA   IN2         ;make sure that file is not deleted
BE73: FØ C5     223           BEQ   NEXTENT
BE75: 29 ØF     224           AND   #$ØF
BE77: 8D 8Ø Ø2  225           STA   IN2
BE7A: AD 9Ø Ø2  226           LDA   FILETYPE    ;make sure file type is correct
BE7D: 49 FF     227           EOR   #$FF        ;we look for system programs...
BE7F: FØ Ø4     228           BEQ   CLOSE
BE81: 49 F9     229  CHEKTYPE EOR   #6!$FF      ;...and binary ones.
BE83: DØ B5     23Ø           BNE   NEXTENT
BE85: Ø8        231  CLOSE    PHP               ;close all files - do not change carry
BE86: 2Ø ØØ BF  232           JSR   PRODOS
BE89: CC        233           DFB   $CC         ;CLOSE
BE8A: CF BE     234           DA    CLOSEP
BE8C: 28        235           PLP
BE8D: 6Ø        236  ANRTS    RTS
                237  * Read file and call it.
                238  * Name should be found at IN2
                239  * Prefix must be set.
BE8E: AE 9Ø Ø2  24Ø  READFILE LDX   FILETYPE    ;if a system program, set to read to $2ØØØ
BE91: A9 2Ø     241           LDA   #$2Ø
BE93: E8        242           INX
BE94: FØ Ø6     243           BEQ   SETDEST
BE96: AE 9F Ø2  244           LDX   AUXCODE     ;else, set to read in file at address
BE99: AD AØ Ø2  245           LDA   AUXCODE+1   ;found in auxcode
BE9C: 8E EC BE  246  SETDEST  STX   READ2D
BE9F: 8D ED BE  247           STA   READ2D+1
BEA2: 2Ø ØØ BF  248           JSR   PRODOS      ;Open file
BEA5: C8        249           DFB   $C8         ;OPEN
BEA6: E4 BE     25Ø           DA    OPEN2P
BEA8: BØ DB     251           BCS   CLOSE
BEAA: AD E9 BE  252           LDA   OPEN2N
BEAD: 8D EB BE  253           STA   READ2N
BEBØ: 2Ø ØØ BF  254           JSR   PRODOS      ;Read file into memory
BEB3: CA        255           DFB   $CA         ;READ
BEB4: EA BE     256           DA    READ2P
```

```
                257           JSR   CLOSE
                258           BCS   ANRTS
                259           JMP   (READ2D)   ;call the file just loaded
                26Ø  ******* ProDOS MLI parameter lists
                261  ONLINEP  DFB   2          ;Online parameter list
                262  ONLINEN  DS    1
                263           DA    VOLNAME+1
                264  *
BEC2: Ø1        265  PFX1P    DFB   1          ;to set prefix to SETUP
BEC3: 1A BD     266           DA    DIRNAME
                267  *
BEC5: Ø1        268  PFX2P    DFB   1          ;to set prefix to volume directory
BEC6: ØØ BD     269           DA    VOLNAME
                27Ø  *
BEC8: Ø4 ØØ ØØ  271  QUITP    DFB   4,Ø,Ø,Ø,Ø,Ø,Ø
BECB: ØØ ØØ ØØ ØØ
                272  *
BECF: Ø1 ØØ     273  CLOSEP   DFB   1,Ø        ;close all files
                274  *
BED1: Ø3        275  OPENP    DFB   3          ;open directory
BED2: 1A BD     276  NAMEPTR  DA    DIRNAME    ;pathname pointer
BED4: ØØ B9     277           DA    IOBUFFER
BED6: ØØ        278  OPENN    DS    1          ;reference number
                279  *
BED7: Ø2        28Ø  MARKP    DFB   2          ;set mark in directory
BED8: ØØ        281  MARKN    DS    1
BED9: ØØ ØØ ØØ  282  MARK     DS    3
                283  *
BEDC: Ø4        284  READP    DFB   4          ;read directory
BEDD: ØØ        285  READN    DS    1
BEDE: 8Ø Ø2     286           DA    IN2        ;target address
BEEØ: 27 ØØ     287           DA    39         ;length
BEE2: ØØ ØØ     288           DS    2
                289  *
BEE4: Ø3        29Ø  OPEN2P   DFB   3          ;open setup or system file
BEE5: 8Ø Ø2     291           DA    IN2
BEE7: ØØ B9     292           DA    IOBUFFER
BEE9: ØØ        293  OPEN2N   DS    1
                294  *
BEEA: Ø4        295  READ2P   DFB   4          ;read setup or system file
BEEB: ØØ        296  READ2N   DS    1
BEEC: ØØ ØØ     297  READ2D   DS    2          ;destination of file is self-mod here
BEEE: ØØ B1     298           DA    $B9ØØ-$8ØØ ;ask for largest possible that will fit
BEFØ: ØØ ØØ     299           DS    2
                3ØØ  *
BEF2: C9 CE D3  3Ø1  VOLTEXT  ASC   "INSERT VOLUME"
BEF5: C5 D2 D4 AØ
BEF9: D6 CF CC D5
BEFD: CD C5
BEFF: E8        3Ø2           CHK              ;checksum - eor for all previous bytes

--End assembly, 512 bytes, Errors: Ø
```

No DOS Boot Sector

The following program can be ZAPped onto track 0, sector 0 of a DOS 3.3 disk to create a data disk. You can then change the disk free sector bitmap to use tracks 1 and 2 on your DOS 3.3 disks, giving you 32 extra sectors for data. See Chapter 4 of *Beneath Apple DOS* for information on the disk free sector bit map.

```
                1     ************************
                2     *
                3     * NO DOS BOOT
                4     *
                5     ************************
                6     *
                7              ORG   $Ø8ØØ
                8     SLOTNUM  EQU   $2B
                9     REBOOT   EQU   $FE
                1Ø    RESET    EQU   $Ø3F4
                11    MOTOROFF EQU   $CØ88
                12    BELL     EQU   $FBDD
                13    COUT     EQU   $FDED
                14    GETKEY   EQU   $FD1B
                15    HOME     EQU   $FC58
                16    CV       EQU   $25
                17    VTAB     EQU   $FC22
                18    *
Ø8ØØ: Ø1        19             DFB   $Ø1         ; LOAD ONLY 1 SECTOR FROM DISK (TØ
SØ)
Ø8Ø1: A6 2B     2Ø    ENTRY    LDX   SLOTNUM     ; LEFT OVER FROM BOOT Ø ROM
Ø8Ø3: BD 88 CØ  21             LDA   MOTOROFF,X  ; TURN OFF THE MOTOR
Ø8Ø6: 8A        22             TXA
Ø8Ø7: 4A        23             LSR
Ø8Ø8: 4A        24             LSR
Ø8Ø9: 4A        25             LSR
Ø8ØA: 4A        26             LSR
Ø8ØB: Ø9 CØ     27             ORA   #$CØ        ; $Øx -> $Cx
Ø8ØD: 85 FF     28             STA   REBOOT+1    ; KEEP IT FOR LATER
Ø8ØF: A9 ØØ     29             LDA   #$ØØ
Ø811: 85 FE     3Ø             STA   REBOOT
Ø813: EE F4 Ø3  31             INC   RESET       ; USER RESET => REBOOT
Ø816: 2Ø 58 FC  32             JSR   HOME        ; CLEAR THE SCREEN
Ø819: 2Ø DD FB  33             JSR   BELL        ; BEEP TO ALERT USER
                34                               ; REMOVE PREVIOUS LINE FOR NO BELL
Ø81C: A9 Ø9     35             LDA   #$Ø9        ; GO DOWN TO LINE 1Ø
Ø81E: 85 25     36             STA   CV          ; STORE IT THEN
Ø82Ø: 2Ø 22 FC  37             JSR   VTAB        ;  SET BASL,H FOR PRINTING
Ø823: A2 ØØ     38             LDX   #$ØØ        ; START AT THE BEGINNING OF THE MSG
Ø825: BD 36 Ø8  39    PRINT    LDA   MSG,X       ; GET EACH LETTER
Ø828: FØ Ø6     4Ø             BEQ   DONE        ; $ØØ MEANS WE'RE FINISHED
Ø82A: 2Ø ED FD  41             JSR   COUT        ; PRINT IT
Ø82D: E8        42             INX               ; ADVANCE TO NEXT CHARACTER
```

```
Ø82E: DØ F5     43            BNE   PRINT      ; LOOP TO PRINT IT
Ø83Ø: 2Ø 1B FD  44   DONE     JSR   GETKEY     ; WAIT FOR KEYPRESS
Ø833: 6C FE ØØ  45            JMP   (REBOOT)   ; REBOOT SYSTEM
Ø836: AØ AØ AØ  46   MSG      ASC   "       THIS DISKETTE CANNOT BOOT."
Ø839: AØ AØ AØ AØ D4 C8 C9 D3
Ø841: AØ C4 C9 D3 CB C5 D4 D4
Ø849: C5 AØ C3 C1 CE CE CF D4
Ø851: AØ C2 CF CF D4 AE
Ø857: 8D        47            DFB   $8D        ; CARRIAGE RETURN
Ø858: AØ AØ AØ  48            ASC   "       PLEASE PLACE A SYSTEM DISK"
Ø85B: AØ AØ AØ AØ DØ CC C5 C1
Ø863: D3 C5 AØ DØ CC C1 C3 C5
Ø86B: AØ C1 AØ D3 D9 D3 D4 C5
Ø873: CD AØ C4 C9 D3 CB
Ø879: 8D        49            DFB   $8D        ; CARRIAGE RETURN
Ø87A: AØ AØ AØ  5Ø            ASC   "     IN DRIVE #1 AND PRESS <RETURN>"
Ø87D: AØ AØ C9 CE AØ C4 D2 C9
Ø885: D6 C5 AØ A3 B1 AØ C1 CE
Ø88D: C4 AØ DØ D2 C5 D3 D3 AØ
Ø895: BC D2 C5 D4 D5 D2 CE BE
Ø89D: 8D        51            DFB   $8D        ; CARRIAGE RETURN
Ø89E: ØØ        52            DFB   $ØØ        ; $ØØ MEANS END OF MESSAGE
Ø89F: ØØ ØØ ØØ  53            DS    $9ØØ-*
Ø8A2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8AA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8B2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8BA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8C2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8CA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8D2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8DA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8E2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8EA: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8F2: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8FA: ØØ ØØ ØØ ØØ ØØ ØØ

--End assembly, 256 bytes, Errors: Ø
```

Sample Decryption Log

```
---------------BurgerTime--------------
A 4am crack                  2Ø15-12-31
---------------------------------------

Name: BurgerTime
Genre: arcade
Year: 1982
Author: Data East USA
Publisher: Mattel
Media: single-sided 5.25-inch floppy
OS: DOS 3.3 with custom bootloader and
    custom file loader and custom RWTS,
    so not really like DOS 3.3 at all
Previous cracks: The Freeze

                   ~

              Chapter Ø
 In Which Various Automated Tools Fail
          In Interesting Ways


COPYA
  read error on first pass

Locksmith Fast Disk Backup
  bizarre combination of success and
  failure

                 --v--

     LOCKSMITH 7.Ø  FAST DISK BACKUP


   R.**********.***..*...**************
   W
HEX ØØØØØØØØØØØØØØØØ1111111111111111222
TRK Ø123456789ABCDEFØ123456789ABCDEFØ12
   Ø.AAAAAAAAAA.AAA.......DDDDDDDDDDDDD
   1.AAAAAAAAAA.AAA.......DDDDDDDDDDDDD
   2.AAAAAAAAAA.AAA.......DDDDDDDDDDDDD
   3.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
   4.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
   5.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
   6.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
   7.AAAAAAAAAA.AAA..D...DDDDDDDDDDDDDD
   8.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
   9.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
   A.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
   B.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
   C.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
   D.AAAAAAAAAA.AAA......DDDDDDDDDDDDDD
12 E.AAAAAAAAAA.AAA......DDDDDDDDDDDDD.
   F.AAAAAAAAAA.AAA......DDDDDDDDDDDDD.
[                ] PRESS [RESET] TO EXIT

                 --^--

  Tracks $ØØ, $ØB, $ØF, $1Ø, $12, $13,
  and $14 are standard. Also a few
  sectors of T15 and T22, and all of
  T11 except for what appears to be a
  bad sector in the middle. (The
  original disk boots fine, and the
  entire disk catalog on that track is
  fake anyway, so I think I got lucky.)

EDD 4 bit copy (no sync, no count)
  no errors, but copy displays an error
  "UNABLE TO LOAD GAME" and hangs

Copy ][+ nibble editor
  TØ1-TØA, TØC-TØE are unformatted
    (hi-res disk scan confirms this)
  T15+ use a modified data prologue
    ("D5 AA AA" instead of "D5 AA AD")

Disk Fixer
  TØØ -> custom bootloader
  T11 -> DOS 3.3 style disk catalog
    (fake, all files point to nothing)
  ["O" -> "Input/Output Control"]
    set Data Prologue to "D5 AA AA"
  Success! T15+ readable

Why didn't COPYA work?
  unformatted tracks, modified prologue

Why didn't Locksmith FDB work?
  modified prologue

Why didn't my EDD copy work?
  I assume there's some sort of
  protection check during boot. Disks
  don't say "UNABLE TO LOAD GAME"
  unless someone tells them to.

Next steps:

  1. Trace the boot
  2. ???

                   ~

               Chapter 1
    In Which We Start From Scratch


We're starting from bare metal on this
one. My automated tools, they do nothing
for us. Strap in.

[S6,D1=original disk]

[S6,D2=crack-in-progress (the partial
  copy I made with Locksmith Fast Disk
  Backup)]

[S5,D1=my work disk]
```

```
]PR#5
...
]CALL -151

*96ØØ<C6ØØ.C6FFM

; copy boot sector (TØØ,SØØ) to the
; graphics page so it survives a reboot
96F8-   AØ ØØ       LDY   #$ØØ
96FA-   B9 ØØ Ø8    LDA   $Ø8ØØ,Y
96FD-   99 ØØ 2Ø    STA   $2ØØØ,Y
97ØØ-   C8          INY
97Ø1-   DØ F7       BNE   $96FA

; turn off slot 6 drive motor
97Ø3-   AD E8 CØ    LDA   $CØE8

; reboot to my work disk in slot 5
97Ø6-   4C ØØ C5    JMP   $C5ØØ

*96ØØG
...reboots slot 6...
...reboots slot 5...

]BSAVE BOOTØ,A$2ØØØ
]CALL -151

*8ØØ<2ØØØ.2ØFFM

*8Ø1L

; starts off looking like a DOS 3.3
; bootloader -- zp$27 will be $Ø9 the
; first time through, so this is a way
; to do one-time initialization
Ø8Ø1-   A5 27       LDA   $27
Ø8Ø3-   C9 Ø9       CMP   #$Ø9
Ø8Ø5-   DØ 1B       BNE   $Ø822

; munge reset vector
Ø8Ø7-   A5 2B       LDA   $2B
Ø8Ø9-   8D F4 Ø3    STA   $Ø3F4

; munge boot slot into a vector to read
; more sectors
Ø8ØC-   4A          LSR
Ø8ØD-   4A          LSR
Ø8ØE-   4A          LSR
Ø8ØF-   4A          LSR
Ø81Ø-   Ø9 CØ       ORA   #$CØ
Ø812-   85 D7       STA   $D7
Ø814-   A9 5C       LDA   #$5C
Ø816-   85 D6       STA   $D6
```

Now ($D6) points to $Cx5C (based on the boot slot, e.g. $C65C for slot 6). This entry point will read a sector from track $ØØ and exit via $Ø8Ø1.

As a bonus, setting $D6 to anything other than Ø sets the RUN flag, so if I managed to break to a BASIC prompt, any command would run. Even trying to do a "CALL -151" to get to the monitor would be blocked. Lovely.

