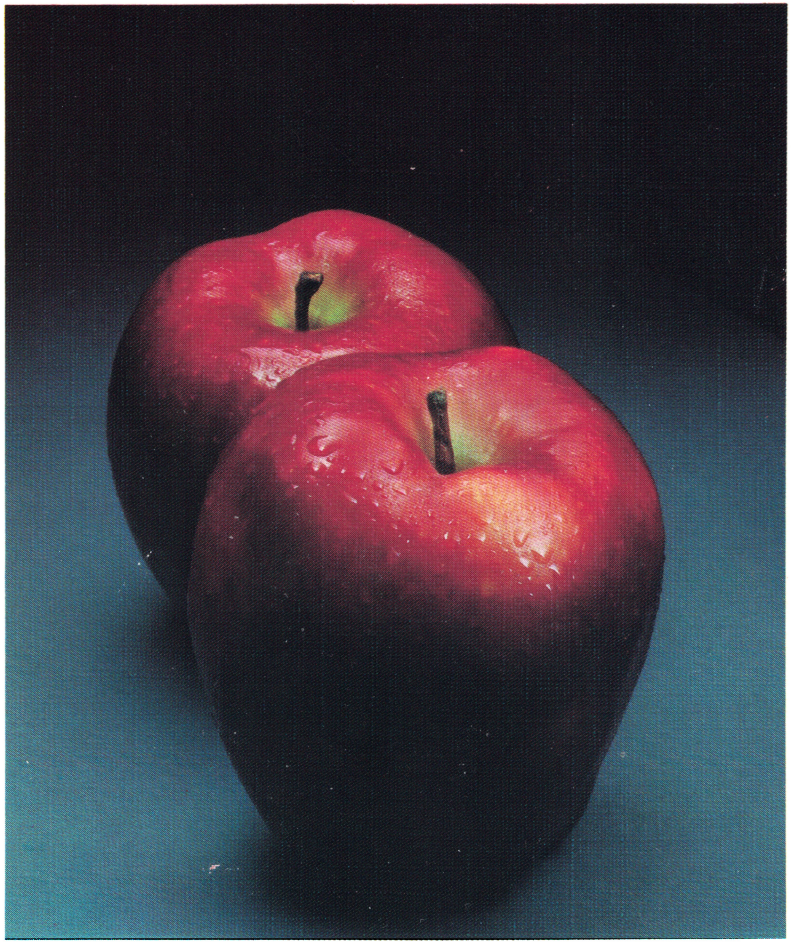
Apple ProDOS:

Advanced Features for Programmers



Brady

Gary B. Little

Apple ProDOS

APPLE PRODOS

Advanced Features for Programmers

Gary B. Little

Brady Communications Company, Inc.
A Prentice-Hall Publishing Company
Bowie, Maryland

Apple ProDOS

Library of Congress Cataloging in Publication Data

Little, Gary B., 1954–
Apple ProDOS.

Bibliography: p.
Includes index.
1. ProDOS (Computer operating system) 2. Apple II (Computer)—Programming. I. Title.
QA76.76.63L56 1985 001.64'2 85-5732

ISBN 0-89303-441-X

Prentice-Hall of Australia, Pty., Ltd., *Sydney*
Prentice-Hall Canada, Inc., Scarborough, *Ontario*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte., Ltd., *Singapore*
Editora Prentice-Hall Do Brasil LTDA., *Rio de Janeiro*
Whitehall Books, Limited, Petone, *New Zealand*

Printed in the United States of America

85 86 87 88 89 90 91 92 93 94 95 1 2 3 4 5 6 7 8 9 10

This book is dedicated to my fellow SAGE members

Walter Cooke
Chris Howerton
Henry Matthison
Archie Reid
John Spraggs
Steve Wozniak

Thanks for the inspiration

Note to Authors

Have you written a book related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. Brady produces a complete range of books for the personal computer market. We invite you to write to David Culverwell, Publishing Director, Brady Communications Company, Inc., Bowie, MD 20715.

Publishing Director: David Culverwell
Acquisition Editor: Chris Williams
Production Editor: Barbara Werner
Text Design: Nancy Epting
Art Director: Don Sellers
Assistant Art Director: Bernard Vervin
Cover Photography: George Dodson
Manufacturing Director: John A. Komsa

Typesetting: Automated Graphic Systems Inc., White Plains, Maryland
Printing: R. R. Donnelley & Sons Co., Harrisonburg, Virginia
Typefaces: Melior (text), ITC Korinna (display) Universal Monospace (program)

Contents

Preface

New software, especially systems software like ProDOS, fascinates and captivates me. To analyze it, I keep repeating a ritualistic four-step procedure until I finally understand how the software works from the inside out:

- I boot it.
- I study its code.
- I make notes.
- I turn off the Apple (usually around 2 a.m.!).

My reaction to ProDOS was no different. As soon as I received my prerelease copy from Apple in early 1984 I rushed straight to my computer room and booted it on the trusty //e. Much PEEKing, POKEing, and CALLing took place that night and continued for several hacker-years thereafter until I finally felt confident enough to convert my cryptic notes into intelligible form.

This book, of course, is the result of those many nights of studying the internals of ProDOS. It is primarily a guide for intermediate to advanced programmers since it will be presumed that you are reasonably familiar with 6502 assembly language and Applesoft programming techniques. Even if you're not, however, you should find the general descriptions of how ProDOS handles files and manages peripheral devices useful and revealing.

Some of the more interesting topics that will be covered are:

- how ProDOS organizes files on disks
- how to use the ProDOS MLI (machine language interface) commands to access disk files
- how the BASIC.SYSTEM (Applesoft) interpreter works
- how to write and install you own BASIC.SYSTEM disk commands
- how to write and install I/O (input/output) drivers for disk devices
- how to write and install clock drivers
- how ProDOS manages interrupts from I/O devices

Several programming examples are presented throughout the book to highlight important discussions and to help clarify difficult concepts. This includes a program to read or write any data block on a disk so that you can easily explore the internal structures of directories and files and also a program that creates a high-speed RAMdisk using an area of the Apple II's main memory for disk-like storage. Most of the programs are written in 6502 assembly language and were developed using Glen Bredon's Merlin Pro assembler (Roger Wagner Publishing, Inc., 1984). Some of the unique instructions of this assembler are reviewed in Appendix I so that you will be able to understand them even if you are using a different ProDOS-compatible assembler such as the one included with Apple's "ProDOS Assembler Tools."

There are several specialized topics that are referred to in this book that are not dealt with in great detail because they really have little to do with ProDOS itself. Instead, you will usually be referred to my earlier book, *Inside the Apple*

//e. If you are using an Apple //c you will want to refer to another of my books, *Inside the Apple //c*, instead. Both books were published in 1985 by Brady Communications.

Please note that this book is not a tutorial on how to use the standard Applesoft disk commands that are provided by the ProDOS BASIC.SYSTEM interpreter; if you require such a book, I suggest that you read Apple's own *BASIC Programming With ProDOS*. Instead, I will be concentrating on the lower-level internal ProDOS commands that are defined by its machine language interface (MLI) and which are accessible from 6502 assembly language programs only.

Finally, I would point out that there is no need to suffer the agony of manually entering the programs that will be described. Instead, you can order a diskette directly from me that contains these programs (in both source and object code formats) as well as some additional "bonus" programs that are useful ProDOS utilities (they are described in Appendix V). For ordering information, see the last page of the book.

Following its usual practice, Brady Communications commissioned several independent writers and programmers to review this manuscript for style and technical accuracy prior to publication. Although I was not told who these reviewers were (this keeps the reviewers honest) they now certainly know me and I would like to thank them for their useful comments.

It was inevitable that I did learn the identity of one of the reviewers who went above and beyond the call of duty to assist me in polishing the manuscript. Thank you, Cecil Fretwell.

Gary B. Little

Vancouver, British Columbia, Canada
May 1985

Chapter 1

AN INTRODUCTION TO ProDOS

In this book we are going to take a close look at the inner workings of ProDOS the ("Professional Disk Operating System"), the newest disk operating system for the Apple II series of microcomputers. ProDOS is a complex assembly language program that, like all operating systems, manages the flow of data to and from a permanent storage medium (such as a 5.25-inch floppy diskette). It does so by translating the high-level disk commands used by an applications program into the low-level instructions necessary to operate a disk drive controller that is able to read data from and store data to any position on the medium.

ProDOS is also responsible for defining: the data structures to be used for storing related groups of information, called *files*, on the disk; the directories where the names of files (and other relevant information) are stored; the method used to keep track of what parts of the medium are in use; the method used to load the operating system; and so on. We will be analyzing these definitions in later chapters.

ProDOS works well with both disk devices that Apple has released for the Apple II family: the Disk][and the ProFile. And, as we will see in Chapter 7, it can also be modified easily to work with non-Apple disk devices as well.