```
Ø818-   18          CLC
Ø819-   AD F9 Ø8    LDA   $Ø8F9
Ø81C-   6D FA Ø8    ADC   $Ø8FA
Ø81F-   8D F9 Ø8    STA   $Ø8F9
Ø822-   AE FA Ø8    LDX   $Ø8FA
Ø825-   FØ 15       BEQ   $Ø83C
```

```
*8F9.8FA

Ø8F9- Ø3 Ø4
```

We're loading 4 sectors directly into the text page ($Ø4ØØ..$Ø7FF). Nice.

```
; set up the rest of the zero page
; globals for the sector read routine
Ø827-   BD FB Ø8    LDA   $Ø8FB,X
Ø82A-   85 3D       STA   $3D
Ø82C-   CE FA Ø8    DEC   $Ø8FA
Ø82F-   AD F9 Ø8    LDA   $Ø8F9
Ø832-   85 27       STA   $27
Ø834-   CE F9 Ø8    DEC   $Ø8F9

; read a sector (exits via $Ø8Ø1)
Ø837-   A6 2B       LDX   $2B
Ø839-   6C D6 ØØ    JMP   ($ØØD6)

; Execution continues here (from $Ø825)
; after we've read all the sectors we
; wanted to read (into $Ø4ØØ..$Ø7FF).
; Switch to the hi-res graphics screen
; (uninitialized).
Ø83C-   2C 52 CØ    BIT   $CØ52
Ø83F-   2C 55 CØ    BIT   $CØ55
Ø842-   2C 57 CØ    BIT   $CØ57
Ø845-   2C 5Ø CØ    BIT   $CØ5Ø

; call part of the code we just read
Ø848-   2Ø ØA Ø4    JSR   $Ø4ØA
```

And that's where I need to interrupt the boot.

~

Chapter 2
In Which We Eschew Obfuscation

```
*96ØØ<C6ØØ.C6FFM

; set up callback after bootØ loads the
; code into the text page
96F8-   A9 4C       LDA   #$4C
96FA-   8D 48 Ø8    STA   $Ø848
96FD-   A9 ØA       LDA   #$ØA
96FF-   8D 49 Ø8    STA   $Ø849
97Ø2-   A9 97       LDA   #$97
97Ø4-   8D 4A Ø8    STA   $Ø84A

; start the boot
97Ø7-   4C Ø1 Ø8    JMP   $Ø8Ø1

; callback is here --
; copy the text page to higher memory
; so it survives a reboot
97ØA-   A2 Ø4       LDX   #$Ø4
97ØC-   AØ ØØ       LDY   #$ØØ
97ØE-   B9 ØØ Ø4    LDA   $Ø4ØØ,Y
9711-   99 ØØ 24    STA   $24ØØ,Y
9714-   C8          INY
9715-   DØ F7       BNE   $97ØE
9717-   EE 1Ø 97    INC   $971Ø
971A-   EE 13 97    INC   $9713
971D-   CA          DEX
971E-   DØ EE       BNE   $97ØE

; turn off the slot 6 drive motor
972Ø-   AD E8 CØ    LDA   $CØE8
```

```
; reboot to my work disk
9723-   4C 00 C5    JMP   $C500

*BSAVE TRACE,A$9600,L$126
*9600G
...reboots slot 6...
...reboots slot 5...

]BSAVE BOOT1 0400-07FF,A$2400,L$400
]CALL -151
```

I'll need to leave this code at $2400 for inspection, since I can't put it in the text page and also inspect it in the monitor. Relative branches will look correct, but absolute addresses will be off by $2000.

```
*240AL

; wipe most of main memory, $0900+
240A-   A0 09       LDY   #$09
240C-   84 01       STY   $01
240E-   A9 00       LDA   #$00
2410-   85 00       STA   $00
2412-   A8          TAY
2413-   91 00       STA   ($00),Y
2415-   C8          INY
2416-   D0 FB       BNE   $2413
2418-   E6 01       INC   $01
241A-   A6 01       LDX   $01
241C-   E0 C0       CPX   #$C0
241E-   D0 F3       BNE   $2413
2420-   60          RTS
```

Continuing in boot0, from $084B...

```
*BLOAD BOOT0,A$800

*84BL

; set a (presumably unfriendly) reset
; vector
084B-   A9 00       LDA   #$00
084D-   8D F2 03    STA   $03F2
0850-   A9 04       LDA   #$04
0852-   8D F3 03    STA   $03F3
0855-   49 A5       EOR   #$A5
0857-   8D F4 03    STA   $03F4
085A-   EA          NOP
085B-   EA          NOP
085C-   EA          NOP
085D-   EA          NOP
085E-   20 21 04    JSR   $0421

*2421L
```

This code is exquisitely obfuscated, so let's just take it one line at a time. Everything is important.

```
2421-   A9 00       LDA   #$00    ;A=00
2423-   85 82       STA   $82     ; =00

; Stack pointer is uninitialized at
; this point (and not guaranteed to be
; anything in particular on boot), so I
; assume this is the first part of a
; verification that the stack pointer
; isn't being modified by anything the
; original developers didn't know about
; (like an NMI card).
2425-   BA          TSX
2426-   8A          TXA
2427-   49 BC       EOR   #$BC
2429-   85 50       STA   $50

242B-   29 00       AND   #$00    ;A=00
242D-   AA          TAX           ;X=00
242E-   A0 D6       LDY   #$D6    ;Y=D6
2430-   49 54       EOR   #$54    ;A=54
2432-   84 81       STY   $81     ; =D6
```

Now ($81) points to $00D6. That address was set to $5C (at $0816), because it was used as the pointer to $Cx5C to read more sectors from disk (including the code we're now executing).

```
; X=00, so this is taking ($81), which
; is zp$D6, which is $5C, then EOR'ing
; it with A, which is $54
2434-   41 81       EOR   ($81,X) ;A=08
2436-   85 21       STA   $21     ; =08
2438-   98          TYA           ;A=D6
2439-   49 E0       EOR   #$E0    ;A=36
243B-   85 81       STA   $81     ; =36
```

Now ($81) points to $0036, the output vector.

```
243D-   49 A4       EOR   #$A4    ;A=92
243F-   81 81       STA   ($81,X) ; =92
2441-   E6 81       INC   $81     ; =37
2443-   38          SEC
2444-   E9 8B       SBC   #$8B    ;A=07
2446-   30 F0       BMI   $2438   ;nope
2448-   81 81       STA   ($81,X) ; =07
```

Now ($36) points to $0792.

```
244A-   49 07       EOR   #$07    ;A=00
244C-   85 20       STA   $20     ; =00
```

Now ($20) points to $0800, which is the start of the boot0 code.

```
244E-   85 51       STA   $51     ; =00
2450-   49 04       EOR   #$04    ;A=04
2452-   F0 02       BEQ   $2456   ;nope
2454-   85 46       STA   $46     ; =04
2456-   8A          TXA           ;A=00
2457-   85 45       STA   $45     ; =00
```

Now ($45) points to $0400, which is the start of the boot1 code.

```
2459-   18          CLC
245A-   69 D6       ADC   #$D6    ;A=D6
245C-   85 81       STA   $81     ; =D6
```

Now ($81) points to $00D6 again.

```
245E-   4A          LSR           ;A=6B
245F-   AA          TAX           ;X=6B

; $16 + $6B = $81, and ($81) points to
; $00D6, and zp$D6 is still $5C.
2460-   A1 16       LDA   ($16,X) ;A=5C
2462-   49 32       EOR   #$32    ;A=6E
2464-   A8          TAY           ;Y=6E
2465-   49 6A       EOR   #$6A    ;A=04
2467-   F0 E5       BEQ   $244E   ;nope
2469-   85 08       STA   $08     ; =04
246B-   49 3E       EOR   #$3E    ;A=3A
246D-   88          DEY           ;Y=6D
```

```
; ($45) points to $Ø4ØØ, and Y=$6D,
; so ($45),Y = $Ø46D, which is $88 (the
; "DEY" instruction we just executed).
; A=$3A, and $3A EOR $88 = $B2.
246E-   51 45       EOR   ($45),Y ;A=B2

; ($36) points to $Ø792, and Y=$6D,
; so ($36),Y = $Ø7FF, which is $AØ
; (trust me on that). A=$B2 after the
; previous EOR, so $B2 EOR $AØ = $12.
247Ø-   51 36       EOR   ($36),Y ;A=12
2472-   38          SEC

; ($2Ø) points to $Ø8ØØ, and Y=$6D,
; so ($2Ø),Y = $Ø86D, which is $Ø9.
; A=$12 after the two previous EORs, so
; $12 - $Ø9 = $Ø9. (Think hex.)
2473-   F1 2Ø       SBC   ($2Ø),Y ;A=Ø9

; store that in $Ø7FF
2475-   91 36       STA   ($36),Y ; =Ø9

; done decrypting this page?
2477-   CØ ØØ       CPY   #$ØØ

; nope, branch back to finish it
2479-   DØ F2       BNE   $246D
```

That branch goes to $Ø46D, which is a "DEY" instruction, so we're decrypting memory from the top down.

```
247B-   84 81       STY   $81     ; =ØØ

; decrement page count (initialized to
; Ø4 at $Ø469)
247D-   C6 Ø8       DEC   $Ø8

; jump forward if we're done decrypting
247F-   FØ Ø5       BEQ   $2486

; otherwise decrement the target page
; and loop back to decrypt it
2481-   C6 37       DEC   $37
2483-   4C 6D Ø4    JMP   $Ø46D
```

On the first pass, Y started at $6D, so we only decrypted part of the page. But the index was offset from ($36), which was $Ø792 (set at $Ø448), so we end up decrypting $Ø7FF..$Ø792. Then zp$37 is decremented and Y rolls over to $FF, so we decrypt a full $1ØØ bytes in round 2 ($Ø791..$Ø692). Then $Ø691..$Ø592. Then $Ø591..$Ø492. Then zp$Ø8 hits Ø and we branch forward to $Ø486.

```
; zp$5Ø was set from the uninitialized
; stack pointer (EOR'd with some magic
; number), and now we're using that as
; an index into the stack to get some
; other (also uninitialized) byte.
; Again, this will probably be checked
; later to see if the stack has been
; tampered with.
2486-   A6 5Ø       LDX   $5Ø
2488-   BD ØØ Ø1    LDA   $Ø1ØØ,X
248B-   85 46       STA   $46
248D-   49 47       EOR   #$47
248F-   8D 33 6Ø    STA   $6Ø33

; everything after this was decrypted
; by the loop we just executed, so this
; listing is meaningless
2492-   64          ???
2493-   1E 11 B4    ASL   $B411,X
2496-   FE 36 96    INC   $9636,X
2499-   7Ø FE       BVS   $2499
```

To sum up: boot1 decrypts itself using both the calling code and itself as the decryption key, then immediately falls through to decrypted code (at $Ø492). Patching bootØ to interrupt the boot will cause the decryption to fail. Patching boot1 to interrupt the boot will cause the decryption to fail.

I am stuck.

~

Chapter 3
In Which We Get Unstuck

The decryption routine at $Ø421 is complicated and interconnected, but one thing it does not rely on is the caller address. I can simply call it myself and see what happens. As long as bootØ and boot1 are pristine by the time the routine is called, it should work.

```
*96ØØ<C6ØØ.C6FFM

; set up callback after bootØ loads
; boot1
96F8-   A9 4C       LDA   #$4C
96FA-   8D 48 Ø8    STA   $Ø848
96FD-   A9 ØA       LDA   #$ØA
96FF-   8D 49 Ø8    STA   $Ø849
97Ø2-   A9 97       LDA   #$97
97Ø4-   8D 4A Ø8    STA   $Ø84A

; start the boot
97Ø7-   4C Ø1 Ø8    JMP   $Ø8Ø1

; callback is here --
; undo the patch we made earlier, so
; bootØ is in a "pristine" state for
; the decryption routine
97ØA-   A9 2Ø       LDA   #$2Ø
97ØC-   8D 48 Ø8    STA   $Ø848
97ØF-   A9 ØA       LDA   #$ØA
9711-   8D 49 Ø8    STA   $Ø849
9714-   A9 Ø4       LDA   #$Ø4
9716-   8D 4A Ø8    STA   $Ø84A

; call the decryption routine
9719-   2Ø 21 Ø4    JSR   $Ø421

; copy the (hopefully decrypted) boot1
; code to higher memory so it survives
; a reboot
971C-   A2 Ø4       LDX   #$Ø4
971E-   AØ ØØ       LDY   #$ØØ
972Ø-   B9 ØØ Ø4    LDA   $Ø4ØØ,Y
9723-   99 ØØ 24    STA   $24ØØ,Y
9726-   C8          INY
9727-   DØ F7       BNE   $972Ø
9729-   EE 22 97    INC   $9722
972C-   EE 25 97    INC   $9725
972F-   CA          DEX
973Ø-   DØ EE       BNE   $972Ø
```

```
; turn off slot 6 drive motor and
; reboot to my work disk
9732-   AD E8 CØ    LDA   $CØE8
9735-   4C ØØ C5    JMP   $C5ØØ

*BSAVE TRACE2,A$96ØØ,L$138
*96ØØG
...reboots slot 6...
...reboots slot 5...

]BSAVE BOOT1 DECRYPTED,A$24ØØ,L$4ØØ
]CALL -151
```

Let's see what that decrypted code was that we were falling through to.

```
*2492L

2492-   6Ø          RTS
```

Well OK then. I guess it worked.

~

Chapter 4
In Which We Have A Revelation

Continuing the trace from $Ø861 (after calling the decryption routine)...

```
*BLOAD BOOTØ,A$8ØØ
*861L

Ø861-   A9 ØØ       LDA   #$ØØ
Ø863-   85 24       STA   $24
Ø865-   85 83       STA   $83
Ø867-   85 ØØ       STA   $ØØ
Ø869-   8D 78 Ø2    STA   $Ø278
Ø86C-   A9 Ø9       LDA   #$Ø9
Ø86E-   85 25       STA   $25
Ø87Ø-   85 Ø1       STA   $Ø1
```

Not sure what all that's about, but ($24) points to $Ø9ØØ and ($ØØ) points to $ØØØØ.

```
; Hey, I have this decrypted now!
Ø872-   2Ø CØ Ø4    JSR   $Ø4CØ

*24CØL

; get boot slot (x16) -- generally a
; precursor to doing something disk-
; related
24CØ-   A6 2B       LDX   $2B

; (not shown) this subroutine divides
; X by 16 and puts it in Y, so if we
; booted from slot 6, Y is now 6
24C2-   2Ø 6A Ø5    JSR   $Ø56A

; set up some stuff based on boot slot
; (this looks suspiciously like the
; kind of initialization that DOS 3.3
; does, tracking the current track of
; each disk in each slot+drive)
24C5-   A9 ØØ       LDA   #$ØØ
24C7-   99 78 Ø2    STA   $Ø278,Y
24CA-   99 88 Ø2    STA   $Ø288,Y
24CD-   8D 78 Ø2    STA   $Ø278

; don't know what these values mean,
; but they're probably important
24DØ-   A9 83       LDA   #$83
24D2-   85 76       STA   $76
24D4-   A9 11       LDA   #$11
24D6-   85 8Ø       STA   $8Ø
24D8-   A9 Ø7       LDA   #$Ø7
24DA-   85 81       STA   $81
24DC-   85 82       STA   $82
24DE-   2Ø BE Ø6    JSR   $Ø6BE

*26BEL

26BE-   2Ø Ø6 Ø5    JSR   $Ø5Ø6

*25Ø6L

25Ø6-   A9 8Ø       LDA   #$8Ø
25Ø8-   85 Ø2       STA   $Ø2
25ØA-   A9 ØØ       LDA   #$ØØ
25ØC-   85 Ø3       STA   $Ø3
```

Now ($Ø2) points to $ØØ8Ø, which was set to $11 at $Ø4D6.

```
25ØE-   AØ ØØ       LDY   #$ØØ     ;Y=ØØ
251Ø-   B1 Ø2       LDA   ($Ø2),Y  ;A=11
2512-   85 4Ø       STA   $4Ø      ; =11
2514-   C8          INY            ;Y=Ø1
2515-   B1 Ø2       LDA   ($Ø2),Y  ;A=Ø7
2517-   85 F2       STA   $F2      ; =Ø7
2519-   C8          INY            ;Y=Ø2
251A-   B1 Ø2       LDA   ($Ø2),Y  ; =Ø7
251C-   85 3F       STA   $3F      ; =Ø7
251E-   E6 Ø2       INC   $Ø2
252Ø-   E6 Ø2       INC   $Ø2
2522-   E6 Ø2       INC   $Ø2
```

Now ($Ø2) points to $ØØ83, which was set to $ØØ at $Ø865.

```
2524-   A9 D8       LDA   #$D8     ;A=D8
2526-   85 47       STA   $47      ; =D8
2528-   A4 F2       LDY   $F2      ;Y=Ø7

; map of physical to logical sectors
252A-   B9 5A Ø5    LDA   $Ø55A,Y
252D-   85 E9       STA   $E9
252F-   2Ø 71 Ø5    JSR   $Ø571

*2571L

2571-   A6 2B       LDX   $2B
2573-   2Ø 7A Ø5    JSR   $Ø57A
2576-   2Ø FF Ø5    JSR   $Ø5FF
2579-   6Ø          RTS
```

I won't show $Ø57A, but it moves the drive arm to the track given in zp$4Ø, which was set to $11 at $Ø512.

```
*25FFL

; disk read routine that looks
; suspiciously like the one in the
; Disk II firmware (at $C65C)
25FF-   18          CLC
26ØØ-   Ø8          PHP
26Ø1-   BD 8C CØ    LDA   $CØ8C,X
26Ø4-   1Ø FB       BPL   $26Ø1
26Ø6-   49 D5       EOR   #$D5
26Ø8-   DØ F7       BNE   $26Ø1
26ØA-   BD 8C CØ    LDA   $CØ8C,X
```

```
26ØD-   1Ø FB       BPL   $26ØA
26ØF-   C9 AA       CMP   #$AA
2611-   DØ F3       BNE   $26Ø6
2613-   EA          NOP
2614-   BD 8C CØ    LDA   $CØ8C,X
2617-   1Ø FB       BPL   $2614
2619-   C9 96       CMP   #$96
261B-   FØ Ø9       BEQ   $2626
261D-   28          PLP
261E-   9Ø DF       BCC   $25FF
262Ø-   49 AA       EOR   #$AA    <-- !
2622-   FØ 25       BEQ   $2649
2624-   DØ D9       BNE   $25FF
.
. [rest of routine is uninteresting,
.  except to note that it stores the
.  sector data in ($24), which points
.  to $Ø9ØØ (set at $Ø86E)]
.
```

Two key takeaways here. First, we’re reading the unreadable sector T11,SØ7. I originally thought this was a bad sector on my original disk, but I was wrong. Going back to it in a sector editor, it’s perfectly readable if I set the data prologue to “D5 AA AA”. It looks like this:

```
                 --v--

-------------- DISK EDIT --------------
TRACK $11/SECTOR $Ø7/VOLUME $FE/BYTE$ØØ
---------------------------------------
$ØØ:>FF<ØØ ØØ 11 Ø7 ØØ ØØ ØØ   .@@QG@@@
$Ø8: ØØ ØØ ØØ 22 ØF Ø2 C8 C5   @@@"OBHE
$1Ø: CC CC CF AØ AØ AØ AØ AØ   LLO
$18: AØ AØ AØ AØ AØ AØ AØ AØ
$2Ø: AØ AØ AØ AØ AØ AØ AØ AØ
$28: AØ AØ AØ AØ Ø2 ØØ 22 ØD       B@"M
$3Ø: Ø4 C2 D5 D2 C7 C5 D2 D4   DBURGERT
$38: C9 CD C5 AØ AØ AØ AØ AØ   IME
$4Ø: AØ AØ AØ AØ AØ AØ AØ AØ
$48: AØ AØ AØ AØ AØ AØ AØ B5          5
$5Ø: ØØ 17 Ø8 Ø4 CD C4 D3 C1   @WHDMDSA
$58: C4 CA AØ AØ AØ AØ AØ AØ   DJ
$6Ø: AØ AØ AØ AØ AØ AØ AØ AØ
$68: AØ AØ AØ AØ AØ AØ AØ AØ
$7Ø: AØ AØ 26 ØØ ØØ ØØ ØØ ØØ     &@@@@@
$78: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   @@@@@@@@
---------------------------------------
BUFFER Ø/SLOT 6/DRIVE 1/MASK OFF/NORMAL

---------------------------------------
COMMAND : _

                 --^--
```

That looks *exactly* like a DOS 3.3 catalog sector. I’ll come back to that revelation later.

Takeaway #2: the RWTS that reads this sector (and presumably the rest of the disk) is tucked away behind the insane decryption routine at $Ø421. In order to modify it to read a standard format disk (by changing $Ø621 from $AA to $AD to match the third nibble of the data prologue), I’ll need to write the unencrypted bootloader back to disk and disable the decryption routine.

~

Chapter 5
In Which We Make The Sort Of Progress That Doesn’t Feel Like Progress

```
]PR#5
...
]CALL -151

; straightforward multi-sector write
; loop, via the RWTS vector at $Ø3D9
Ø8CØ-   A9 Ø8       LDA   #$Ø8
Ø8C2-   AØ E8       LDY   #$E8
Ø8C4-   2Ø D9 Ø3    JSR   $Ø3D9
Ø8C7-   AC ED Ø8    LDY   $Ø8ED
Ø8CA-   88          DEY
Ø8CB-   1Ø Ø5       BPL   $Ø8D2
Ø8CD-   AØ ØF       LDY   #$ØF
Ø8CF-   CE EC Ø8    DEC   $Ø8EC
Ø8D2-   8C ED Ø8    STY   $Ø8ED
Ø8D5-   CE F1 Ø8    DEC   $Ø8F1
Ø8D8-   CE E1 Ø8    DEC   $Ø8E1
Ø8DB-   DØ E3       BNE   $Ø8CØ
Ø8DD-   6Ø          RTS

*8EØ.8FF

Ø8EØ- ØØ Ø5 ØØ ØØ ØØ ØØ ØØ ØØ
         ^^
    sector count

Ø8E8- Ø1 6Ø Ø1 ØØ ØØ Ø4 FB Ø8
         ^^ ^^    ^^ ^^
         S6 D1    TØ S4

Ø8FØ- ØØ 27 ØØ ØØ Ø2 ØØ FE 6Ø
      ^^^^^       ^^
     address    write

Ø8F8- Ø1 ØØ ØØ ØØ Ø1 EF D8 ØØ

*BSAVE WRITE BOOT1,A$8CØ,L$4Ø

; load decrypted bootloader
*BLOAD BOOTØ,A$23ØØ
*BLOAD BOOT1 DECRYPTED,A$24ØØ

; disable decryption loop (we’ll still
; let it run, but this disables the
; actual “STA” instruction at $Ø475,
; so it won’t overwrite the already-
; decrypted code we’re about to write
; to disk)
*2475:24

; normalize RWTS by fixing the third
; nibble of the data prologue
*2621:AD

[S6,D1=crack-in-progress (the partial
  copy I made with Locksmith Fast Disk
  Backup)]

; write decrypted+patched bootloader
; to disk
*8CØG
...write write write...
```

Now I need to convert the protected tracks that use the non-standard data prologue. Super Demuffin only converts

whole tracks, but track $15 and $22 are a mix of protected and unprotected sectors. I'll need to make an RWTS file and pump it through Advanced Demuffin.

```
[S6,D1=DOS 3.3 system master]

]PR#6
...
]CALL -151

; copy RWTS
*38ØØ<B8ØØ.BFFFM

; modify third nibble of data prologue
*38FC:AA

*BSAVE RWTS,A$38ØØ,L$8ØØ,S5,D1

*BRUN ADVANCED DEMUFFIN 1.5,S5,D1

[S6,D1=original disk]
[S6,D2=crack-in-progress]

[press "5" to switch to slot 5]

[press "R" to load a new RWTS module]
  --> At $B8, load "RWTS" from drive 1

[press "6" to switch to slot 6]

[press "C" to convert disk]

[press "Y" to change default values]
```

--v--

```
ADVANCED DEMUFFIN 1.5    (C) 1983, 2Ø14
ORIGINAL BY THE STACK    UPDATES BY 4AM
=======================================


INPUT ALL VALUES IN HEX


SECTORS PER TRACK? (13/16) 16

START TRACK: $15        <-- change this
START SECTOR: $Ø3       <-- change this

END TRACK: $22
END SECTOR: $ØD         <-- change this

INCREMENT: 1

MAX # OF RETRIES: Ø

COPY FROM DRIVE 1
TO DRIVE: 2
=======================================
16SC $15,$Ø3-$22,$ØD BY$Ø1 S6,D1->S6,D2
```

--^--

And here we go...

--v--

```
ADVANCED DEMUFFIN 1.5    (C) 1983, 2Ø14
ORIGINAL BY THE STACK    UPDATES BY 4AM
=======PRESS ANY KEY TO CONTINUE=======
TRK:                     ..............
+.5:
    Ø123456789ABCDEFØ123456789ABCDEFØ12
SCØ:                      .............
SC1:                      .............
SC2:                      .............
SC3:                     ..............
SC4:                     ..............
SC5:                     ..............
SC6:                     ..............
SC7:                     ..............
SC8:                     ..............
SC9:                     ..............
SCA:                     ..............
SCB:                     ..............
SCC:                     ..............
SCD:                     ..............
SCE:                     .............
SCF:                     .............
=======================================
16SC $15,$Ø3-$22,$ØD BY$Ø1 S6,D1->S6,D2
```

--^--

And another conversion with the same RWTS, this time for that one protected sector on track $11.

--v--

```
ADVANCED DEMUFFIN 1.5    (C) 1983, 2Ø14
ORIGINAL BY THE STACK    UPDATES BY 4AM
=======PRESS ANY KEY TO CONTINUE=======
TRK:                 .
+.5:
    Ø123456789ABCDEFØ123456789ABCDEFØ12
SCØ:
SC1:
SC2:
SC3:
SC4:
SC5:
SC6:
SC7:                 .
SC8:
SC9:
SCA:
SCB:
SCC:
SCD:
SCE:
SCF:
=======================================
16SC $11,$Ø7-$11,$Ø7 BY$Ø1 S6,D1->S6,D2
```

--^--

```
[S6,D1=crack-in-progress]

]PR#6
..."UNABLE TO LOAD GAME"...
```

There is an explicit protection check somewhere after it reads T11,SØ7. Now that I've decrypted the bootloader and patched the RWTS, it's finally able to get far enough to detect that I'm running an unauthorized copy.

In other words, I now have a COPYA-able copy that is just as broken as my EDD bit copy. This we call progress.

~

Chapter 6
In Which We Find That Which Does Not Wish To Be Found

Picking up where we left off, at $Ø6C1:

```
]PR#5
...
]BLOAD BOOT1 DECRYPTED,A$24ØØ
]CALL -151

*26C1L

26C1-   A6 2B       LDX   $2B

; I'll return to this in a moment
26C3-   2Ø FA Ø6    JSR   $Ø6FA

; roll one bit of the accumulator into
; the carry
26C6-   2A          ROL

; if carry is clear, skip over the
; following code
26C7-   9Ø 2F       BCC   $26F8

; turn off the drive motor
26C9-   A4 2B       LDY   $2B
26CB-   B9 88 CØ    LDA   $CØ88,Y

; clear text page 2
26CE-   A9 AØ       LDA   #$AØ
26DØ-   A2 ØØ       LDX   #$ØØ
26D2-   9D ØØ Ø8    STA   $Ø8ØØ,X
26D5-   E8          INX
26D6-   DØ FA       BNE   $26D2
26D8-   EE D4 Ø6    INC   $Ø6D4
26DB-   AC D4 Ø6    LDY   $Ø6D4
26DE-   CØ ØC       CPY   #$ØC
26EØ-   DØ FØ       BNE   $26D2

; display a message
26E2-   AØ ØØ       LDY   #$ØØ
26E4-   B9 89 Ø7    LDA   $Ø789,Y
26E7-   99 ØØ Ø8    STA   $Ø8ØØ,Y
26EA-   C8          INY
26EB-   CØ 28       CPY   #$28
26ED-   DØ F5       BNE   $26E4

; show it
26EF-   AD 51 CØ    LDA   $CØ51
26F2-   AD 55 CØ    LDA   $CØ55

; jump to The Badlands
26F5-   4C DF Ø7    JMP   $Ø7DF

; execution continues here (from $Ø6C7)
26F8-   6Ø          RTS
```

Here's the message that is displayed at $Ø6E2:

```
*FC58G N 4ØØ<2789.27BØM

           UNABLE TO LOAD GAME
```

Don't look now, but I think we just found the copy protection.

```
*26FAL

26FA-   BD 89 CØ    LDA   $CØ89,X
26FD-   A9 56       LDA   #$56
26FF-   85 11       STA   $11
27Ø1-   DØ Ø1       BNE   $27Ø4
27Ø3-   DØ C6       BNE   $26CB
27Ø5-   12          ???
```

Oh joy, more obfuscated code. The "BNE" at $Ø7Ø1 unconditionally branches into the middle of the next instruction, which confuses the monitor's built-in disassembler.

```
*27Ø3:EA
*26FAL

26FA-   BD 89 CØ    LDA   $CØ89,X
26FD-   A9 56       LDA   #$56
26FF-   85 11       STA   $11
27Ø1-   DØ Ø1       BNE   $27Ø4
27Ø3-   EA          NOP
27Ø4-   C6 12       DEC   $12
27Ø6-   FØ Ø3       BEQ   $27ØB
27Ø8-   DØ ØA       BNE   $2714
27ØA-   DØ C6       BNE   $26D2
27ØC-   11 DØ       ORA   ($DØ),Y
```

Oops, there's another one. $Ø7Ø6 is a branch to $Ø7ØB, which is shown in the middle of an instruction again.

```
*27ØA:EA
```

I'll spare you the gory details, but there are a few more of these. Just a few. 16. There are 16 more.