The standard disk drive unit for the Apple II computers (except the Apple //c) is called the Disk][. It is interfaced to the system through a cable connected to a disk controller card plugged into one of the slots at the back of the Apple II (slot 6 is recommended). On the slotless Apple //c, the disk drive and controller are of a different design and are built in to the computer. Fortunately (but certainly not accidentally), the drive and controller are totally compatible with

the Disk][and its controller. This means that each drive can read and write diskettes formatted by the other.

Apple's 5-megabyte fixed disk (or *hard disk*), the ProFile, can also be used by all members of the Apple II family except the Apple //c. It is said to be "fixed" because the magnetic medium, unlike a floppy diskette, is not removable from the drive unit. This device can access information much more quickly and hold much more of it than the Disk][. The ProFile is interfaced to the Apple II through its own controller card, one quite different from that used with the Disk][.

A History of DOS for the Apple II

When the Apple II was first released in 1977, the cassette recorder was the only mass storage device available to most users. The reason was simple: the original Apple II had a built-in cassette port that made it convenient and simple to hook up a recorder, but an Apple-compatible disk drive and controller had yet to be invented.

Working with normal cassette tape as a storage medium is no treat. The program storage and loading rate is very slow and you're never sure if glitches on the tape have rendered the program unreadable until it's too late to recover. Furthermore, files on cassette tape cannot be named or automatically located by the Apple II, so it is necessary to keep meticulous written records of which programs are stored where so that the tape can be properly positioned by hand.

Steve Wozniak, the inventor of the Apple II, was apparently as frustrated with cassette tape as anyone else. In the winter of 1977-78 he designed a disk controller peripheral card for the Apple II that could be used with a standard drive unit that was later to be called the Disk][. At the same time, Bob Shepardson, and later Randy Wigginton, Dick Huston, and Rick Auricchio, were busy writing a disk operating system that would allow programmers to create and access files easily on the 5.25-inch diskette media that were used with the drives.

The Disk][, its controller card, and the first released version of the disk operating system (called DOS 3.1) were finally shipped in the early summer of 1978. This was probably the most important event in the early history of Apple because it meant that, for the first time, complex business software could be written for the Apple II. Such software was needed to create and manipulate large database files quickly and easily, a feat that would be next to impossible if cassette tapes were used instead of diskettes.

Several changes were made to DOS 3.1 in the months following its initial release in order to fix the inevitable bugs that wriggled to the surface. DOS finally stabilized at version 3.2.1 by mid-1979. This early version of DOS formatted diskettes with 35 data tracks and with thirteen 256-byte sectors per track (for a total of 113.75K of storage, where 1K = 1,024 bytes). In fact, the program in ROM on the disk controller card could only start up (boot) diskettes that used this specific 13-sector format.

Apple also released its Pascal operating system in 1979. This system manages files quite differently than either DOS 3.2.1, DOS 3.3, or ProDOS, but this does not cause much concern because it is not intended to be used in the Applesoft programming environment supported by the other DOS's. It is sometimes convenient to be able to transfer a Pascal textfile to a DOS diskette (and vice versa) and this can be done with special programs that are available from independent commercial sources and users groups.

Apple upgraded DOS 3.2.1 in a substantial way in 1980 to support the new 16-sector-per-track formatting scheme used by Apple Pascal. The result was DOS 3.3., a version that was still current when ProDOS was ultimately released in early 1984. The formatting change also necessitated a change in the ROM boot program on the disk controller card. The main advantage of switching to the new formatting scheme was that it enabled diskettes to hold an additional 16.25K of information (for a total of 140K). The main disadvantage was that it was not possible for DOS 3.3 to read files directly from a DOS 3.2.1-formatted diskette (and DOS 3.2.1 diskettes could not be directly booted). Fortunately, Apple supplied a program called MUFFIN that could be used to transfer files from the old diskette format to the new one and another program called BOOT13 that allowed DOS 3.2.1 diskettes to be booted.

ProDOS was first released by Apple in January 1984. It will run on any standard Apple //e and Apple //c, or on an Apple II Plus that has a 16K extended memory card (often called a *language card*) installed in peripheral slot #0. It will also run on the original Apple II with a 16K card if the Applesoft language, not the Integer BASIC language, is installed in ROM. With the release of ProDOS, Apple served notice that it would no longer release new software products that use DOS 3.3 and has urged independent software developers to do the same. Nevertheless, DOS 3.3 remains a popular operating system and new programs that use it are still being released (usually simultaneously with ProDOS versions).

Early versions of ProDOS suffered from several annoying bugs that were subsequently removed in later versions. As of this writing the current version was 1.1.1. It is still not entirely bug-free, so we can undoubtedly look forward to at least one more version in the near future. The release dates for the various versions of ProDOS are shown in Appendix II.

The scheme used to store files on a ProDOS diskette is quite unlike that used with DOS 3.3 although the same general 16-sector-per-track format is used. Apple supplies a program called CONVERT on the ProDOS master diskette that permits interchange of files between the two kinds of formats. Unfortunately, not all programs that are converted this way will execute properly with the other operating system because of important differences in the command set and memory usage.

Also announced in January 1984 was a ProDOS-compatible controller card for the 5-megabyte ProFile fixed disk that Apple had released a couple of years earlier for use with its Apple /// system. On bootup, ProDOS automatically recognizes the presence of the ProFile and will interact with it just as if it were another Disk][device (except that ProDOS knows the ProFile has a much greater storage capacity). The internal structure of ProDOS is such that it can easily

deal with even higher capacity devices; a volume size of up to 32 megabytes is supported.

Apple never did announce DOS 3.3 support for the ProFile and the reason is understandable: there is no simple mechanism in that operating system for integrating disk devices other than the Disk][, or for properly managing files stored on them. Furthermore, the single directory structure used by DOS 3.3 is inappropriate in an environment that can manipulate hundreds of files. Several independent fixed disk vendors do support DOS 3.3 by providing software that tricks DOS 3.3 into thinking that the fixed disk is simply a group of individually numbered 140K (diskette size) volumes. The problem with this approach is that very few commercial software systems support it and file sizes cannot exceed the capacity of one volume.

Comparing ProDOS with DOS 3.3

DOS 3.3 is made up of two main modules: the I/O (input/output) driver, which communicates directly with the disk device, and the Applesoft command interpreter, which parses and executes the Applesoft disk commands that DOS 3.3 defines (OPEN, READ, CATALOG, and so on). The equivalent modules in ProDOS are split into two program files that are called PRODOS (the I/O driver) and BASIC.SYSTEM (the Applesoft command interpreter). In most applications, PRODOS will automatically load BASIC.SYSTEM when a disk is started up. Thus, it is necessary to compare DOS 3.3 to the PRODOS-BASIC.SYSTEM combination and not simply to PRODOS proper.

Short descriptions of the Applesoft disk commands used by BASIC.SYSTEM and DOS 3.3 are given in Table 1-1. Most of these commands are available in both environments, but some are unique to one or the other. In general, the BASIC.SYSTEM versions of the duplicated commands are more powerful than their DOS 3.3 counterparts because they support more command parameters. (For example, consider the RUN command. The syntax for the BASIC.SYSTEM (ProDOS) version is

```
RUN filename,@#,S#,D#
```

where the "#" in ",@#" represents the line number at which execution of the Applesoft program is to begin; the DOS 3.3 version of RUN does not support the line number parameter.) We'll be reviewing these parameters in Chapter 5. Furthermore, some commands behave slightly differently in one system than they do in the other.

Not surprisingly, the more powerful PRODOS-BASIC.SYSTEM environment occupies a lot more memory space than does DOS 3.3. In fact, it uses almost twice as much space. Fortunately, most of ProDOS resides in a 16K bank-switched RAM space that does not conflict with the space used by the Applesoft interpreter. This space is built in to an Apple //e or //c and can be added to an Apple II or Apple II Plus by installing a 16K memory card in slot #0. Two side

Table 1-1. Comparing the BASIC.SYSTEM and DOS 3.3 Applesoft disk commands.

Command	*Description*	*Availability* ProDOS	DOS 3.3
APPEND	Open a file and prepare to add data to it	YES	YES
BLOAD	Load a file (usually binary)	YES	YES
BRUN	Load and execute an assembly language program that is in a binary file	YES	YES
BSAVE	Save a file (usually binary)	YES	YES
CATALOG	List all the files on the medium (long form)	YES	YES
CLOSE	Close a file	YES	YES
DELETE	Delete a file	YES	YES
EXEC	Execute commands from a textfile	YES	YES
IN#	Redirect character input	YES	YES
LOAD	Load an Applesoft program	YES	YES
LOCK	Lock a file	YES	YES
NOMON	[Permitted but ignored under ProDOS]	YES	YES
OPEN	Open a file	YES	YES
POSITION	Prepare to read from or write to a specific position in the file	YES	YES
PR#	Redirect character output	YES	YES
READ	Read from a file	YES	YES
RENAME	Rename a file	YES	YES
RUN	Load and execute an Applesoft program (or, if no filename is specified, execute the program in memory)	YES	YES
SAVE	Save an Applesoft program	YES	YES
UNLOCK	Unlock a file	YES	YES
VERIFY	Check for the existence of a file; if no filename is specified, display a copyright notice	YES	YES
WRITE	Write to a file	YES	YES
- ("dash")	Execute an Applesoft, binary, text, or system file	YES	NO
BYE	Transfer control to another system program	YES	NO
CAT	List the files on the medium (short form)	YES	NO
CHAIN	Transfer control to another Applesoft program while maintaining the current variables	YES	NO (1)
CREATE	Create a file (usually a directory file)	YES	NO
FLUSH	Write the contents of a file buffer to the medium	YES	NO

(continued)

Table 1-1. Comparing the BASIC.SYSTEM and DOS 3.3 Applesoft disk commands (continued).