```
*2713:EA
*271B:EA
*2722:EA
*272A:EA
*2731:EA
*2739:EA
*274Ø:EA
*274A:EA
*2751:EA
*2754:EA
*275E:EA
*2766:EA
*276E:EA
*2776:EA
*277D:EA
*2786:EA
```

Here's the final, as-unobfuscated-as-I-can-make-it copy protection routine:

```
*26FAL

; turn on drive motor
26FA-   BD 89 CØ    LDA   $CØ89,X

; set up Death Counters
26FD-   A9 56       LDA   #$56
26FF-   85 11       STA   $11
27Ø1-   DØ Ø1       BNE   $27Ø4
27Ø3-   EA          NOP

; this is actually uninitialized, but
; it doesn't much matter because it's
; just the low byte of the 16-bit Death
; Counter
27Ø4-   C6 12       DEC   $12
27Ø6-   FØ Ø3       BEQ   $27ØB
```

```
27Ø8-   DØ ØA       BNE   $2714
27ØA-   EA          NOP
27ØB-   C6 11       DEC   $11
27ØD-   DØ Ø5       BNE   $2714

; if Death Counter hits Ø, set A=Ø1 and
; exit
27ØF-   A9 Ø1       LDA   #$Ø1
2711-   DØ 74       BNE   $2787
2713-   EA          NOP

; find standard address prologue
; (D5 AA 96)
2714-   BD 8C CØ    LDA   $CØ8C,X
2717-   1Ø FB       BPL   $2714
2719-   DØ Ø1       BNE   $271C
271B-   EA          NOP
271C-   C9 D5       CMP   #$D5
271E-   FØ Ø3       BEQ   $2723
272Ø-   DØ E2       BNE   $27Ø4
2722-   EA          NOP
2723-   BD 8C CØ    LDA   $CØ8C,X
2726-   1Ø FB       BPL   $2723
2728-   DØ Ø1       BNE   $272B
272A-   EA          NOP
272B-   C9 AA       CMP   #$AA
272D-   FØ Ø3       BEQ   $2732
272F-   DØ D3       BNE   $27Ø4
2731-   EA          NOP
2732-   BD 8C CØ    LDA   $CØ8C,X
2735-   1Ø FB       BPL   $2732
2737-   DØ Ø1       BNE   $273A
2739-   EA          NOP
273A-   C9 96       CMP   #$96
273C-   FØ Ø3       BEQ   $2741
273E-   DØ C4       BNE   $27Ø4
274Ø-   EA          NOP

; skip over an $FF nibble
2741-   AØ ØA       LDY   #$ØA
2743-   BD 8C CØ    LDA   $CØ8C,X
2746-   1Ø FB       BPL   $2743
2748-   DØ Ø1       BNE   $274B
274A-   EA          NOP
274B-   C9 FF       CMP   #$FF
274D-   DØ Ø3       BNE   $2752
274F-   FØ B3       BEQ   $27Ø4
2751-   EA          NOP
2752-   FØ Ø1       BEQ   $2755
2754-   EA          NOP

; Read data latch exactly once (no BPL
; loop here!) and make sure its value
; is correct. We're out of sync here
; because of all the branching, so the
; exact value of the data latch depends
; on timing bit after the $FF nibble.
; This is where my EDD bit copy failed.
2755-   BD 8C CØ    LDA   $CØ8C,X
2758-   C9 A5       CMP   #$A5
275A-   DØ A8       BNE   $27Ø4
275C-   FØ Ø1       BEQ   $275F
275E-   EA          NOP

; calculate a checksum on the following
; nibbles
275F-   BD 8C CØ    LDA   $CØ8C,X
2762-   1Ø FB       BPL   $275F
2764-   DØ Ø1       BNE   $2767
2766-   EA          NOP
2767-   85 1Ø       STA   $1Ø
2769-   88          DEY
276A-   DØ Ø3       BNE   $276F
276C-   FØ 1Ø       BEQ   $277E
276E-   EA          NOP
276F-   BD 8C CØ    LDA   $CØ8C,X
2772-   1Ø FB       BPL   $276F
2774-   DØ Ø1       BNE   $2777
2776-   EA          NOP
2777-   45 1Ø       EOR   $1Ø
2779-   DØ EC       BNE   $2767
277B-   FØ EA       BEQ   $2767
277D-   EA          NOP

; final checksum must be $6Ø
277E-   A5 1Ø       LDA   $1Ø
278Ø-   49 6Ø       EOR   #$6Ø
2782-   DØ 8Ø       BNE   $27Ø4
2784-   FØ Ø1       BEQ   $2787
2786-   EA          NOP

; on exit, A=ØØ on success (after
; falling through) or Ø1 on failure
; (coming from $Ø711)
2787-   6Ø          RTS
```

This routine always returns to the caller, and the accumulator indicates success (Ø) or failure (1).

At $Ø6C6, I can change the “ROL” to a “CLC”. Then the “BCC” at $Ø6C7 will unconditionally branch to $Ø6F8, as if the protection routine had passed.

TØØ,SØ3,$C6 change 2A to 18

```
]PR#6
...offers input selection, then hangs
   with the drive motor on...
```

That's actually a lot of progress. I can now calibrate my joystick before the game tells me to go f--- myself.

~

Chapter 7
In Which I'd Like To Add You To My Professional Network Of Linked Catalog Sectors

Let's go back to that “unreadable” sector on track $11, the one that looks exactly like a DOS 3.3 catalog sector, because it is a DOS 3.3 catalog sector. It's just not linked into the VTOC.

T11,SØØ currently points to T11,SØF as the first catalog sector, but all of the “files” in that catalog sector are fake. What if I changed it to point to sector $Ø7 instead?

T11,SØØ,$Ø2 change ØF to Ø7

```
]PR#5
...
]CATALOG,S6,D1

C1983 DSR^C#254
324 FREE

 A ØØ2 HELLO
 B 181 BURGERTIME
```

```
 B Ø38 MDSADJ
```

Well would you look at that.

```
]BRUN BURGERTIME
...crashes...
```

It was worth a shot.

```
]PR#5
...
]BLOAD BURGERTIME,S6,D1
```

This actually works, but only because my work disk is running Diversi-DOS 64K which relocates most of DOS to the language card. The file starts at $ØCØØ and is $B2ØØ in length. Whatever is at $ØCØØ isn't code, which explains why I couldn't just BRUN it. But this is definitely (at least part of) the game code, because switching to the hi-res graphics screen shows the game's title screen.

```
]BRUN MDSADJ,S6,D1
...displays input selection screen,
   allows me to actually calibrate my
   joystick, then hangs with the drive
   motor on...
```

These files are definitely not fake. The bootloader is actually using this (previously hidden, but well-formed) catalog sector to load the input selection routine (MDSADJ), then doing it again to load the actual game. But something is getting stuck between those two, so I need to look at MDSADJ to find out what's going on.

```
]BLOAD MDSADJ
]CALL -151

*BF55.BF56

BF55- ØØ 64    ; start address

*64ØØL

64ØØ-   A9 9D       LDA   #$9D    ;A=9D
64Ø2-   85 5Ø       STA   $5Ø     ; =9D
64Ø4-   DØ Ø1       BNE   $64Ø7   ;yes
64Ø6-   75 A9       ADC   $A9,X
64Ø8-   86 A2       STX   $A2
64ØA-   ØØ          BRK
```

Right out of the gate, I can tell this is going to be fun(*). The branch at $64Ø4 jumps into the middle of the next instruction, which confuses the monitor disassembler.

(*) not guaranteed, actual fun may vary

```
*64Ø7L

64Ø7-   A9 86       LDA   #$86    ;A=86
64Ø9-   A2 ØØ       LDX   #$ØØ    ;X=ØØ
64ØB-   FØ Ø1       BEQ   $64ØE   ;yes
64ØD-   24 95       BIT   $95
```

And again.

```
*64ØEL

64ØE-   95 21       STA   $21,X
641Ø-   86 2Ø       STX   $2Ø
```

Now ($2Ø) points to $86ØØ.

```
6412-   A9 Ø1       LDA   #$Ø1    ;A=Ø1
6414-   DØ Ø1       BNE   $6417   ;yes
6416-   DØ 6C       BNE   $6484
```

And again.

```
*6417L

6417-   6C 2Ø ØØ    JMP   ($ØØ2Ø) ;86ØØ

*86ØØL

; decrypt everything in the file we
; just loaded ($64ØØ..$85FF)
86ØØ-   A2 22       LDX   #$22
86Ø2-   AØ ØØ       LDY   #$ØØ
86Ø4-   B9 ØØ 85    LDA   $85ØØ,Y
86Ø7-   59 ØØ 84    EOR   $84ØØ,Y
86ØA-   49 Ø1       EOR   #$Ø1
86ØC-   99 ØØ 85    STA   $85ØØ,Y
86ØF-   C8          INY
861Ø-   DØ F2       BNE   $86Ø4
8612-   CE Ø6 86    DEC   $86Ø6
8615-   CE Ø9 86    DEC   $86Ø9
8618-   CE ØE 86    DEC   $86ØE
861B-   CA          DEX
861C-   DØ E6       BNE   $86Ø4

; and continue with decrypted code
861E-   4C 1D 64    JMP   $641D
```

Luckily (for me), this decryption loop is self-contained and does not rely on itself as a decryption key. I should be able to put an "RTS" at $861E and run it from the monitor.

```
*861E:6Ø
*86ØØG
*641DL

641D-   8D 5A DB    STA   $DB5A
642Ø-   49 7Ø       EOR   #$7Ø
6422-   E8          INX
6423-   43          ???
6424-   4F          ???
6425-   C9 28       CMP   #$28
6427-   BF          ???
```

I've missed something.

Wait. On the last pass, it's decrypting $64ØØ..$64FF by EOR'ing it with the page under that, which is $63ØØ..$63FF. Which is not part of this file. Which means that this decryption loop is not self-contained after all; it depends on the value of the page at $63ØØ.

Backtracking the boot, the only thing that ever touched $63ØØ was the routine at $Ø4ØA that wiped all of main memory with zeroes.

```
*BLOAD MDSADJ
```

```
*6300:00 N 6301<6300.63FEM
*861E:60
*8600G
*641DL

641D-   20 5D 64    JSR   $645D
6420-   20 00 65    JSR   $6500
6423-   20 2C 64    JSR   $642C
6426-   20 00 85    JSR   $8500
6429-   4C B0 68    JMP   $68B0
642C-   A6 2B       LDX   $2B
642E-   BD 8E C0    LDA   $C08E,X
6431-   BD 8C C0    LDA   $C08C,X
6434-   BD 89 C0    LDA   $C089,X
```

Bingo.

```
*BSAVE MDSADJ DECRYPTED,A$6400,L$2400,
 S5,D1
```

~

Chapter 8
In Which We Discover A Decryption Most Foul (Again)

Continuing from $641D...

```
*641DL

641D-   20 5D 64    JSR   $645D

*645DL

; wipe language card
645D-   AD 83 C0    LDA   $C083
6460-   AD 83 C0    LDA   $C083
6463-   A0 00       LDY   #$00
6465-   84 00       STY   $00
6467-   A9 D0       LDA   #$D0
6469-   85 01       STA   $01
646B-   91 00       STA   ($00),Y
646D-   C8          INY
646E-   D0 FB       BNE   $646B
6470-   E6 01       INC   $01
6472-   D0 F7       BNE   $646B

; back to ROM
6474-   AD 81 C0    LDA   $C081

; and over here, then out
6477-   20 80 64    JSR   $6480
647A-   60          RTS

*6480L

; Scan all peripheral (slot) ROMs, test
; whether the first byte of the slot
; ROM changes over a short period of
; time, and if not, save it to an array
; at $BF73. (I'm assuming this is done
; again later, and if the values don't
; match, the game knows that it's not
; running on the same machine -- i.e.
; that someone used a memory capture
; card to save the game code to disk
; then reload it without going through
; this bootloader.)
6480-   A2 07       LDX   #$07
6482-   A9 00       LDA   #$00
6484-   8D 9B 64    STA   $649B
6487-   8D 9E 64    STA   $649E
648A-   8A          TXA
648B-   18          CLC
648C-   69 C0       ADC   #$C0
648E-   8D 9C 64    STA   $649C
6491-   8D 9F 64    STA   $649F
6494-   A9 0A       LDA   #$0A
6496-   85 A3       STA   $A3
6498-   A0 00       LDY   #$00
649A-   AD 00 C0    LDA   $C000
649D-   CD 00 C0    CMP   $C000
64A0-   D0 0E       BNE   $64B0
64A2-   88          DEY
64A3-   D0 F8       BNE   $649D
64A5-   C6 A3       DEC   $A3
64A7-   D0 EF       BNE   $6498
64A9-   9D 73 BF    STA   $BF73,X
64AC-   CA          DEX
64AD-   D0 D3       BNE   $6482
64AF-   60          RTS
64B0-   A9 00       LDA   #$00
64B2-   4C A9 64    JMP   $64A9
```

Since I'm planning to retain this entire bootloader (albeit without the decrypt-y bits or the protect-y bits), I won't bother to disable it now. But if it turns out I can't retain the bootloader, I'll need to revisit it.

Continuing from $6420...

```
*6420L

; (not shown) This is the actual input
; selection routine that sets joystick
; or keyboard mode. It returns when
; the user has made a selection or time
; has run out.
6420-   20 00 65    JSR   $6500
```

Once that returns, things suddenly get a lot more interesting (for me, at least).

```
6423-   20 2C 64    JSR   $642C

*642CL

; turn on drive motor
642C-   A6 2B       LDX   $2B
642E-   BD 8E C0    LDA   $C08E,X
6431-   BD 8C C0    LDA   $C08C,X
6434-   BD 89 C0    LDA   $C089,X

; clear hi-res screen 2
6437-   A9 00       LDA   #$00
6439-   85 00       STA   $00
643B-   A0 40       LDY   #$40
643D-   84 01       STY   $01
643F-   A8          TAY
6440-   91 00       STA   ($00),Y
6442-   C8          INY
6443-   D0 FB       BNE   $6440
6445-   E6 01       INC   $01
6447-   A6 01       LDX   $01
6449-   E0 60       CPX   #$60
644B-   D0 F3       BNE   $6440

; and show it (blank)
644D-   AD 50 C0    LDA   $C050
6450-   AD 52 C0    LDA   $C052
6453-   AD 55 C0    LDA   $C055
```

```
6456-   AD 57 CØ    LDA   $CØ57

; clear keyboard strobe
6459-   AD 1Ø CØ    LDA   $CØ1Ø
645C-   6Ø          RTS
```

Continuing from $6426...

```
*6426L

6426-   2Ø ØØ 85    JSR   $85ØØ

*85ØØL

; oh God, here we go again
85ØØ-   A9 ØØ       LDA   #$ØØ    ;A=ØØ
85Ø2-   85 82       STA   $82     ; =ØØ

; save stack pointer again
85Ø4-   BA          TSX
85Ø5-   8A          TXA
85Ø6-   49 BC       EOR   #$BC
85Ø8-   85 5Ø       STA   $5Ø

85ØA-   29 ØØ       AND   #$ØØ    ;A=ØØ
85ØC-   AA          TAX           ;X=ØØ
85ØD-   AØ D6       LDY   #$D6    ;Y=D6
85ØF-   49 3A       EOR   #$3A    ;A=3A
8511-   84 81       STY   $81     ; =D6
```

Now ($81) points to $ØØD6. That address was set to $5C (at $Ø816) and has never been touched since.

```
; X=ØØ, so this is taking ($81), which
; is zp$D6, which is $5C, then EOR'ing
; it with A, which is $3A
8513-   41 81       EOR   ($81,X) ;A=66
8515-   85 Ø1       STA   $Ø1     ; =66
```

Now ($ØØ) points to $66ØØ.

```
8517-   98          TYA           ;A=D6
8518-   49 D2       EOR   #$D2    ;A=Ø4
851A-   85 81       STA   $81     ; =Ø4
```

Now ($81) points to $ØØØ4.

```
851C-   49 Ø4       EOR   #$Ø4    ;A=ØØ

; ($81) points to $ØØØ4 and X=ØØ, so
; this sets zp$Ø4 to $ØØ.
851E-   81 81       STA   ($81,X) ; =ØØ
852Ø-   E6 81       INC   $81
```

Now ($81) points to $ØØØ5.

```
8522-   38          SEC
8523-   E9 7D       SBC   #$7D    ;A=83
8525-   1Ø FØ       BPL   $8517   ;nope

; ($81) points to $ØØØ5 and X=ØØ, so
; this sets zp$Ø5 to $83, so now ($Ø4)
; points to $83ØØ.
8527-   81 81       STA   ($81,X) ; =83
8529-   49 83       EOR   #$83    ;A=ØØ
852B-   85 ØØ       STA   $ØØ     ; =ØØ
852D-   85 51       STA   $51     ; =ØØ
852F-   49 85       EOR   #$85    ;A=85
8531-   FØ Ø2       BEQ   $8535   ;nope
8533-   85 Ø3       STA   $Ø3     ; =85
8535-   8A          TXA           ;A=ØØ
8536-   85 Ø2       STA   $Ø2     ; =ØØ
```

Now ($Ø2) points to $85ØØ.

```
8538-   18          CLC
8539-   69 D6       ADC   #$D6    ;A=D6
853B-   85 81       STA   $81     ; =86
```

Now ($81) points to $ØØD6 again.

```
853D-   4A          LSR           ;A=6B
853E-   AA          TAX           ;X=6B

; $16 + $6B = $81, and ($81) points to
; $ØØD6, and zp$D6 is still $5C.
853F-   A1 16       LDA   ($16,X) ;A=5C
8541-   49 5C       EOR   #$5C    ;A=ØØ
8543-   A8          TAY           ;Y=ØØ
8544-   49 Ø4       EOR   #$Ø4    ;A=Ø4
8546-   FØ E5       BEQ   $852D   ;nope
8548-   85 Ø8       STA   $Ø8     ; =Ø4
854A-   49 7A       EOR   #$7A    ;A=7E
854C-   88          DEY           ;Y=FF

; EOR with ($Ø2),Y = $85ØØ,Y = $85FF
854D-   51 Ø2       EOR   ($Ø2),Y

; EOR with ($Ø4),Y = $83ØØ,Y = $83FF
854F-   51 Ø4       EOR   ($Ø4),Y

; SBC ($ØØ),Y = $66ØØ,Y = $66FF
8551-   38          SEC
8552-   F1 ØØ       SBC   ($ØØ),Y

; store it in ($Ø4),Y = $83ØØ,Y = $83FF
8554-   91 Ø4       STA   ($Ø4),Y

; done decrypting this page?
8556-   CØ ØØ       CPY   #$ØØ

; nope, branch back to finish it
8558-   DØ F2       BNE   $854C
```

That branch goes to $854C, which is a "DEY" instruction, so we're decrypting memory from the top down. This is the same basic pattern as the earlier decryption routine at $Ø421, which was equally insane.

```
855A-   84 81       STY   $81     ; =ØØ

; decrement page count (initialized to
; Ø4 at $8548)
855C-   C6 Ø8       DEC   $Ø8

; jump forward if we're done decrypting
855E-   FØ Ø5       BEQ   $8565

; otherwise decrement the target page
; and loop back to decrypt it
856Ø-   C6 Ø5       DEC   $Ø5
8562-   4C 4C 85    JMP   $854C
```

The first pass decrypts $83FF..$83ØØ against $85FF..$85ØØ and $66FF..$66ØØ. The second pass decrypts $82FF..$82ØØ; then $81FF..$81ØØ; then $8ØFF..$8ØØØ. Then zp$Ø8 hits Ø and we branch forward to $8565.

```
; get some byte from the stack
8565-   A6 5Ø       LDX   $5Ø
8567-   BD ØØ Ø1    LDA   $Ø1ØØ,X
```

```
856A-   85 Ø3        STA   $Ø3
856C-   49 47        EOR   #$47
856E-   6Ø           RTS
```

This routine is almost self-contained. The only thing it relies on to be pre-initialized is zero page $D6, which must be $5C. Other than that, I can run it directly from the monitor.

```
*D6:5C N 85ØØG

*8ØØØL

8ØØØ-   A1 A9        LDA   ($A9,X)
8ØØ2-   Ø9 85        ORA   #$85
8ØØ4-   25 85        AND   $85
8ØØ6-   Ø1 A9        ORA   ($A9,X)
8ØØ8-   ØØ           BRK
8ØØ9-   85 24        STA   $24
8ØØB-   85 83        STA   $83
8ØØD-   85 ØØ        STA   $ØØ
8ØØF-   2Ø CØ Ø4     JSR   $Ø4CØ
```

That *almost* looks like real code. Wait, I see it now.

```
*8ØØ1L

8ØØ1-   A9 Ø9        LDA   #$Ø9
8ØØ3-   85 25        STA   $25
8ØØ5-   85 Ø1        STA   $Ø1
8ØØ7-   A9 ØØ        LDA   #$ØØ
8ØØ9-   85 24        STA   $24
8ØØB-   85 83        STA   $83
8ØØD-   85 ØØ        STA   $ØØ
8ØØF-   2Ø CØ Ø4     JSR   $Ø4CØ
```

There we go.

```
*BSAVE MDSADJ DECRYPTED TWICE,A$64ØØ,
 L$24ØØ,S5,D1
```

~

Chapter 9
In Which We'll Think About It After, Like An After-Thought

Continuing from... um... $6429...

```
*6429L

6429-   4C BØ 68     JMP   $68BØ

*68BØL

; copy newly decrypted code to the text
; page (overwriting the original RWTS)
68BØ-   A9 ØØ        LDA   #$ØØ
68B2-   85 ØØ        STA   $ØØ
68B4-   A9 Ø4        LDA   #$Ø4
68B6-   85 Ø1        STA   $Ø1
68B8-   A9 ØØ        LDA   #$ØØ
68BA-   85 Ø2        STA   $Ø2
68BC-   A9 8Ø        LDA   #$8Ø
68BE-   85 Ø3        STA   $Ø3
68CØ-   A2 Ø5        LDX   #$Ø5
68C2-   AØ ØØ        LDY   #$ØØ
68C4-   B1 Ø2        LDA   ($Ø2),Y
68C6-   91 ØØ        STA   ($ØØ),Y
68C8-   C8           INY
68C9-   DØ F9        BNE   $68C4
68CB-   CA           DEX
68CC-   FØ Ø7        BEQ   $68D5
68CE-   E6 Ø3        INC   $Ø3
68DØ-   E6 Ø1        INC   $Ø1
68D2-   4C C4 68     JMP   $68C4

; execution continue here (from $68CC)
; and we set yet another reset vector
68D5-   A9 96        LDA   #$96
68D7-   8D F2 Ø3     STA   $Ø3F2
68DA-   A9 Ø4        LDA   #$Ø4
68DC-   8D F3 Ø3     STA   $Ø3F3
68DF-   49 A5        EOR   #$A5
68E1-   8D F4 Ø3     STA   $Ø3F4

; destroy the code we just decrypted
; and copied (so the only real version
; is on the text page, which would be
; destroyed if an evil hacker tried to
; break into the monitor right now)
68E4-   2Ø ØØ 85     JSR   $85ØØ

; and continue inside the new code
68E7-   4C C1 Ø7     JMP   $Ø7C1
```

Since $8ØØØ+ is copied to $Ø4ØØ+, $Ø7C1 is currently at $83C1.

```
*83C1L

; I swear to God, it's another layer of
; encryption
83C1-   A2 ØØ        LDX   #$ØØ
83C3-   AØ Ø3        LDY   #$Ø3
83C5-   BD CØ Ø4     LDA   $Ø4CØ,X
83C8-   49 A5        EOR   #$A5
83CA-   9D CØ Ø4     STA   $Ø4CØ,X
83CD-   E8           INX
83CE-   DØ F5        BNE   $83C5
83DØ-   EE C7 Ø7     INC   $Ø7C7
83D3-   EE CC Ø7     INC   $Ø7CC
83D6-   88           DEY
83D7-   DØ EC        BNE   $83C5
83D9-   FØ Ø1        BEQ   $83DC
```

I can reproduce this easily enough. Rewriting it at $Ø3ØØ:

```
Ø3ØØ-   A2 ØØ        LDX   #$ØØ
Ø3Ø2-   AØ Ø3        LDY   #$Ø3
Ø3Ø4-   BD CØ 8Ø     LDA   $8ØCØ,X
Ø3Ø7-   49 A5        EOR   #$A5
Ø3Ø9-   9D CØ 8Ø     STA   $8ØCØ,X
Ø3ØC-   E8           INX
Ø3ØD-   DØ F5        BNE   $Ø3Ø4
Ø3ØF-   EE Ø6 Ø3     INC   $Ø3Ø6
Ø312-   EE ØB Ø3     INC   $Ø3ØB
Ø315-   88           DEY
Ø316-   DØ EC        BNE   $Ø3Ø4
Ø318-   6Ø           RTS

*3ØØG

*BSAVE MDSADJ DECRYPTED THRICE,A$64ØØ,
 L$24ØØ,S5,D1
```

I'm beginning to suspect that this disk is nothing more than an infinite series of decryption routines with a game bolted on as an afterthought.

~

```
            Chapter 10
  In Which We Find Two Of Everything


Continuing from... um... $83DC I guess
(branched from $83D9):

*83DCL

83DC-   A9 01       LDA   #$01    ;A=01
83DE-   85 36       STA   $36     ; =01
83E0-   A9 01       LDA   #$01    ;A=01
83E2-   38          SEC
83E3-   69 00       ADC   #$00    ;A=02
83E5-   F0 0E       BEQ   $83F5   ;nope
83E7-   10 03       BPL   $83EC   ;yes

83E9-   30 F0       BMI   $83DB   ;fake
83EB-   3D 2A 85    AND   $852A,X ;fake
83EE-   37          ???           ;fake

*83ECL

83EC-   2A          ROL           ;A=04
83ED-   85 37       STA   $37     ; =04

Now ($36) points to $0401.

83EF-   D0 01       BNE   $83F2   ;yes
83F1-   6C 6C 36    JMP   ($366C) ;fake
83F4-   00          BRK           ;fake
83F5-   4C 12 04    JMP   $0412   ;fake
83F8-   4C 00 04    JMP   $0400   ;fake
83FB-   4C 0F 06    JMP   $060F   ;fake

*83F2L

83F2-   6C 36 00    JMP   ($0036)

So execution continues at $0401.

The code that ends up at $0401 is
currently in memory at $8001, so I'm
going to leave it there and try not to
get too confused.

*8001L

8001-   A9 09       LDA   #$09
8003-   85 25       STA   $25
8005-   85 01       STA   $01
8007-   A9 00       LDA   #$00
8009-   85 24       STA   $24
800B-   85 83       STA   $83
800D-   85 00       STA   $00
800F-   20 C0 04    JSR   $04C0

That's at $80C0.

*80C0L

80C0-   20 6F 85    JSR   $856F

That's actually at $856F. Weird. This
second RWTS relies on MDSADJ still
being in memory. So many interlocking
dependencies...

*856FL

856F-   A5 51       LDA   $51     ;A=00
8571-   49 AA       EOR   #$AA    ;A=AA
8573-   49 AA       EOR   #$AA    ;A=00
8575-   85 36       STA   $36     ; =00
8577-   49 1C       EOR   #$1C    ; =1C
8579-   85 37       STA   $37     ; =1C
857B-   60          RTS

Now ($36) points to $1C00.

Continuing from $04C3, which is at
$80C3...

*80C3L

; looks like we're going to read that
; catalog sector again (T11,S07)
80C3-   A9 11       LDA   #$11
80C5-   85 80       STA   $80
80C7-   A9 07       LDA   #$07
80C9-   85 81       STA   $81
80CB-   85 82       STA   $82
80CD-   A5 D7       LDA   $D7
80CF-   8D 01 02    STA   $0201
80D2-   20 C3 06    JSR   $06C3

That's at $82C3.

*82C3L

; (not shown) This is the sector read
; routine. It follows the same pattern
; as the first RWTS -- moving the drive
; head, reading a sector. But it has
; its own disk read routine, which
; explains why my copy hung with the
; drive motor on. It's actually trying
; to read T11,S07 and failing because
; it's looking for the protected data
; prologue ("D5 AA AA") again.
82C3-   20 FA 04    JSR   $04FA

The new sector read routine starts at
$0604, which is in memory at $8204.

*8204L

8204-   18          CLC
8205-   08          PHP
8206-   BD 8C C0    LDA   $C08C,X
8209-   10 FB       BPL   $8206
820B-   49 D5       EOR   #$D5
820D-   D0 F7       BNE   $8206
820F-   BD 8C C0    LDA   $C08C,X
8212-   10 FB       BPL   $820F
8214-   C9 AA       CMP   #$AA
8216-   D0 F3       BNE   $820B
8218-   EA          NOP
8219-   BD 8C C0    LDA   $C08C,X
821C-   10 FB       BPL   $8219
821E-   C9 96       CMP   #$96
8220-   F0 09       BEQ   $822B
8222-   28          PLP
8223-   90 DF       BCC   $8204
8225-   49 AA       EOR   #$AA    <-- !
8227-   F0 25       BEQ   $824E
8229-   D0 D9       BNE   $8204

That $AA at $0626 needs to be changed
to $AD now that my disk uses standard
data prologues. (This explains why my
crack-in-progress hung with the drive
motor on -- it was trying to read the
disk with this second RWTS.)
```

Continuing from $Ø6C6 (after the call to $Ø4FA returns)...

```
*82C6L

82C6-   A6 2B       LDX   $2B
82C8-   2Ø EE Ø6    JSR   $Ø6EE
82CB-   29 Ø1       AND   #$Ø1
82CD-   AA          TAX
82CE-   BD D4 Ø6    LDA   $Ø6D4,X
82D1-   FØ Ø5       BEQ   $82D8
82D3-   6Ø          RTS
82D4-   2Ø ØØ 64    JSR   $64ØØ
82D7-   4C A9 8F    JMP   $8FA9
```

Lots going on here. I'll look at $Ø6EE in a minute, but check out what we're doing with the return code. If the low bit of the accumulator is Ø, X will end up being Ø and we'll read $Ø6D4 (=$2Ø), not take the "BEQ" branch, and exit via the "RTS" at $Ø6D3. But if the low bit of the accumulator is 1, X will end up being 1 and we'll read $Ø6D5 (=$ØØ), take the branch to $Ø6D8, and end up... doing what exactly?