Command	*Description*	*Availability*	
		ProDOS	*DOS 3.3*
FRE	Perform Applesoft garbage collection	YES	NO (2)
PREFIX	Set up the name of the active directory	YES	NO
RESTORE	Restore Applesoft variables for a file	YES	NO
STORE	Save Applesoft variables to a file	YES	NO
FP	Initialize Applesoft mode	NO	YES
INIT	Format a diskette	NO	YES (3)
INT	Initialize Integer BASIC mode	NO	YES
MAXFILES	Create space for file buffers	NO	YES
MON	Enable the display of DOS operations	NO	YES

Notes:

1. Applesoft programs can be chained under DOS 3.3 by loading and calling a subroutine called CHAIN that is found on the DOS 3.3 master diskette.
2. The Applesoft FRE command can be used to garbage-collect under DOS 3.3 (and ProDOS also). It executes much more slowly than the corresponding ProDOS command.
3. Under ProDOS, a diskette is formatted by using a separate program called FILER that is found on the ProDOS master diskette.

effects of the use of this space by ProDOS are that ProDOS cannot function with a program that uses the memory card for data storage or with the original version of Apple BASIC called Integer BASIC. In a DOS 3.3 environment, Integer BASIC is loaded into the same bank-switched RAM area that ProDOS uses and then is selected simply by throwing a special software-controlled switch that selects the Integer BASIC RAM area instead of the Applesoft ROM area that occupies the same address space. The loss of Integer BASIC is not a serious one, however, since virtually all useful BASIC software for the Apple has been written in Applesoft.

The other major difference between DOS 3.3 and BASIC.SYSTEM is in the handling of file buffers. A file buffer is a memory area reserved for use by an open file; it holds the data contained in the active part of the file, as well as information defining the location of the file on the disk. When DOS 3.3 is first started it automatically sets up three such buffers; a different number (from 1 to 16) can be reserved by using a command called MAXFILES. The DOS 3.3 file buffers are each 595 bytes long and are stored between the top of the Applesoft program space (this address is stored at $73/$74 and is called HIMEM) and the start of the DOS 3.3 code (at $9D00).

ProDOS, on the other hand, initially sets up no file buffers; it dynamically allocates and de-allocates file buffers as files are opened and closed. When a file is opened, HIMEM is lowered by 1024 bytes and the buffer is assigned to the 1024-byte space beginning at HIMEM+1024. When a file is closed, the file

buffers below its own are repositioned and then HIMEM is raised by 1024 bytes. (A total of 8 files can be open simultaneously.) Because this dynamic space allocation method is used, it is not possible to use the DOS 3.3 technique of reserving a safe space for a machine-language program by lowering HIMEM and then storing the program between the current and previous HIMEMs. Happily, there is an alternative method for freeing up space above HIMEM and we'll be examining it in Chapter 5.

Important Features of ProDOS and BASIC.SYSTEM

There are many useful features supported in a PRODOS- BASIC.SYSTEM environment that improve program execution speed and permit easy integration of non-Apple devices into the system. Here are some of the more important features:

Machine Language Interface. Probably the most important feature of ProDOS is the special disk command interpreter, called the machine language interface (MLI), which allows easy access to files using assembly language programming techniques. DOS 3.3 has no such interface, and is very cumbersome to deal with at this level. The MLI commands can be used to perform such standard file-handling chores as opening, reading, writing, closing, and so on; and the calling parameters for each command have been carefully defined by Apple. We'll be taking a detailed look at the MLI in Chapter 4.

Date-Stamping of Files. Whenever ProDOS creates or writes to files, it reads the current time and date from a clock card (if one is installed in the system) and then stores the information in the file's directory entry on the disk. When the disk is cataloged, the time and date of creation and of last modification is displayed next to the filename. As we will see in Chapter 8, ProDOS has an internal clock driver that can be called to read the time and date from a Thunderware Thunderclock, or a compatible clock card such as the Prometheus Versacard or the Applied Engineering Timemaster II. It is also possible to install clock drivers for other clock cards, and we'll see how to do this in Chapter 8 as well.

Disk Controller Card and Device Driver Protocols. One of the annoyances of DOS 3.3 is that it is extremely difficult to integrate foreign disk devices (non-Apple fixed disks, higher-density floppy disk drives, and so on) into the system. Not so with ProDOS. Apple has published a carefully defined disk controller protocol recognized by ProDOS that, if followed, permits such devices to be automatically installed on bootup. This protocol defines the addresses in the disk controller card ROM space at which information relating to the size of the volume, the characteristics of the volume, and the address of the disk driver subroutine responsible for performing disk I/O operations is stored. Apple has also defined how parameters are passed to a disk driver subroutine and how the driver can return error codes to the caller. We'll see how to write a disk driver subroutine in Chapter 7.

Improved Interrupt Handling. In Chapter 6 we will see that ProDOS automatically installs its own internal interrupt-handling subroutine which takes control whenever an active IRQ (interrupt request) signal is generated by an I/O device. This subroutine will, in turn, call up to four other subroutines that can be installed by the user to service such interrupts. This means that it is very simple to integrate an interrupt subroutine even though others may already be in use.

Hierarchical Directory Structure. Using ProDOS, it is possible to create several directories, each of which can contain several files, on one disk. This allows a common group of files to be conveniently arranged in one directory for easier access. The directories are organized in such a way that each is contained within another (called the *parent*); the path of directories ultimately leads back to the root directory (called the *volume* directory). The volume directory is the one that is created and named when the disk is first formatted. The hierarchical structure of directories will be analyzed in Chapter 2.

/RAM "Disk" Device. Both the Apple //c and the Apple //e (with an extended 80-column text card) have 64K of *auxiliary* memory in addition to the 64K of main memory that is normally used for program storage. ProDOS uses this memory space for program storage and retrieval (using standard disk commands) just as if it represented storage space on a floppy diskette or fixed disk. The RAM medium is called a RAMdisk. The main differences between using the RAMdisk and conventional disks are that I/O operations are executed much more quickly (after all, there are no mechanical parts to move about) and that the RAMdisk will vanish when the power is turned off. As we will see in Chapter 2, each disk installed in the system has a name associated with it (referred to as the *volume* name). The volume name for the RAMdisk is "/RAM" and we'll examine its characteristics in Chapter 7.

Extensibility of BASIC.SYSTEM. The BASIC.SYSTEM program defines a reasonably simple method that can be used to add more commands to the BASIC.SYSTEM command set. We'll see how this is done in Chapter 5.

"Separation of Powers." The low-level ProDOS command interpreter that performs all fundamental disk I/O operations is not mixed in, like in DOS 3.3, with the BASIC.SYSTEM interpreter that provides the set of "English" disk commands used in an Applesoft program. This means that if you wish to write another language interpreter, or a 100% assembly language program, you can save about 12K of memory space by loading it instead of BASIC.SYSTEM.

"-", The Intelligent Run Command. The "dash" command is a useful BASIC.SYSTEM command very popular with people who do not like to type. It is used to execute either an Applesoft program file (just like RUN does), a binary file (BRUN), or a textfile (EXEC). Dash automatically determines what type of file has been specified and then performs the steps needed to execute such a file. Dash can also be used to execute system program files like BASIC.SYSTEM, FILER, and CONVERT. (See Chapter 5 for a description of system programs. Briefly, a system program is a stand-alone assembly language program that defines a programming environment, or one that performs a specific function without relying on the presence of another system program.)

Useful Parameters. Many BASIC.SYSTEM commands support useful parameters that allow greater control (than possible with DOS 3.3) over how they are to be executed. For example, the ",@#" suffix (where "#" represents a line number) can be used with the BASIC.SYSTEM RUN command to load a program and then run it beginning at any line number. In addition, the ",E#" suffix (where "#" represents a memory address) can be used to specify an ending address when using a binary file command (BLOAD and BSAVE). You can also use a ",Ttype" suffix with BLOAD or BSAVE to work with any type of file other than standard BIN (binary) files. ("type" is the three-character mnemonic for the file type: BAS for BASIC, BIN for binary, TXT for text, and so on.) One other useful new parameter is ",F#"; it can be used when reading a textfile in order to skip over a specified number of fields (a field is a group of characters followed by a carriage return). The parameters recognized by BASIC.SYSTEM will be discussed in Chapter 5.