```
*82D8L

82D8-   A9 8F       LDA   #$8F     ;A=8F
82DA-   18          CLC
82DB-   69 Ø7       ADC   #$Ø7     ;A=96
82DD-   85 36       STA   $36      ; =96
82DF-   A9 1Ø       LDA   #$1Ø     ;A=1Ø
82E1-   38          SEC
82E2-   E9 ØC       SBC   #$ØC     ;A=Ø4
82E4-   85 37       STA   $37      ; =Ø4
```

Now ($36) points to $Ø496.

```
82E6-   2A          ROL            ;A=Ø9
82E7-   2A          ROL            ;A=12
82E8-   85 39       STA   $39      ; =12
82EA-   6C 36 ØØ    JMP   ($ØØ36)
```

So we jump to $Ø496, which is in memory at $8Ø96.

```
*8Ø96L

8Ø96-   AØ Ø5       LDY   #$Ø5
8Ø98-   2Ø A2 Ø4    JSR   $Ø4A2

*8ØA2L

; wipe all memory, starting at the page
; given in A (=$Ø5)
8ØA2-   84 Ø1       STY   $Ø1
8ØA4-   A9 ØØ       LDA   #$ØØ
8ØA6-   85 ØØ       STA   $ØØ
8ØA8-   A8          TAY
8ØA9-   91 ØØ       STA   ($ØØ),Y
8ØAB-   C8          INY
8ØAC-   91 ØØ       STA   ($ØØ),Y
8ØAE-   C8          INY
8ØAF-   91 ØØ       STA   ($ØØ),Y
8ØB1-   C8          INY
8ØB2-   91 ØØ       STA   ($ØØ),Y
8ØB4-   C8          INY
8ØB5-   DØ F2       BNE   $8ØA9
8ØB7-   E6 Ø1       INC   $Ø1
8ØB9-   A6 Ø1       LDX   $Ø1
8ØBB-   EØ CØ       CPX   #$CØ
8ØBD-   DØ EA       BNE   $8ØA9
8ØBF-   6Ø          RTS
```

So it appears that $Ø6EE is a Very Important Routine, and it is vital that the low bit of the accumulator end up being Ø at the end of it. It's the same logic as the protection check! Or should I say, the *first* protection check, because it appears that there are two of them.

Four decryption loop.
Three hidden files.
Two nibble checks.
And a classic game on a floppy.

Merry Crackmas.

~

Chapter 11
And A Happy New Year

Let's go look at $Ø6EE.

```
*82EEL

82EE-   BD 89 CØ    LDA   $CØ89,X
82F1-   A9 56       LDA   #$56
82F3-   85 11       STA   $11
82F5-   DØ Ø1       BNE   $82F8
82F7-   DØ C6       BNE   $82BF
82F9-   12          ???
```

If this were a job, I'd quit.

```
*82F7:EA
*82FE:EA
*83Ø7:EA
*83ØF:EA
*8316:EA
*831E:EA
*8325:EA
*832D:EA
*8334:EA
*833E:EA
*8345:EA
*8348:EA
*8352:EA
*835A:EA
*8362:EA
*836A:EA
*8371:EA
*837A:EA

*82EEL

; turn on drive motor
82EE-   BD 89 CØ    LDA   $CØ89,X

; initialize Death Counters
82F1-   A9 56       LDA   #$56
82F3-   85 11       STA   $11
82F5-   DØ Ø1       BNE   $82F8
82F7-   EA          NOP
82F8-   C6 12       DEC   $12
82FA-   FØ Ø3       BEQ   $82FF
82FC-   DØ ØA       BNE   $83Ø8
82FE-   EA          NOP
82FF-   C6 11       DEC   $11
83Ø1-   DØ Ø5       BNE   $83Ø8
```

```
; if Death Counters hit Ø, load A=FF
; and exit
83Ø3-   A9 FF       LDA   #$FF
83Ø5-   DØ 7C       BNE   $8383
83Ø7-   EA          NOP

; find standard address prologue
; (D5 AA 96)
83Ø8-   BD 8C CØ    LDA   $CØ8C,X
83ØB-   1Ø FB       BPL   $83Ø8
83ØD-   DØ Ø1       BNE   $831Ø
83ØF-   EA          NOP
831Ø-   C9 D5       CMP   #$D5
8312-   FØ Ø3       BEQ   $8317
8314-   DØ E2       BNE   $82F8
8316-   EA          NOP
8317-   BD 8C CØ    LDA   $CØ8C,X
831A-   1Ø FB       BPL   $8317
831C-   DØ Ø1       BNE   $831F
831E-   EA          NOP
831F-   C9 AA       CMP   #$AA
8321-   FØ Ø3       BEQ   $8326
8323-   DØ D3       BNE   $82F8
8325-   EA          NOP
8326-   BD 8C CØ    LDA   $CØ8C,X
8329-   1Ø FB       BPL   $8326
832B-   DØ Ø1       BNE   $832E
832D-   EA          NOP
832E-   C9 96       CMP   #$96
833Ø-   FØ Ø3       BEQ   $8335
8332-   DØ C4       BNE   $82F8
8334-   EA          NOP

; skip over an $FF nibble
8335-   AØ ØA       LDY   #$ØA
8337-   BD 8C CØ    LDA   $CØ8C,X
833A-   1Ø FB       BPL   $8337
833C-   DØ Ø1       BNE   $833F
833E-   EA          NOP
833F-   C9 FF       CMP   #$FF
8341-   FØ Ø3       BEQ   $8346
8343-   DØ B3       BNE   $82F8
8345-   EA          NOP
8346-   FØ Ø1       BEQ   $8349
8348-   EA          NOP

; Read data latch exactly once (no BPL
; loop here!) and check its value.
; We're out of sync here because of all
; the branching, so the exact value of
; the data latch depends on timing bit
; after the $FF nibble. This is
; essentially the same technique as the
; first protection check, but done in a
; different way.
8349-   BD 8C CØ    LDA   $CØ8C,X
834C-   C9 Ø8       CMP   #$Ø8
834E-   BØ A8       BCS   $82F8
835Ø-   DØ Ø1       BNE   $8353
8352-   EA          NOP

; calculate a checksum on the following
; nibbles
8353-   BD 8C CØ    LDA   $CØ8C,X
8356-   1Ø FB       BPL   $8353
8358-   DØ Ø1       BNE   $835B
835A-   EA          NOP
835B-   85 1Ø       STA   $1Ø
835D-   88          DEY
835E-   DØ Ø3       BNE   $8363
836Ø-   FØ 1Ø       BEQ   $8372
8362-   EA          NOP
8363-   BD 8C CØ    LDA   $CØ8C,X
8366-   1Ø FB       BPL   $8363
8368-   DØ Ø1       BNE   $836B
836A-   EA          NOP
836B-   45 1Ø       EOR   $1Ø
836D-   DØ EC       BNE   $835B
836F-   FØ EA       BEQ   $835B
8371-   EA          NOP

; final checksum must be $6Ø
8372-   A5 1Ø       LDA   $1Ø
8374-   49 6Ø       EOR   #$6Ø
8376-   DØ 8Ø       BNE   $82F8
8378-   FØ Ø1       BEQ   $837B
837A-   EA          NOP

; wipe this entire protection check
; from memory
837B-   AØ 8E       LDY   #$8E
837D-   99 EE Ø6    STA   $Ø6EE,Y
838Ø-   88          DEY
8381-   DØ FA       BNE   $837D

; on exit, A=$ØØ on success (after
; falling through) or $FF on failure
; (coming from $Ø7Ø5)
8383-   6Ø          RTS
```

Revisiting the caller at $Ø6C6...

```
*82C6L

; execute protection check
82C6-   A6 2B       LDX   $2B
82C8-   2Ø EE Ø6    JSR   $Ø6EE

; get low bit
82CB-   29 Ø1       AND   #$Ø1

; Ø=success, 1=failure
82CD-   AA          TAX
82CE-   BD D4 Ø6    LDA   $Ø6D4,X

; failure path branches to The Badlands
82D1-   FØ Ø5       BEQ   $82D8

; success path returns to caller
82D3-   6Ø          RTS
82D4-  [2Ø ØØ]
```

To bypass this, I can change the "AND #$Ø1" at $Ø6CB to "AND #$ØØ", to act as if the protection check always passes.

To sum up: MDSADJ decrypts itself three separate times, revealing a second RWTS and a second protection check. Now that I've decrypted it (three times), I can patch the second RWTS and disable the second protection check.

```
; jump to the start of the code that
; was decrypted by the first decryption
; loop (which has already been done)
*64ØØ:4C 1D 64

; disable second decryption loop (also
; already done) by putting an "RTS" at
; the beginning of the routine at $85ØØ
*85ØØ:6Ø

; skip over third decryption loop (also
```

Acknowledgements

Great things in business are never done by one person.
They're done by a team of people. — Steve Jobs

The Editor acknowledges the work of the following people in the production of this book:

DON WORTH AND PIETER LECHNER

Steven Weyhrich, on his website apple2history.org, posted an email from Don Worth where he provided the story of how the book came to be written.

> Date: Wednesday, June 6, 2001 10:01 AM
> Subject: Re: Beneath Apple DOS
>
> In case you're interested, a friend and I disassembled DOS 3.1 (including RWTS – for which there was a listing in xeroxed form, and the File Manager). Vic Tolomei, who later became involved in writing books/software for the Exidy Sorcerer computer, prompted me to give it a try – I had been complaining about there being no documented way to open and read/write files from assembly language on the Apple II. Vic and I took apart my Disk II drive on his dining room table and twiddled the I/O addresses one by one to see what the arm did. Vic ran out a disassembly of DOS (using the disassembler in the Apple II ROM) and started marking it up – here and there, kind of like a crossword puzzle – I took what he started with and finished it off. When we were done, we had a fully commented listing of DOS.
>
> Vic and I put our listing in a binder, comb bound it, and made a couple of copies for friends. I sent one copy up to

> Apple just for fun. I got a somewhat panicked call from Woz. He wanted to know if we were planning to publish the source code – he was afraid it would damage Apple's copyright or some such. I promised we would not. I got personal Christmas cards from him for the next few years.
>
> I did some seminars at a computer store in Burbank, California on the API for the File Manager, which is where I met Pieter Lechner. We became friends, and a couple of years later decided to write Beneath Apple DOS together. Pieter wrote the low-level stuff about track formatting and RWTS. I wrote the higher level stuff about the File Manager. We included the comments from Vic's and my source listing with address ranges – but I didn't (technically) publish the source code. (A fine distinction – but by then I guess Woz no longer cared so much about it.) Since the File Manager API was still unknown, the book took off like a rocket. It's interesting to consider that, if Shepardson had finished the documentation for DOS as Apple had wanted them to, then there probably wouldn't have been a market for our book!
>
> Later on we took apart Integer BASIC and the Galfo Integer BASIC runtime code so that we could port the latter to Atari's and IBM PCs (which is how Beneath Apple Manor managed to move to those platforms) – but that's another story. We also disassembled ProDOS when that came out – a much bigger job than DOS. I can't even imagine trying to reverse engineer today's OS's – they are all produced from higher level language compilers anyway so they'd look pretty odd.

Don Worth has a personal website at http://worth.bol.ucla.edu/ and is active on the Apple II Enthusiasts page.

SEAN MCNAMARA

Sean "europlus" McNamara is an Apple II enthusiast in Sydney, Australia. An Apple II user since 1982, he began collecting Apple IIs in 1998 when he was gifted a IIgs, with europluses being his passion. He acquired more Apple IIs over the years to prevent them being junked, and started being involved in the local community in 2011.

In 2015, he attended Oz KFest in Melbourne and really hit the ground running in the local community, including giving a presentation on solid state storage solutions for the Apple II. A week after his return from Melbourne he hosted the first of what would become WOzFests - one day gatherings of Apple II enthusiasts.

Now held three times a year, WOzFests see the introduction of new hardware and software products, Skypes to Apple II enthusiasts and retrocomputer luminaries worldwide, software and hardware preservation/ repair, show-and-tells, chats, cider and pizza - an Apple II enthusiasts heaven!

4 AM*

It's tempting to rewrite history and give myself some noble purpose for starting this hobby, but in this case the truth makes for a better story. My parents bought themselves an Apple //e when I was 10, and it quickly came to dominate my leisure time. Pirated software was rampant, and I idolized the crackers whose names I saw flash and scroll on the "crack screens" of the games I traded with my friends. I also admired the few who documented their methods in cracking tutorials, initially distributed as BBS text files and later collated and redistributed on disk. I `PEEK`'d and `POKE`'d and `CALL`'d many late nights as a teenager, but I could never quite put it all together.

In late 2013, I acquired a real Apple //e and bought a few lots of original disks on eBay, mostly arcade games that I had acquired illicitly in my youth: *Sneakers*, *Repton*, *Dino Eggs*. To my surprise, the originals had more content than I remembered! Sneakers has an animated boot sequence. *Repton* has a multi-page introduction that explains the back story of the game. So I set out to create *complete* cracks that faithfully reproduced the original experience. I decided to document my methods because I enjoy technical writing, and because I had admired the classic crackers who had done so. (I decided to leave out the crack screens.)

One of those eBay lots had an educational game, *Ten Little Robots*. After cracking it, I couldn't find any copies of it online, which seemed odd. By 2014, surely everything has been cracked? Perhaps it was just misnamed or misfiled? Then I found another disk that seemed to be a first-time preservation. And another. And it slowly dawned on me that maybe not everything has been cracked.

I mentioned this to a friend of mine, and he set me straight. Preservation is driven by pirates, who are driven by ego but constrained by the technical limitations

---

* 4 am provided information for the ProDOS version of *Appendix B: Copy Protection*

of their era. In the 1980s, this meant storage space and modem speed. Nobody got kudos for cracking *Irregular Spanish Verbs in the Future Tense*, no BBS would waste the hard drive space to host it, and no user would sacrifice their phone line to download it. So it never got preserved in any form.

And even the things that did get cracked weren't fully preserved. Those same technical constraints led to a culture where the smallest version of a game always won. That meant stripping out the animated boot sequence, the title screen, the multi-page introduction, the cut scenes, anything deemed *non-essential* to the pirates. The *holy grail* was cutting away so much that you could distribute the game (or what was left of it) as a single file that could be combined with other unrelated games on a single floppy disk.

Thirty years later, that's exactly what I saw: half-preserved arcade games, a smattering of educational software, and virtually nothing else. I realized I could have a real impact while having just as much fun and just as much intellectual challenge. Along the way, I've discovered that educational software is rich with history, personality, humor, and technical achievement. It's been delightful.

Another early misunderstanding of mine was that copy protections were all unique. Nothing could be further from the truth! The most common protection schemes were the ones that were productized and resold to dozens of publishers. This was coordinated through the disk duplication houses, which offered copy protection as a *value add* on top of mastering the disks themselves. Publishers got the benefit of the latest and greatest copy protection without needing to play the cat-and-mouse game themselves. This led to some funny coincidences; my favorite is that *Math Blaster* and *Ultima IV* share the same copy protection. Two different publishers, two wildly different target audiences, but they had identical protection because protection was a product.

Of course, no one is inventing new Apple II protection schemes anymore. The cat-and-mouse game is over, and everyone lost. But make no mistake: copy protection works. I started cracking in earnest when I found one disk that had never been cracked. Since then, we've found thousands. With that many samples, we can start to look for patterns. Because most copy protection was an *add-on*, applied in bulk by automated processes, we can create similar processes to remove it in bulk.

And so we arrive at Passport. Passport is an automated disk verification and copy program. And when I say *automatic*, I mean it. Unlike classic nibble copiers, there are no parameters, no options, no knobs to fiddle. Also unlike classic copiers, the copy it produces is fully unprotected. No boot tracing, no

When you load too many ampersand routines into memory

sector editor patching on the back end. It's all built-in. Passport is a distillation of everything I've learned about cracking: every disk, every variation, every edge case.

This has completely changed my hobby. Passport ensures consistency. I don't worry about missing a patch or mistyping a hex value. I don't spend any time doing the grunt work that computers can do for me. If I find two disks with the same protection, I write a new Passport module to automate it. The same module that cracks *Ultima IV* also cracks *Math Blaster*. Remember, protection was productized. If I've found two samples of a protection scheme, there are likely twenty more. They're out there, undiscovered, rotting away on physical media.

Automation frees me to look beyond the bits. I can spend more time on in-depth write-ups of protection schemes that can't be automated. I can take screenshots and make boot videos to show off all the wonderful educational software I've discovered. The copy protection is the least interesting part of these disks. It's just the part that prevented us from studying all the other parts.

Passport is online at `https://archive.org/details/Passport4am`

Glossary

access time. The time required to locate and read or write data on a direct access storage device, such as a diskette drive.

address. The numeric location of a piece of data in memory, usually given as a hexadecimal number from $0000 to $FFFF (65,535 decimal). A disk address is the location of a data sector expressed in terms of its track and sector numbers.

algorithm. A sequence of steps which may be performed by a program or other process, which will produce a given result.

alphanumeric. An alphabetic character (A-Z) or a numeric digit (0-9). In the past, the term referred to the class of all characters and digits.

analog. Having a value which is continuous, such as a voltage or electrical resistance, as opposed to digital.

AND. The logical process of determining whether two bits are both ones. 0 AND 1 results in 0 (false), 1 AND 1 results in 1 (true).

arm. The portion of a disk drive which suspends the read/write head over the disk's surface. The arm can be moved radially to allow access to different tracks.

ASCII. American Standard Code for Information Interchange. A hexadecimal to character conversion code assignment, such that the 256 possible values of a single byte may each represent a alphabetic, numeric, special, or control character. ASCII is used when interfacing to peripherals, such as keyboards, printers, or video text displays.

assembly language. Also known as machine language. The native programming language of the individual computer. Assembly language is oriented to the machine, and is not humanized as is BASIC, PASCAL, or FORTRAN. An assembler is used to convert assembly language statements to an executable program.

backup. The process of making a copy of a program or data against the possibility of its accidental loss or destruction.

bank switched memory. Also called the language card. An additional 16K of memory which may only be accessed by "throwing" hardware switches to cause portions of the bank switched memory to temporarily replace the Monitor ROM memory in the machine. This is necessary because an Apple can only address 64K, and all addresses are already used with 48K, 4K of I/O and 12K of Monitor ROM.

base. The number system in use. Decimal is base 10, since each digit represents a power of 10 (1,10,100,...). Hexadecimal is base 16 (1,16,256,...). Binary is base 2 (1,2,4,8,...).

BI (BASIC Interpreter). Also called the BASIC System Program. The BI accepts user commands such as CATALOG and LOAD, and translates them into calls to the ProDOS Machine Language Interface (MLI).

binary. A number system based upon powers of 2. Only the digits 0 and 1 are used. For example, 101 in binary is 1 units digit, 0 twos, and 1 fours, or 5 in decimal.

bit cell. The space on a diskette which passes beneath the read/write head in a 4-µsecond interval. A bit cell contains a signal which represents the value of a single binary 0 or 1 (bit).

bit map. A table where each binary bit represents the allocation of a unit of storage. ProDOS uses bit maps to keep track of memory use (System Bit Map) and of disk use (Volume Bit Map).

bit slip marks. The epilogue of a disk field, used to double check that the disk head is still in read sync and the sector has not been damaged.

bit. A single binary digit (a 1 or a 0). A bit is the smallest unit of storage or information in a computer.

block. An arbitrary unit of disk space composed of two sectors or 512 bytes. ProDOS reads and writes a block at a time to improve performance and to allow support for larger devices.

boot/bootstrap. The process of loading a very large program into memory by loading successively larger pieces, each of which loads its successor. The program loads itself by "pulling itself up by its bootstraps".

BRK/Break. An assembly language instruction which can be used to force an interrupt and immediate suspension of execution of a program.

buffer. An area of memory used to temporarily hold data as it is being transferred to or from a peripheral, such as a disk drive.

bug. A programming error. Faulty operation of a program.

byte. The smallest unit of addressable memory in a computer. A byte usually consists of 8 bits and can contain a decimal number ranging from 0 to 255 or a single alphanumeric character.

carriage return. A control character which instructs the printer to end one line and begin another. When printing a carriage return is usually followed by a line feed.

carry flag. A 6502 processor flag which indicates that a previous addition resulted in a carry. Also, used as an error indicator by many system programs.

catalog. A directory of the files on a diskette. See directory.

chain. A linked list of data elements. Data is chained if its elements need not be contiguous in storage and each element can be found from its predecessor via an address or block pointer.

checksum/CRC. A method for verifying that data has not been damaged. When data is written, the sum of all its constituent bytes is stored with it. If, when the data is later read, its sum no longer matches the checksum, it has been damaged.

clobbered. Damaged or destroyed. A clobbered sector is one which has been overwritten such that it is unrecoverable.

code. Executable instructions to the computer, usually in machine language.

coldstart. A restart of a program which reinitializes all of its parameters, usually erasing any work which was in progress at the time of the restart. A DOS coldstart erases the BASIC program in memory.

contiguous. Physically next to. Two bytes are contiguous if they are adjoining each other in memory or on disk.

control block. A collection of data which is used by the operating system to manage resources. Examples of a control block used by DOS are the file buffers. Examples of control blocks used by ProDOS are the Volume Control Block (VCB) or a Volume Directory Header.

control character. A special ASCII code which is used to perform a unique function on a peripheral, but does not generate a printable character. Carriage return, line feed, form feed, and a bell are all control characters.

controller card. A hardware circuit board which is plugged into an Apple connector which allows communication with a peripheral device, such as a disk or printer. A controller card usually contains a small driver program in ROM.

CSWL. A vector in zero page, through which output data is passed for display on the CRT or for printing.

cycle. The smallest unit of time within the central processor of the computer. Each machine language instruction requires two or more cycles to complete. One cycle on the Apple is about one microsecond (one millionth of a second).

data sector buffer. On the Apple, a 256 byte buffer used by DOS to hold the image of any given sector on the diskette. As information is read from the file, data is extracted from the data sector buffer until it is exhausted, at which time it is refilled with the next sector image.

data type. The type of information stored in a byte. A byte might contain a printable ASCII character, binary numeric data, or a machine language instruction.

data. Units of information.

DCT. Device Characteristics Table. Used as an input parameter table to Read/Write Track/Sector (RWTS) to describe the hardware characteristics of the diskette drive.

decimal. A number system based upon powers of 10. Digits range from 0 to 9.

deferred commands. DOS commands which may (or must) be invoked from within an executing

BASIC program. OPEN, READ, WRITE, and CLOSE are all examples of deferred commands. ProDOS commands which may (or must) be invoked from within an executing BASIC program. OPEN, APPEND, READ, WRITE, and CLOSE are all examples of deferred commands.

digital. As opposed to analog. Discrete values as opposed to continuous ones. Only digital values may be stored in a computer. Analog measurements from the real world, such as a voltage or the level of light outside, must be converted into a numerical value which, of necessity, must be "rounded off" to a discrete value.

direct access. Peripheral storage allowing rapid access of any piece of data, regardless of its placement on the medium. Magnetic tape is generally not considered direct access, since the entire tape must be read to locate the last byte. A diskette is considered direct access, since the arm may be rapidly moved to any track and sector.

directory. A catalog of files stored on a diskette. The directory must contain each file's name and its location on the disk as well as other information regarding the type of data stored there. In ProDOS, a directory is a file in itself and one directory can describe other subdirectories.

disk initialization. The process which places track formatting information, including sectors and gaps, on a blank diskette. During disk initialization, DOS also places a VTOC and directory on the newly formatted disk, as well as saving the HELLO program, and the ProDOS FILER also places a copy of the boot loader in Block 0 and creates an empty Volume Directory in Blocks.

displacement. The distance from the beginning of a block of data to a particular byte or field. Displacements are usually given beginning with 0, for the first byte, 1 for the second, etc. Also known as an offset.

DOS. Also called DOS 3.2 and DOS 3.3. An earlier disk operating system for the Apple. DOS was designed to support BASIC programming using the Disk II drive only. When hard disks became available, Apple introduced ProDOS.

driver. A program which provides an input stream to another program or an output device. A printer driver accepts input from a user program in the form of lines to be printed, and sends them to the printer.

dump. An unformatted or partially formatted listing of the contents of memory or a diskette in hexadecimal. Used for diagnostic purposes.

encode. To translate data from one form to another for any of a number of reasons. In DOS 3.3 and ProDOS, data is encoded from 8 bit bytes to 6 bit bytes for storage on a Disk II.

entry point (EPA). The entry point address is the location within a program where execution is to start. This is not necessarily the same as the load point (or lowest memory address of the program).

EOF (End Of File). This mark signals the end of a data file. $00 for APPLE DOS text files. For ProDOS, a 3-byte number ranging from 0 to 16,777,216 (16 megabytes), which represents the offset to the end of the file. If the file is sequential (contains no "holes"), the EOF is also the length of the file in bytes.

epilogue. The last three bytes of a field on a track. These unique bytes are used to insure the integrity of the data which precedes them.

Exclusive OR. A logical operation which compares two bits to determine if they are different. 1 EOR 0 results in 1. 1 EOR 1 results in 0.

field. A group of contiguous bytes forming a single piece of data, such as a person's name, age, or social security number. In disk formatting, a group of bytes surrounded by gaps.

file buffers. In Apple DOS, a collection of buffers used to manage one open file. Included are a data sector buffer, a Track/Sector List sector buffer, a file manager workarea buffer, the name of the file, and pointers. The DOS command, MAXFILES 3, causes 3 of these file buffers to be allocated. In Apple ProDOS, a pair of 512-byte buffers used by the BASIC interpreter to manage one open file. Included are a buffer containing the block image of the current index block and one containing the image of the current data block. File buffers are allocated by the BI as needed by moving Applesoft's HIMEM pointer down in memory.

file descriptive entry. A single entry in a disk directory which describes one file. Included are the name of the file, its data type, its length, its access restrictions, its creation date, its location on the diskette, etc.

file manager. That portion of DOS which manages files. The file manager handles such general operations as OPEN, CLOSE, READ, WRITE, POSITION, RENAME, DELETE, etc.