Speed. ProDOS performs disk I/O operations at the rate of about 8K bytes/second. This is significantly faster than the DOS 3.3 rate of about 1K bytes/second. Furthermore, BASIC.SYSTEM includes a version of the FRE command that can garbage collect Applesoft string variables much faster than the Applesoft command of the same name; BASIC.SYSTEM will also garbage collect automatically, before the slow Applesoft routine has a chance to do so. With BASIC.SYSTEM, garbage collection never takes more than a few seconds whereas, under DOS 3.3, it could take up to several minutes. [See Chapter 4 of *Inside the Apple //e* (Brady, 1985) for a description of the garbage collection process.]

File Size and Volume Size. ProDOS can deal with files that are up to 16 megabytes in size and with block-structured (disk-like) devices that can hold up to 32 megabytes of information (this includes the 5-megabyte ProFile fixed disk device).

Compatability with the Apple ///. The Apple /// SOS operating system organizes files on disk in the same way ProDOS does. This means that files created with ProDOS can be accessed by SOS, and vice versa. Furthermore, a ProDOS-formatted diskette can be booted on an Apple /// if the necessary SOS system files are stored on the diskette.

Chapter 2

FILES AND FILE MANAGEMENT

The purpose of this chapter is, first, to familiarize you with the concept of a ProDOS file and second, to explain how files are organized on disk media (such as floppy diskettes) and how they are identified. You will need to know this information if you want to fully understand the internal ProDOS file-handling commands we'll be studying in Chapter 4.

The concept of a file is, without exception, fundamental to all disk operating systems. A file is basically just a collection of data that can define an executable program, a typewritten document, a spreadsheet model, or other information that a program can deal with. The general structure of the file is defined by the operating system itself, and the operating system also provides the various commands that can be used to manipulate the file in different ways: create, open, read, write, close, destroy, rename, and so on.

Naming ProDOS Files

When you first store a file on the disk, you must assign it a unique filename that will be used to identify it thereafter. A ProDOS filename can be up to 15 characters long. It must begin with an alphabetic letter (A-Z), but the other characters may be any combination of letters, digits (0-9), and periods (.). Lowercase letters may also be used, but ProDOS automatically converts them to uppercase. Here are some examples of valid ProDOS filenames:

FORM.LETTER
CONTRACT.3
CHAPTER.FOUR

Here are some examples of invalid filenames and the reasons they are invalid:

5.EASY.PIECES	(starts with a number)
EXPLORING MARS	(contains an illegal space)
THIS&THAT	(contains an illegal &)
THIRD.AND.TWELVE	(too long)

Probably the most common mistake that arises in naming files is the use of the space as a word separator (see the second example). This is permitted with DOS 3.3 but not with ProDOS. Use a period instead of a space if you wish to improve the readability of your filenames.

Directories and Subdirectories

When a file is saved to disk, it can be stored in any one of several *directories* that can be created on the disk. These directories are somewhat analagous to file folders in that they are usually used to hold groups of related files. For example, you may create one directory to hold word processing documents, another to hold Applesoft programs, and so on. The ability to create separate directories on the same disk makes it much easier to organize large numbers of files efficiently.

When a disk is first formatted, only one directory, called the *volume* directory, exists; you name it right after the formatting process finishes (the rules for naming directories are the same as for naming ordinary files). The volume directory for a standard Disk][diskette or the ProFile fixed disk can hold the names of up to 51 files (contrast this with the 105 files that a DOS 3.3 directory can handle).

You can create additional directories (called *subdirectories*) within the volume directory using the BASIC.SYSTEM or ProDOS CREATE command. Indeed, you can even create subdirectories within subdirectories (64 levels are permitted). Each subdirectory can hold the names of as many files as you wish to store in it (although at some point the disk will become full). This system of nested directories is called a hierarchical directory structure.

The directory in which a file is to be saved is normally specified by tacking on a special prefix to the filename to create a unique identifier called a *pathname*. A pathname is made up of the names of a series of directories, beginning with the name of the volume directory and continuing with the names of all the directories that must be passed through in order to reach the directory you want, followed by the filename itself. Each directory name is separated from the next by a slash ("/"), and a slash must precede the name of the volume directory. The directory names in the chain must define a continuous path: each directory specified must be contained within the preceding directory.

For example, consider a disk that has a volume directory called BASEBALL and two subdirectories within BASEBALL called AMERICAN and NATIONAL. (Such a directory scheme is shown in Figure 2-1.) If you want to save a file called NY.YANKEES in the AMERICAN subdirectory, then you would specify the following pathname:

```
/BASEBALL/AMERICAN/NY.YANKEES
```

If you simply specified the name NY.YANKEES itself, then the file would be saved in the currently active directory, which is usually the volume directory (unless it has been changed using the PREFIX command that we are about to describe).

If most of the files that you are using are contained in the same subdirectory, it quickly becomes very annoying to have to specify the same chain of directory names every time a file is to be used. To alleviate this annoyance, ProDOS supports a PREFIX command that can be used to set the chain of directory names to which any filename specified in a ProDOS command will be automatically appended. (The chain must not be more than 64 characters long.) For example, if PREFIX is set by entering the following BASIC.SYSTEM command:

```
PREFIX/BASEBALL/AMERICAN/
```

then any file contained in the directory at the end of this path (such as NY.YANKEES) can be referred to by entering its filename only.

A name that is a continuation of the prefix could also be entered to access files in lower-level subdirectories; such a name is called a *partial pathname*. If PREFIX has the value just described and if AMERICAN contains a subdirectory called CHAMPS that contains a file called TIGERS.1984, then you could access the file by specifying a partial pathname of CHAMPS/TIGERS.1984. Note that in this case the pathname is not preceded by a slash.

ProDOS prefixes can be up to 64 characters long, including the preceding slash. Partial pathnames can also be up to 64 characters long.

One of the useful features of ProDOS is that whenever a ProDOS or BASIC.SYSTEM command must find the file described by a pathname, it searches each installed disk drive until it is found. Contrast this with the DOS 3.3 environment where it is necessary to explicitly specify the drive and slot number for the file before it can be accessed (using the ",S#" and ",D#" parameters). BASIC.SYSTEM, for reasons of compatibility, also permits the use of the ",S#" and ",D#" parameters, however. If a filename or partial pathname is specified in a command line, and no prefix has yet been defined, or if either the slot or drive parameter is used, BASIC.SYSTEM will automatically use the name of the volume directory specified by the slot and drive parameters (or their defaults) to create the full pathname.

The advantage of subdirectories is often not readily apparent to users of floppy diskettes, but becomes obvious when a fixed disk device where there is enough room for hundreds of files, like the ProFile, is used. If all the files were held in one directory you might have to wait a long time to spot your file when

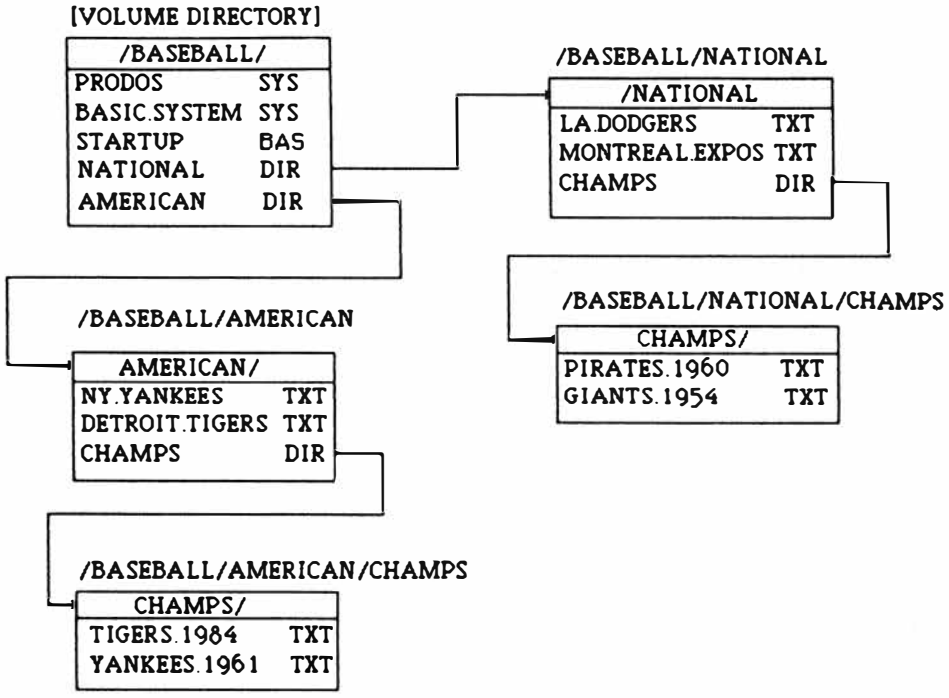


Figure 2-1. Diagram of the BASEBALL directory

the disk was cataloged, and even then you could well miss it among the other files. Fortunately, the hierarchical directory structure used by ProDOS allows related files to be grouped within the same subdirectory for easy access.

Fundamental File-Handling Concepts

As we will see in Chapter 4, ProDOS contains a low-level command interpreter called the MLI (machine language interface) that can perform various file-handling chores. The most common MLI commands used with an existing file are:

- OPEN (open the file for I/O operations)
- READ (read from the file)
- WRITE (write to the file)
- CLOSE (close the file to I/O operations)

Four similar commands are also available in an Applesoft environment when you are using the BASIC.SYSTEM interpreter. Let's review each of these fundamental file-handling operations right now.

Opening a File

Before a file can be accessed, it must first be opened. This is done by using the ProDOS MLI OPEN command and specifying the name of the file to be acted on. ProDOS opens a file by first locating it on the disk and then setting up a special 1024-byte (1K) file buffer for it in memory.

Half of the file buffer holds information that tells ProDOS where the file data is located on the disk; the other half is used to store the most recently accessed portion of the file. Whenever a file I/O operation is requested, ProDOS automatically determines whether the portion of the file to be accessed is already sitting in the file buffer. If it is, then ProDOS does not need, nor does it bother, to access that portion of the file from the disk. Instead, it simply stores the data in the buffer (a write operation) or reads the data from the buffer (a read operation). As a result, file operations are performed much more quickly than if unbuffered disk I/O techniques were used.

ProDOS can open a file at one of five different levels (numbered from 0 to 4). This is done by storing the level number at a particular memory location (LEVEL: $BF94) just before opening the file. The advantage of using different levels is that it is possible (using the ProDOS MLI CLOSE command) to close all files at or above a particular level number without disturbing any files that were opened at a lower level.

Reading and Writing a File

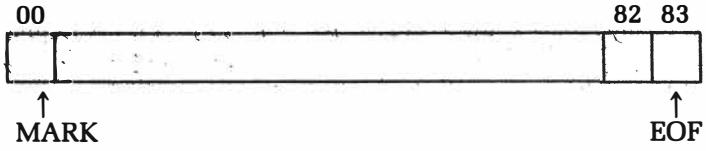
Whenever a file is opened, ProDOS initializes two important internal pointers, called EOF and MARK, that allow it to keep track of the size of the file and the position in the file that is currently being accessed. See Figure 2-2.

EOF is the end-of-file pointer and it always points to the byte after the last byte in the file. If you try to read data from the file past this position, an error will occur (the "end of data" error). EOF normally only changes if information is written to the end of a file; if this happens, EOF will automatically be increased by the appropriate number of bytes and, if necessary, further space on the disk will be allocated. As we will see in Chapter 4, however, ProDOS also supports an MLI command (SET_EOF) that can be used to set EOF to any value you want.

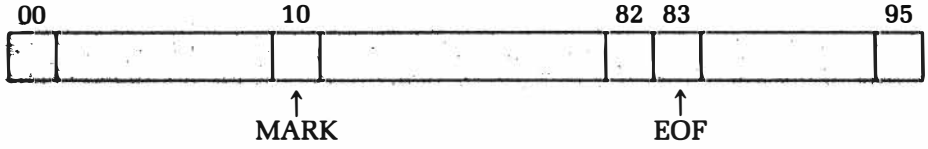
MARK is also called the *position-in-the-file* pointer and it always contains the position at which the next read or write operation will take place. It is set to 0 (the beginning of the file) when the file is first opened, but it will automatically increase as information is read from or written to the file. For example, if MARK is currently 10 (that is, it is pointing to the eleventh byte in the file), and you read or write fourteen more bytes of information, MARK will be set to 24.

It is also possible to explicitly set MARK to any position in the file so that the file can be accessed directly or *randomly*. This means that if fixed-length records are stored in the file, they can be retrieved very quickly since there is no need to read through all preceding records first.

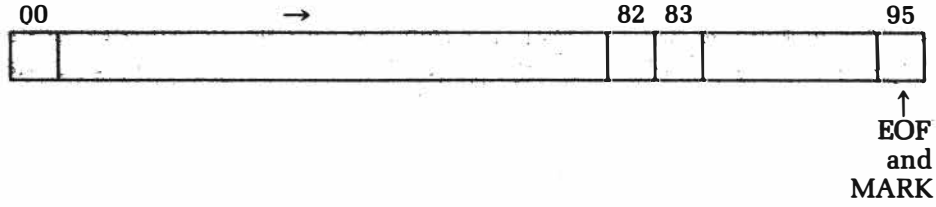
(a) EOF and MARK after an 83-byte file has been opened:



(b) EOF and MARK after 10 bytes of the file have been read:



(c) EOF and MARK after 12 bytes have been written past the end of the file (an append operation):



Note: EOF is automatically extended.

Figure 2-2. The ProDOS EOF and MARK pointers

Closing a File

A file must be closed after use. This serves two main purposes. First, it ensures that any data written to the file buffer, but not yet stored on the disk itself, is actually stored. Second, it causes file information (such as size) in the directory to be updated.

Although it is not necessary to close a file immediately after you're finished with it (you could wait until the program is about to end), it makes good sense to do so in order to reduce the risk of data loss that would occur in the event of an unexpected power loss or a system reset. In addition, ProDOS allows only eight files to be open simultaneously; if you have a lot of inactive, but open, files lingering around, you could be faced with a surprising error message the next time that you open a file. Another compelling reason to close unused files is to free up valuable memory space; each open file reserves a 1K buffer area that will not be made available to the system until the file is closed.

ProDOS File Management

Every disk operating system uses a unique method to organize files on disk and to keep track of what parts of the disk are being used for data storage so that files can be easily and efficiently created, deleted, and accessed. In this section, we will discuss:

- the nature of a ProDOS-formatted diskette
- the structure of the volume bit map that keeps track of block usage on the disk
- the structure of a ProDOS directory and subdirectory
- the structure of a directory entry
- the indexing schemes used to keep track of the data blocks used by a file

ProDOS uses the same general method to organize files on every block-structured, mass-storage device with which it is compatible (such as a Disk][, a ProFile, the /RAM volume, and so on). Specific differences will arise because the storage capacities of these different devices vary and so certain size parameters that ProDOS stores on the media will change. Furthermore, the sizes of two important data structures stored on the media, the volume directory and the volume bit map, might be different. Generally speaking, we will focus on the Disk][(and its 5.25-inch floppy diskettes) in this section; any implementation differences for other devices that are not obvious will be mentioned.

Formatting the Disk Medium

Before a floppy diskette (or any other disk medium) can be used by ProDOS it must be formatted into a state that ProDOS recognizes. This is done using a program called FILER provided on the ProDOS master diskette; other disk media can be formatted by similar programs that are supplied with the disk drive.

The method used to format a disk will depend on the nature of the disk device. Let's look at the method used for the most popular type of medium, a standard 5.25-inch floppy diskette. When you format such a diskette, templates for 35 *tracks* on the diskette are created (numbered from 0 to 34), each of which can hold 4096 bytes of information. These tracks are arranged in concentric rings around the central hub of the disk, with track 0 being located at the outside edge and track 34 at the inside edge. ProDOS can access any track by causing a read/write head (located inside the disk drive) to move to the desired track. This is done using I/O locations which activate a small *stepping motor* that controls the motion of a metal arm to which the read/write head is connected. This arm moves along a radial path beginning at the outside edge of the diskette (track 0) and ending at the inside edge (track 34).

Each of the 35 tracks formatted on a diskette are subdivided into sixteen smaller units called *sectors*. A sector is the smallest unit of data that can be

written to or read from the diskette at one time. The sectors that make up a track are numbered from 0 to 15 and each can hold 256 bytes of information. If you do the mathematics, you will quickly determine that a diskette can hold 560 sectors (140K) of information.

This is the last you'll hear about sectors, however, since ProDOS uses the 512-byte block as the basic unit of file storage; each block is made up of two diskette sectors. An initialized diskette is made up of 280 such blocks (numbered from 0 to 279). Fortunately, it is rarely necessary to know where these blocks are actually located on the diskette, since the internal ProDOS disk driver subroutine automatically maps block numbers to actual physical locations on the diskette.

Block Usage

We are now ready to examine the method that ProDOS uses to manage files on a disk. This will include an analysis of the structures of the directories that hold information concerning files, of the volume bit map that keeps track of block usage on the disk, and of the index blocks that contain the location of the data blocks used by each file.

Before we continue, however, keep in mind that the following descriptions relate to ProDOS only and not to its predecessor, DOS 3.3, or to the Apple Pascal operating system.