file type. The type of data held by a file. Valid DOS file types are Binary, Applesoft, Integer-BASIC, Text, Relocatable, S, A, and B. Valid ProDOS file types include Binary (BIN), Applesoft (BAS), Text (TXT), and System (SYS) files. ProDOS supports up to 256 different file types.

file. A named collection of data on a diskette or other mass storage medium. Files can contain data or programs.

firmware. A middle ground between hardware and software. Usually used to describe micro-code or programs which have been stored in readonly memory (ROM).

gaps. The spaces between fields of data on a diskette. Gaps on an Apple diskette contain self-sync bytes.

garbage collection. The process of combining many small embedded free spaces into one large area. For example, Applesoft performs garbage collection on its string storage to recover memory allocated to strings which have been deleted.

Global Page. A 256-byte area of memory set aside by ProDOS to contain system variables of general interest. Two Global Pages are currently defined: the System Global Page at $BF00; and the BI Global Page at $BE00. The structure of the Global Pages is rigidly defined, allowing external programs to communicate with ProDOS without depending upon release dependent locations. See also vectors.

hard error. An unrecoverable Input/Output error. The data stored in the disk sector can never be successfully read again.

hardware. Physical computer equipment, as opposed to programs which run on the equipment. A disk drive is an example of a hardware component.

head. The read/write head on a diskette drive. A magnetic pickup, similar in nature to the head on a stereo tapedeck, which rests on the spinning surface of the diskette.

hexadecimal/HEX. A numeric system based on powers of 16. Valid hex digits range from 0 to 9 and A to F, where A is 10, B is 11, . . . F is 15. Standard Apple practice is to indicate a number as hexadecimal by preceding it with a dollar sign. $B30 is 11-256s plus 3-16s plus 0-1s, or 2864 in decimal. Two hexadecimal digits can be used to represent the contents of one byte. Hexadecimal is used with computers because it easily converts to binary.

high memory. The memory locations which have high address values. $FFFF is the highest memory location. Also called the "top" of memory.

HIMEM. Applesoft's zero-page address which identifies the first byte past the available memory which can be used to store BASIC programs and their variables.

I/O (Input/Output) error. An error which occurs during transmission of data to or from a peripheral device, such as a disk or cassette tape.

immediate command. A DOS or ProDOS command which may be entered at any time, especially when the operating system is waiting for a command from the keyboard. Deferred commands are the opposite of immediate commands.

index block. A block containing a table of block numbers describing the order and location of the blocks of data within a file. A sapling file has one index block describing up to 256 data blocks. A tree file has a master index block which points to other index blocks, which in turn point to the data blocks in the file.

index. A displacement into a table or block of storage.

instruction. A single step to be performed in an assembly language or machine language program. Instructions perform such operations as addition, subtraction, store, or load.

integer. A "whole" number with no fraction associated with it, as opposed to floating point.

intercept. A program which logically places itself in the execution path of another program, or pair of programs. A video intercept is used to redirect program output from the screen to a printer, for example.

interleave. The practice of selecting the order of sectors on a diskette track to minimize access time due to rotational delay. Also called "skewing" or interlacing.

interpreter. A program which translates user written commands or program statements directly into their intended function. Applesoft is an interpreter. The ProDOS BASIC Interpreter translates ProDOS commands into functions such as loading, saving, reading or writing files. Another name for ProDOS interpreters is System Programs.

interrupt. A hardware signal which causes the computer to halt execution of a program and enter a special handler routine. Interrupts are used to service real-time clock time-outs, BRK instructions, and RESET.

IOB. DOS Input/Output Block. A collection of parameter data, passed to Read/Write Track/Sector, describing the operation to be performed.

JMP. A 6502 assembly language instruction which causes the computer to begin executing instructions at a different location in memory. Similar to a GOTO statement in BASIC.

JSR. A 6502 assembly language instruction which causes the computer to "call" a subroutine. Similar to a GOSUB statement in BASIC

K. A unit of measurement, usually applied to bytes. 1K bytes is equivalent to 1024 bytes.

Kernel. That part of ProDOS which provides the basic operating system support functions. The Kernel resides in the Language Card or bank switched memory and consists of the MLI, interrupt handler, and diskette and calender/clock device drivers.

key block. The first block of a ProDOS file.

KSWL. A vector in zero page through which input data is passed from the keyboard or a remote terminal.

label. A name associated with a location in a program or in memory. Labels are used in assembly language much like statement numbers are used in BASIC.

language card. An additional 16K of RAM added to an Apple II or Apple II Plus using a card in slot 0. The card gets its name from its original use with the Apple UCSD PASCAL system and for loading other versions of BASIC. Apple IIe's have this additional memory built in. See also bank switched memory.

latch. A component into which the Input/Output hardware can store a byte value, which will hold that value until the central processor has time to read it (or vice versa).

link. An address or block pointer in an element of a linked chain of data or buffers.

list. A one dimensional sequential array of data items.

load point (LP). The lowest address of a loaded assembly language program—the first byte loaded. Not necessarily the same as the entry point address (EPA).

locked. A file is locked if it is restricted from certain types of access—usually one which is read only. ProDOS provides control over file access through the use of directory entry bits.

logical. A form of arithmetic which operates with binary "truth" or "false", 1 or 0. AND, OR, NAND, NOR, and EXCLUSIVE OR are all logical operations.

LOMEM. Applesoft's zero-page address which identifies the first byte of available memory which can be used to store BASIC programs and their variables.

loop. A programming construction in which a group of instructions or statements are repeatedly executed.

low memory. The memory locations with the lowest addresses. $0000 is the lowest memory location. Also called the "bottom" of memory.

LSB/lo order. Least Significant Bit or Least Significant Byte. The 1's bit in a byte or the second pair of hexadecimal digits forming an address. In the address $8030, $30 is the LO order part of the address.

mark. A 3-byte "byte number" or position within a ProDOS file. When a file is being read by the MLI, a current mark is maintained as well as the EOF mark. See also EOF.

master disk. A DOS diskette which will boot in an Apple II of any size memory and take full advantage of it.

microsecond. A millionth of a second. Equivalent to one cycle of the Apple II central processor. Also written as "µsec"

MLI (Machine Language Interface). The MLI is part of the ProDOS Kernel which resides in the language card or bank switched memory. The MLI performs such functions as OPENing a file, WRITING to a file, or DESTROYing a file.

monitor. A machine language program which always resides in the computer and which is the first to receive control when the machine is powered up. The Apple monitor resides in ROM and allows examination and modification of memory at a byte level.

MSB/hi order. Most Significant Bit or Most Significant Byte. The 128's bit of a byte (the left-most) or the first pair of hexadecimal digits in an address. In the byte value $83, the MSB is on (is a 1).

nibble/nybble. A portion of a byte, usually 4 bits and represented by a single hexadecimal digit. $FE contains two nibbles, $F and $E.

null. Empty, having no length or value. A null string is one which contains no characters. The null control character ($00) produces no effect on a printer (also called an idle).

object code. A machine language program in binary form, ready to execute. Object code is the output of an assembler.

object module. A complete machine language program in object code form, stored as a file on a diskette.

offset. The distance from the beginning of a block of data to a particular byte or field. Offsets are usually given beginning with 0, for the first byte, 1 for the second, etc. Also known as a displacement.

opcode, operation code. The three letter mnemonic representing a single assembly language instruction. JMP is the opcode for the jump instruction.

operating system. A machine language program which manages the memory and peripherals automatically, simplifying the job of the applications programmer.

OR. The logical operation comparing two bits to determine if either of them are 1. 1 OR 1 results in 1 (true). 1 OR 0 results in 1, 0 OR 0 results in 0 (false).

overhead. The space required by the system, either in memory or on the disk, to manage either. In DOS, the disk directory and VTOC are part of a diskette's overhead. In ProDOS, the boot blocks, Volume Directory, and Volume Bit Map are part of a diskette's overhead.

page. 256 bytes of memory which share a common high order address byte. Zero page is the first 256 bytes of memory ($0000 through $00FF).

parallel. A communication mode which sends all bits in a byte at once, each over a separate line or wire. Opposite of serial.

parameter list. An area of storage set aside for communication between a calling program and a subroutine. The parameter list contains input and output variables which will be used by the subroutine.

parity. A scheme which allows detection of errors in a single data byte, similar to checksums but on a bit level rather than a byte level. An extra parity bit is attached to each byte which is a sum of the bits in the byte. Parity is used to detect or correct single bit failures, and when sending data over communication lines to detect noise errors.

parse. The process of interpreting character string data, such as a command with keywords.

patch. A small change to the object code of an assembly language program. Also called a "zap".

pathname. A string describing the path ProDOS must follow to find a file. A fully qualified pathname consists of the volume name followed by one or more directory names followed by the name of the file itself. If a partial pathname is given, a default prefix is attached to it to form a complete pathname. See also prefix.

peripheral. A device which is external to the computer itself, such as a disk drive or a printer. Also called an Input/Output device.

physical record. A collection of data corresponding to the smallest unit of storage on a peripheral device. For disks, a physical record is a sector.

pointer. The address or memory location of a block of data or a single data item. The address "points" to the data. A pointer may also be a block number, such as the pointer to the Volume Bit Map in the Volume Directory Header.

prefix. A system maintained default character string which is automatically attached to file names entered by the user to form a complete pathname. See also pathname.

prologue. The three bytes at the beginning of a disk field which uniquely identify it from any other data on the track.

PROM (Programmable Read Only Memory). PROMs are usually used on controller cards associated with peripherals to hold the driver program which interfaces the device to applications programs.

prompt. An output string which lets the user know that input is expected. An "*" is the prompt character for the Apple monitor.

protected disk. A diskette whose format or content has been modified to prevent its being copied. Most retail software today is distributed on protected disks to prevent theft.

pseudo-opcode. A special assembly language opcode which does not translate into a machine instruction. A pseudo-opcode instructs the assembler to perform some function, such as skipping a page in an assembly listing or reserving data space in the output object code.

RAM (Random Access Memory). Computer memory which will allow storage and retrieval of values by address.

random access. Direct access. The capability to rapidly access any single piece of data on a storage medium without having to sequentially read all of its predecessors.

recal. Recalibrate the disk arm so that the read/write head is positioned over track zero. This is done by pulling the arm as far as it will go to the outside of the diskette until it hits a stop, producing a "clacking" sound.

record. A collection of associated data items or fields. One or more records are usually associated with a file. Each record might correspond to an employee, for example.

reference number (REF_NUM). An arbitrary number assigned to an open file by the MLI to simplify identification in later calls.

register. A named temporary storage location in the central processor itself. The 6502 has 5 registers: the A, X, Y, S, and P registers. Registers are used by an assembly language program to access memory and perform arithmetic.

release. A version of a distributed piece of software. There have been several releases of DOS.

relocatable. The attribute of an object module file which contains a machine language program and the information necessary to make it run at any memory locations.

return code. A numeric value returned from a subroutine, indicating the success or failure of the operation attempted. A return code of zero usually means there were no errors. Any other value indicates the nature of the error, as defined by the design of the subroutine.

ROM (Read Only Memory). Memory which has a permanent value. The Apple monitor and Applesoft BASIC are stored in ROM.

RWTS (Read/Write Track/Sector). A collection of subroutines which allow access to the diskette at a track and sector level. RWTS is part of DOS and may be called by external assembly language programs.

sapling. A ProDOS file which requires only one index block (2 to 256 data blocks). A sapling ranges from 513 bytes to 131,072 bytes in length. See also seedling and tree.

search. The process of scanning a track for a given sector.

sector address. A disk field which identifies the sector data field which follows in terms of its volume, track, and sector number.

sector data. A disk field which contains the actual sector data in nibbilized form.

sector. The smallest updatable unit of data on a disk track. One sector on an Apple Disk II contains 256 data bytes.

seedling. A ProDOS file which has only a single data block (512 bytes). A seedling file does not require index blocks. See also sapling and tree.

seek. The process of moving the disk arm to a given track.

self-sync. Also called "auto-sync" bytes. Special disk bytes which contain more than 8 bits, allowing synchronization of the hardware to byte boundaries when reading.

sequential access. A mode of data retrieval where each byte of data is read in the order in which it was written to the disk.

serial. A communication mode which sends data bits one at a time over a single line or wire. As opposed to parallel.

shift. A logical operation which moves the bits of a byte either left or right one position, moving a 0 into the bit at the other end.

skewing. The process of interleaving sectors. See interleave.

slave disk. A diskette with a copy of DOS which is not relocatable. The DOS image will always be loaded into the same memory location, regadless of the size of the machine.

soft error. A recoverable I/O error. A worn diskette might produce soft errors occasionally.

software. Computer programs and data which can be loaded into RAM memory and executed.

SOS (Sophisticated Operating System). The standard operating system for the Apple III computer.

source code. A program in a form which is understandable to humans; in a character form as opposed to internal binary machine format. Source assembly code must be processed by an assembler to translate it into machine or "object" code.

sparse file. A files with random organization (see random access) which contains areas which were never initialized. A sparse file might have an END_OF_FILE mark of 26 megabytes but only contain several hundred bytes.

state machine. A process (in software or hardware) which defines a unique target state, given an input state and certain conditions. A state machine approach is used in the ProDOS BASIC Interpreter to keep track of its video intercepts and by the hardware on the disk controller card to process disk data.

strobe. The act of triggering an I/O function by momentarily referencing a special I/O address. Strobing $C030 produces a click on the speaker. Also called "toggling".

subroutine. A program whose function is required repeatedly during execution, and therefore is called by a main program in several places.

system disk. A ProDOS volume which contains the system files necessary to allow ProDOS to be booted into memory. Normally, the PRODOS and BASIC.SYSTEM files are necessary. A STARTUP program may also be present.

system program. A ProDOS program, written in machine language, which acts as an intermediary between the user and the ProDOS Kernel. BASIC.SYSTEM, FILER, and CONVERT are all examples of System Programs. See also interpreter and BI.

T/S list. Track/Sector List. A sector which describes the location of a file by listing the track and sector number for each of its data sectors in the order that they are to be read or written.

table. A collection of data entries, having a similar format, residing in memory. Each entry might contain the name of a program and its address. For examples. A "lookup" can be performed on such a table to locate any given program by name.

toggle. The act of triggering an I/O function by momentarily referencing a special I/O address. Toggling $C030 produces a click on the speaker. Also called "strobe".

tokens. A method where human recognizable words may be coded to single binary byte values for memory compression and faster processing. BASIC statements are tokenized, where hex codes are assigned to words like IF, PRINT, and END.

track. One complete circular path of magnetic storage on a diskette. There are 35 concentric tracks on an Apple diskette.

translate table. A table of single byte codes which are to replace input codes on a one-for-one basis. A translate table is used to convert from 6-bit codes to disk codes.

tree. A ProDOS file which requires several index blocks (131,073 to 16,777,216 bytes of data). See also index block, seedling, and sapling.

TTL (Transistor to Transistor Logic). A standard for the interconnection of integrated circuits which also defines the voltages which represent 0s and 1s.

unlocked. A file which allows all types of access (READ, WRITE, DELETE, RENAME, etc.). See also locked.

utility. A program which is used to maintain, or assist in the development of, other programs or disk files.

vector. A collection of pointers or JMP instructions at a fixed location in memory which allow access to a relocatable program or data.

Volume Directory. The first directory on a disk volume. Also called the "root" directory. All

other directories must be reached by first reading the Volume Directory.

volume. An identification for a diskette, disk platter, or cassette, containing one or more files.

VTOC. Volume Table Of Contents. Based upon the IBM OS/VS VTOC. On the APPLE, a sector mapping the free sectors on the diskette and giving the location of the directory.

warmstart. A restart of a program which retains, as much as is possible, the work which was in progress at the time. A DOS warmstart retains the BASIC program in memory.

write protected. A diskette whose write protect notch is covered, preventing the disk drive from writing on it.

ZAP. From the IBM mainframe utility SUPERZAP. A program which allows updates to a disk at a byte level, using hexadecimal.

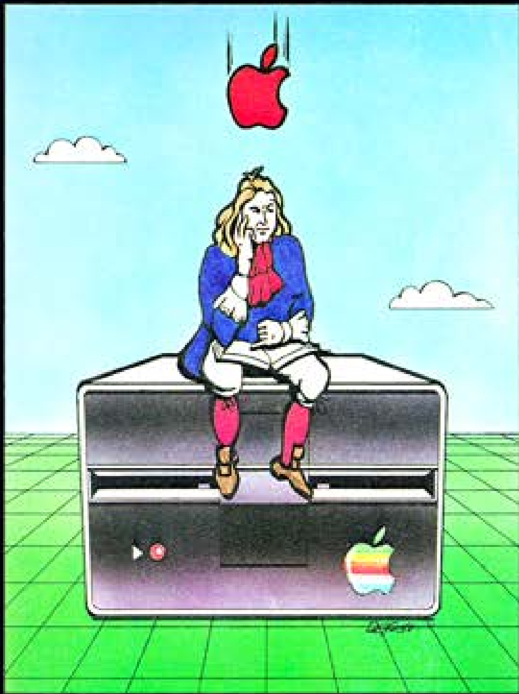
zero page. The first 256 bytes of memory in a 6502 based machine. Zero page locations have special significance to the central processor, making their management and assignment critical.

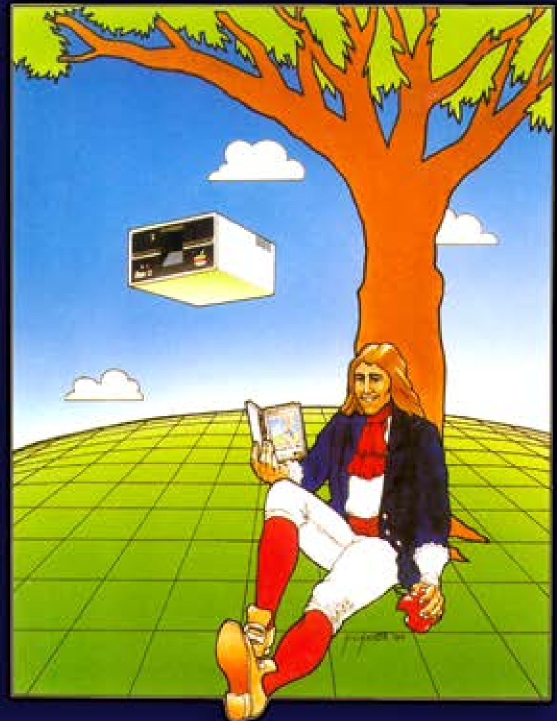
ProDOS
DOS 3.3

Pascal
CP/M
FARCE

ONLINE LINKS FOR THE APPLE II COMMUNITY

Call A.P.P.L.E., an Apple user group
https://www.callapple.org/
applefritter, an online forum
https://www.applefritter.com/
A2Central
https://a2central.com/
Apple 2 Online, a resource for Apple II documentation
https://apple2online.com/
Apple II Enthusiasts Group on Facebook
https://www.facebook.com/groups/5251478676
Apple II History, the definitive history of the Apple II computer
https://apple2history.org/
WOzFest
https://the.europlus.zone/WOzFest
Oz KFest
http://ozkfest.net
Apple IIoz Mailing List
http://www.appleiioz.com/appleiioz/Apple_IIoz/Mailing_List.html





From the Editor:

This book combines two works from the 1980s: *Beneath Apple DOS* and *Beneath Apple ProDOS*. I have tried to keep as much of the original text as possible, but some of the figures and tables have been re-drawn to either make them easier to read or easier to understand (or both).

Also, some of the information is duplicated in both books, but I have kept the duplicates so if you're studying one operating system, you don't need to reference the other. The one main item I combined were the glossaries, since many of the same terms are in both. It is the last chapter in the book.

If this is your first time seeing this book, I hope you experience the joy so many have over the years. If you had this book previously, I hope I've kept the feel of the original book, while updating it for a new generation.

— John Snape