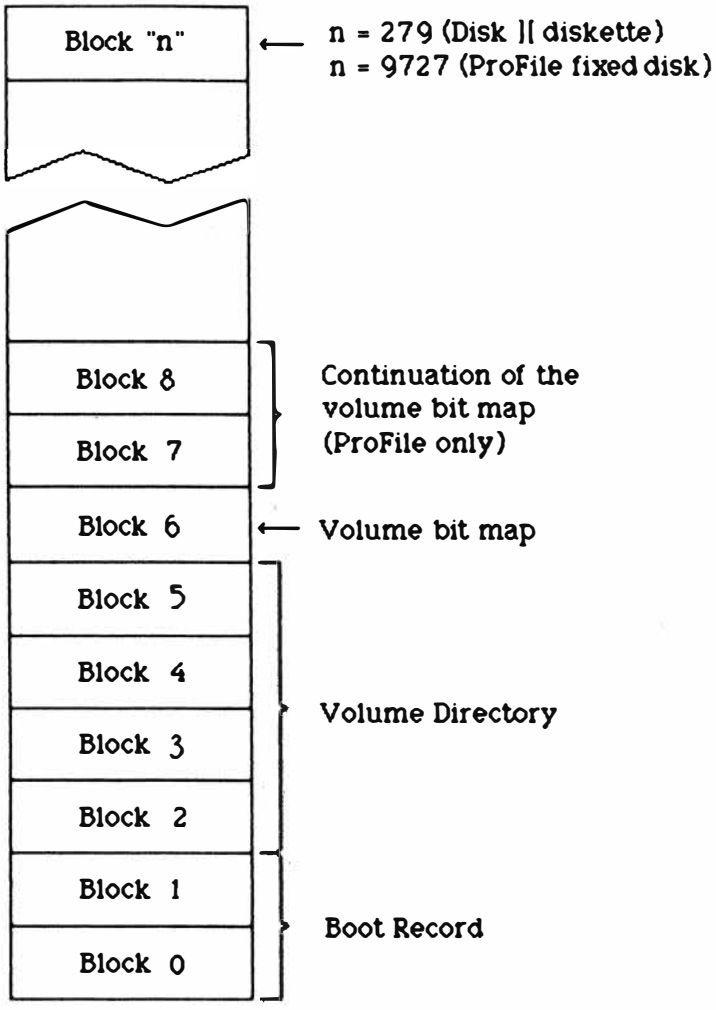
As we have seen, a total of 280 blocks, holding 140K of data, are available for use on a ProDOS-formatted diskette. If Apple's standard disk-formatting program is used, however, seven of these blocks (0–6) are not available for use by files because they are reserved for special purposes. By way of comparison, the capacity of the ProFile fixed disk is 9728 blocks (4,864K), nine of which (0–8) are reserved. Figure 2-3 shows the usage of blocks on a freshly formatted diskette or fixed disk.

Blocks 0 and 1 contain a short assembly language program that is automatically loaded into memory and executed by the firmware on the Disk][or ProFile controller card whenever a disk is booted. This program is called the bootstrap loader and it will locate, load, and execute a special system file called PRODOS if it finds it on the disk. (A system file is one that has a file type code of $FF and a CATALOG mnemonic of SYS; we'll be discussing file type codes later in this chapter.) PRODOS is the program that is ultimately responsible for installing and activating the ProDOS operating system and then loading the active programming environment (such as the environment defined by BASIC.SYSTEM).

Blocks 2 through 5 are the blocks that contain the volume directory for the disk. The structure of this directory will be described following the next section.

Block 6 contains the volume bit map for the disk. Each bit in the map is used to indicate whether the block to which it corresponds is free or in use. The ProFile also uses blocks 7 and 8 to store its volume bit map.

The remaining blocks, 273 for a Disk][diskette or 9720 for a ProFile fixed disk, are free for use by files that are stored on the disk.



Each block holds 512 bytes.
Total storage capacity is 280 blocks (140K) for a Disk][diskette.
Total storage capacity is 9728 blocks (4,864K) for a ProFile fixed disk.

Figure 2-3. Map of block usage on a Disk][diskette and a ProFile fixed disk

The Volume Bit Map

The volume bit map is used to keep track of which blocks on the disk are in use and which are free. ProDOS reads the bit map whenever it must allocate

new space to a file so that it can quickly locate free blocks on the disk. ProDOS writes to the bit map when new file blocks must be reserved (this occurs when an existing file is extended or a new one is saved), or when blocks must be freed up (this occurs when a file is shortened or deleted.)

The standard formatting subroutine for Disk][diskettes and the ProFile fixed disk uses block 6 as the volume bit map. Note, however, that block 6 is only the conventional location for the bit map and it is permissible to store the map in any free block on the disk. For example, the volume bit map for the /RAM volume is stored in block 3. As we will see in the next section, the block number used for the bit map is stored in the directory header that describes the characteristics of the disk volume.

For a Disk][diskette, only the first 35 bytes (280 bits) in the volume bit map block are actually used and each bit in each byte corresponds to a unique block number. A one-block bit map such as this can handle volumes of up to 4096 blocks. For larger volumes, such as the ProFile, ProDOS expects to find a continuation of the bit map in the blocks on the disk that immediately follow the first one used. For example, the 9728- block ProFile requires three blocks for its bit map; the standard formatting program stores the first part of the map in block 6 and the continuation in blocks 7 and 8. (ProDOS determines the size of the volume bit map by examining two bytes in the volume directory header that hold the size of the disk; these bytes are placed there by the program that is used to format the disk. We will describe volume directory headers in a later section.)

The structure of the volume bit map for Disk][diskettes is shown in Figure 2-4. As can be seen, the bits in each byte in the bit map block reflect the states of eight contiguous blocks; bit 0 corresponds to the highest-numbered block in the octet and bit 7 to the lowest-numbered. If the bit corresponding to a particular block is 1, then that block is free. If it is 0, then it is being used by a stored file.

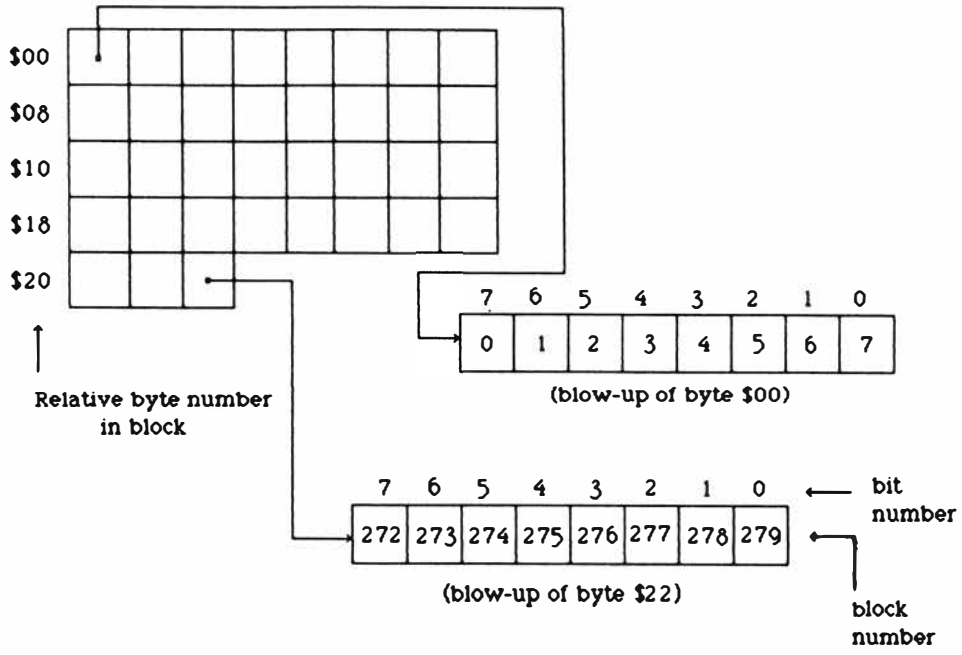
The byte number (from 0 to 34), and the bit number within that byte (from 0 to 7), that corresponds to any given block number can be calculated using the following Applesoft formulas:

```
BYTENUM = INT(BLOCKNUM/8)
BITNUM  = 7 - BLOCKNUM - 8 * BYTENUM
```

Volume Directories and Subdirectories

A directory is an intricate data structure that ProDOS uses to hold important information concerning each file on the disk. This includes the file name, type, size, creation date, the location of the file's data, and so on. Without this information it would be impossible to efficiently manage multiple files on a disk.

As we saw earlier, ProDOS permits multiple directories to be created on one disk. With the exception of the volume directory (the one through which



Each byte in the volume bit map defines the states of 8 contiguous blocks. The bit corresponding to a given block number can be calculated by first dividing the block number by 8; the whole part of the result gives you the byte number that is involved. To get the specific bit number within that byte, subtract the remainder from 7.

Figure 2-4. The ProDOS volume bit map for Disk][diskettes

all the others must be accessed), these directories can be stored just about anywhere on the disk since they are treated almost like standard files. The volume directory, however, always begins at block 2; if the standard formatting program for the Disk][or ProFile is being used, it will also occupy blocks 3, 4, and 5.

A ProDOS directory is an example of a doubly linked list data structure. The links are actually pairs of two-byte pointers that are stored at the beginning of each directory block. One of these pointers (bytes $00/$01) contains the number of the previous directory block in the chain—or zero if there is no previous block—and the other (bytes $02/$03) contains the number of the next directory block—or zero if there is no ensuing block. This allows very large directories to be created.

Each block used by any directory can hold up to thirteen 39-byte file entries. (This means that the four-block volume directory used with the Disk][can hold a total of 52 entries, one of which is an entry that holds the volume name itself.) The map of a directory block is shown in Table 2-1.

Table 2-1. Map of a ProDOS directory block.

Byte number in Directory Block	*Meaning of Entry*
$000-$001	Block number of the previous directory block (low byte first). This will be zero if this is the first directory block.
$002-$003	Block number of the next directory block (low byte first). This will be zero if this is the last directory block.
$004-$02A	Directory entry for file #1 or, if this is the key (first) block of the directory (bytes $00 and $01 are both 0), the directory header.
$02B-$051	Directory entry for file #2
$052-$078	Directory entry for file #3
$079-$09F	Directory entry for file #4
$0A0-$0C6	Directory entry for file #5
$0C7-$0ED	Directory entry for file #6
$0EE-$114	Directory entry for file #7
$115-$13B	Directory entry for file #8
$13C-$162	Directory entry for file #9
$163-$189	Directory entry for file #10
$18A-$1B0	Directory entry for file #11
$1B1-$1D7	Directory entry for file #12
$1D8-$1FE	Directory entry for file #13
$1FF	[Not used]

The Directory Header

The first block used by a directory (or subdirectory) is called the *key block* and is configured slightly differently than the others. The difference is that the 39-byte entry that normally describes the first file in the block is instead used to describe the directory itself. This entry is called the *directory header*.

The meaning of each of the 39 bytes that make up a directory header are shown in Table 2-2. Notice the differences between the header for a volume directory and the header for a subdirectory that appear at absolute positions $27–$2A in the block.

Standard Directory Entries

All directory entries, other than the directory header entry, represent either standard data files (for example, binary files, textfiles, and Applesoft programs) or subdirectory files. The formats of the directory entries for both of these two types of files are virtually identical and are as shown in Table 2-3.

Table 2-2. Map of a ProDOS directory header.

Byte number in Key Block	*Description (Disk][entries in parentheses)*
$04	High four bits: storage type code - $F for a volume directory - $E for a subdirectory Low four bits : length of directory name
$05-$13	Directory name (in standard ASCII with bit 7 = 0); the length of the name is contained in the low-order half of byte $04
$14-$1B	[Reserved]
$1C-$1D	The date on which this directory was created; (Format: MMMDDDDD YYYYYYYM, see Figure 8-1)
$1E-$1F	The minute (byte $1E) and hour (byte $1F) at which this directory entry was created; (Format: see Figure 8-1)
$20	The version number of ProDOS that created this directory; see Appendix II for a list of version numbers
$21	The lowest version of ProDOS that is capable of using this directory; see Appendix II for a list of version numbers
$22	The access code for this directory (see Figure 4-2 in Chapter 4)
$23	The number of bytes occupied by each directory entry (39)
$24	The number of directory entries that can be stored on each block (13)
$25-$26	The number of active files in this directory (not including the directory header)
$27-$28	VOLUME DIRECTORY: The block where the volume bit map is located (6) SUBDIRECTORY: the block in which the entry defining this subdirectory is located (this is in the parent directory of the subdirectory)
$29-$2A	VOLUME DIRECTORY: The size of the volume in blocks (280)
$29	SUBDIRECTORY: The directory entry number within the block given by $27/$28 that defines this subdirectory (1 to 13)
$2A	SUBDIRECTORY: The number of bytes in each directory entry of the parent directory (39)

File Type Codes

The only way to determine the general nature of the file to which a particular file entry corresponds is to examine the file type code that appears at relative position $10 within the entry. Although 256 different codes are possible, only

Table 2-3. Map of a ProDOS directory file entry.

Relative Byte Number Within Entry	*Meaning of Entry*
$00	High four bits: storage type code (see text) - $0 for an inactive (or deleted) file - $1 for a seedling file - $2 for a sapling file - $3 for a tree file - $D for a subdirectory file Low four bits : length of file name
$01-$0F	File name (in standard ASCII with bit 7 = 0)
$10	File type code (see Table 2-4)
$11-$12	Key pointer; if a subdirectory file, the block number of the key block of the subdirectory; if a standard file, the block number of the index block of the file (or the sole data block if this is a seedling file)
$13-$14	Size of the file in blocks
$15-$17	End-of-file (EOF) position; this is the size of the file in bytes (low-order bytes first)
$18-$19	The date on which this file was created; (Format: MMMDDDDD YYYYYYYM, see Figure 8-1)
$1A-$1B	The minute (byte $1A) and hour (byte $1B) at which this file was created; (Format: see Figure 8-1)
$1C	The version number of ProDOS that created this file; see Appendix II for a list of version numbers
$1D	The lowest version of ProDOS that is capable of using this file; see Appendix II for a list of version numbers
$1E	The access code for this file (see Figure 4-2 in Chapter 4)
$1F-$20	The auxiliary type code for the file; this code is used for special purposes by each system program; for example, BASIC.SYSTEM stores the default loading address here (for a binary file) or the field length (for a textfile); it also stores $801 here if the file is an Applesoft program file
$21-$22	The date on which this file was last modified; (Format: MMMDDDDD YYYYYYYM, see Figure 8-1)
$23-$24	The minute (byte $23) and hour (byte $24) at which this file was created; (Format: see Figure 8-1)
$25-$26	The block number of the key block of the directory that holds this file entry

a few are commonly used with ProDOS by BASIC.SYSTEM, and they are shown in Table 2-4. The three-character mnemonics used to represent these file types in a CATALOG listing are also shown in Table 2-4. If other file type codes are

used, the CATALOG mnemonic will be of the form "$HH" where "HH" represents the two-digit hexadecimal file type code. All file type codes except $F1-$F8 are reserved for use by ProDOS; user programs may freely use these non-reserved codes for any purpose.

The meaning of the content of any specific file type is actually dependent on the program that created the file in the first place. For example, in a BASIC.SYSTEM environment, several file type codes are used to identify files that contain specific information useful in an Applesoft environment. Let's look at five of the most common file types used by BASIC.SYSTEM.

Table 2-4. ProDOS file type codes.

File Type Code	*CATALOG Mnemonic*	*Type of File*
$00		Typeless file
$01		Bad blocks file
$04	TXT	ASCII text file
$06	BIN	Binary file
$0F	DIR	Directory file
$19	ADB	AppleWorks database file
$1A	AWP	AppleWorks word processing file
$1B	ASP	AppleWorks spreadsheet file
$C0-$EE		[Reserved for future use]
$EF	PAS	Pascal file
$F0	CMD	ProDOS added command file
$F1-$F8		[User-defined files]
$FA	INT	Integer BASIC program file
$FB	IVR	Integer BASIC variable file
$FC	BAS	Applesoft program file
$FD	VAR	Applesoft variable file
$FE	REL	EDASM relocatable code file
$FF	SYS	ProDOS system file

CATALOG mnemonics for the blank entries in this table are displayed as "$HH", where "HH" represents the hexadecimal file type code.

Note: All other file type codes are reserved for use by the Apple /// SOS operating system.

TXT (code $04, Figure 2-5). This type of file usually contains ASCII-encoded text. ("Standard" ASCII codes, with bit 7 cleared to zero, are used. Note that DOS 3.3 creates textfiles that contain codes with bit 7 set to 1.) Each line of text is terminated by a carriage return code ($0D) and, if it's a standard sequential textfile (that is, one where the lines of text are stored right after one another), the last byte in the file is usually followed by a $00 end-of-file marker (the exact size of the file is stored in its directory entry). The other general type of textfile, the random-access textfile, is made up of many fixed-length records, each of which can contain several lines of text. Each line of text is called a field entry.

If the number of characters stored in a record is less than the record size, the rest of the record will be padded out with $00 bytes; these $00 bytes are *not* end-of-file markers. The record length of a textfile is stored as the auxiliary code in the directory entry (at relative bytes $1F/$20); if the number stored here is zero, then it is a sequential textfile.

This program . . .

```
100 PRINT  CHR$ (4);"OPEN TEXTFILE"
200 PRINT  CHR$ (4);"WRITE TEXTFILE"
300 PRINT "THIS IS A TEST"
400 PRINT "AND SO IS THIS"
500 PRINT  CHR$ (4);"CLOSE"
```

Generates this (sequential) TXT file . . .

```
0000: 54 48 49 53 20 49 53 20 THIS IS
0008: 41 20 54 45 53 54       A TEST
000E: 0D                      (carriage return)
000F: 41 4E 44 20 53 4F 20    AND SO
0016: 49 53 20 54 48 49 53    IS THIS
001D: 0D                      (carriage return)
```

Note that the text is stored as standard ASCII codes (that is, with bit 7 = 0); DOS 3.3 stores text as "negative" ASCII codes (with bit 7 = 1).

The size of a TXT file is stored at relative bytes $15–$17 in its directory entry.

The auxiliary type code for a TXT file (stored at relative bytes $1F adn $20 in the file's directory entry is its record length; it is zero for a sequential textfile.

Figure 2-5. The structure of a TXT file

BAS (code $FC, Figure 2-6). This type of file contains an Applesoft program in its standard compressed and tokenized form. Tokens are one-byte codes for Applesoft keywords such as PRINT, INPUT, and so on. (For a detailed description of this form, refer to Chapter 4 of *Inside the Apple* //e.) A BAS file is automatically created by using the BASIC.SYSTEM SAVE command to transfer the image of the Applesoft program that starts at the address pointed to by $67/$68 (usually $801) and ends at the address pointed to by $AF/$B0. The auxiliary type code for such a file is usually $801, the standard loading address for an Applesoft program.

This Applesoft program . . .

```
100 TEXT : HOME
200 VTAB 12: HTAB 10
300 PRINT "THIS IS A 'BAS' FILE"
400 VTAB 22
```

Is stored as this BAS file . . .

```
0000: 09 08                                  [address of next
                                             line]
0002: 64 00                                  [line number =
                                             100]
0004: 89                                     [token for TEXT]
0005: 3A                                     :
0006: 97                                     [token for HOME]
0007: 00                                     [end of line]
0008: 15 08                                  [address of next
                                             line]
000A: C8 00                                  [line number =
                                             200]
000C: A2                                     [token for VTAB]
000D: 31 32                                  12
000F: 3A                                     :
0010: 96                                     [token for HTAB]
0011: 31 30                                  10
0013: 00                                     [ end of line]
0014: 31 08                                  [address of next
                                             line]
0016: 2C 01                                  [line number =
                                             300]
0018: BA                                     [token for PRINT]
0019: 22 54 48 49 53 20 49 53                "THIS IS
0021: 20 41 20 27 42 41 53 27                A 'BAS'
0029: 20 46 49 4C 45 22                       FILE"
002F: 00                                     [end of line]
0030: 39 08                                  [address of next
                                             line]
0032: 90 01                                  [line number =
                                             400]
0034: A2                                     [token for VTAB]
0035: 32 32                                  22
0037: 00                                     [end of line]
0038: 00 00                                  [end of program]
```

The size of a BAS file is stored at relative bytes $15–$17 in its directory entry.

The auxiliary type code for a BAS file (stored at relative bytes $1F and $20 in the file's directory entry) is simply the address stored in the start-of-program pointer ($67/$68) when the program was saved; this address is usually $0801.

Figure 2-6. The structure of a BAS file

BIN (code $06, Figure 2-7). A BIN file is a general-purpose binary data file that can contain just about anything: programs, data, text, and so on. It is created using the BASIC.SYSTEM BSAVE command. The exact meaning of the contents of such a file cannot be generalized although many such files contain executable

6502 code. The auxiliary type code for a BIN file is the address from which it was BSAVEd to disk.

This program . . .

```
            ORG $300

HALFTIME    DFB $00
LENGTH      DFB $00

            LDY #255
            LDA LENGTH
            STA $325
NOTE1       LDX HALFTIME
            LDA $C030
            JMP $31A
```

Is stored as this BIN file . . .

```
0000: 00                    DFB $00
0001: 00                    DFB $00
0002: A0 FF                 LDY #$FF
0004: AD 01 03              LDA $0301
0007: 8D 2F 03              STA $032F
000A: AE 00 03              LDX $0300
000D: AD 30 C0              LDA $C030
0010: 4C 1A 03              JMP $031A
```

The size of a BIN file is stored at relative bytes $15–$17 in its directory entry.

The auxiliary type code for a BIN file (stored at relative bytes $1F and $20 in the file's directory entry) is its loading address—$300 in the above example.

Figure 2-7. The structure of a BIN file

SYS (code $FF). A SYS file is just like a BIN file except that it is expected to contain an executable program called a system program or interpreter. The characteristics of a standard system file will be described in Chapter 5.

VAR (code $FD, Figure 2-8). A VAR file contains a set of Applesoft program variables in a special packed form. It is automatically created by using the BASIC.SYSTEM STORE command and can be reloaded using the RESTORE command. The first five bytes of a VAR file are used to store the total length of the simple (undimensioned) and array (dimensioned) variable tables that are created by an Applesoft program (2 bytes), the length of the simple variable space itself (2 bytes), and the HIMEM page number in effect when the file was saved (1 byte). These bytes are followed by the images of the two variable tables, and, finally, the contents of each of the string variables. The auxiliary type code for a VAR file contains the address from which the image of the compressed

variables were stored. (For a description of the structure of the Applesoft variable tables, refer to Chapter 4 of *Inside the Apple //e*.

This program . . .

```
100 A = 1:B% = 2:C$ = "TEST":D$ = "REPEAT"
200 DIM E(3):E(0) = 0:E(1) = 1:E(2) = 2:E(3) = 3
300 PRINT  CHR$ (4);"STORE VARS"
```

Generates this VAR file . . .

```
0000: 37 00                     Size of entire
                                variable table
0002: 1C 00                     Size of simple
                                variable table
0004: 96                        HEMEM page number
0005: 41 00                     Variable name (A)
0007: 81 00 00 00 00             -- value (1)
000C: C2 80                     Variable name (B%)
000E: 00 02 00 00 00             -- value (2)
0013: 43 80                     Variable name (C$)
0015: 04 FC 95 00 00             -- length+pointer
001A: 44 80                     Variable name (D$)
001C: 06 F6 95 00 00             -- length+pointer
0021: 45 00                     Variable name (E)
0023: 1B 00 01 00 04             -- dimensioning
                                 bytes
0028: 00 00 00 00 00             -- E(0)=0
002D: 81 00 00 00 00             -- E(1)=1
0032: 82 00 00 00 00             -- E(2)=2
0037: 82 40 00 00 00             -- E(3)=3
003C: 52 45 50 45 41 54          -- REPEAT
0042: 54 45 53 54                -- TEST
```

The size of a VAR file is stored at relative bytes $15–$17 in its directory entry.

The auxiliary type code for a VAR file (stored at relative bytes $1F and $20 in the file's directory entry) is the starting address of the block of variables saved to the file.

Figure 2-8. The structure of a VAR file

File Access Codes

Relative byte $1E within each directory entry is a one-byte code that contains four bits that reflect the read (bit 0), write (bit 1), rename (bit 6), and destroy (bit 7) status of the file. A fifth bit (bit 5) acts as a flag to indicate whether the file has been modified since the last time it was backed up. (It is the back-up program's responsibility to clear this bit to 0 when it makes a copy of the file;

Apple's FILER program does *not* clear the bit.) If a bit is set to 1, then the file is enabled for the operation associated with that bit. The three remaining bits (bits 2, 3, and 4) are not used and are always set to 0. See Figure 4-2 in Chapter 4 for a detailed description of the access code byte.

The BASIC.SYSTEM LOCK and UNLOCK commands can also be used to affect the file's access status: LOCK disables write, rename, and destroy accesses; UNLOCK enables them. A locked file can be easily identified because an asterisk is displayed to the left of its name in a CATALOG listing. The asterisk will also be displayed if only one or two of these three types of access modes is disabled. If the file is just read disabled, however, the asterisk will not appear, but a FILE LOCKED error message will be displayed if an attempt is made to read the file using a BASIC.SYSTEM command.

Unfortunately, there is no BASIC.SYSTEM command that allows individual bits of the file access code to be set or cleared so that a particular security level can be easily associated with a file. As we will see in Chapter 4, however, this can be done by using the ProDOS SET _ FILE _ INFO MLI command to set the code to any valid quantity.

Time and Date Formats

Each directory entry contains eight bytes that hold the creation and modification time and date for the file that it describes. The formats for the time and date bytes are the same as those shown for TIME and DATE in Figure 8-1 in Chapter 8.

Organizing File Data

ProDOS uses an efficient tree-structured indexing scheme to keep track of the blocks that hold the data for any particular non-directory file on the disk. In the most common implementation of this scheme (the one that is used for files that are between 2 and 256 data blocks in length), the key block pointer in the file's directory entry (stored at relative bytes $11 and $12) points to an index block that contains an ordered list of the numbers of each block on the disk that the file uses to store its data. The main advantage of using an indexing scheme like this is that a file can occupy any collection of blocks on the disk and not just a group of consecutive ones. (The Apple Pascal operating system, for example, forces a file to use a group of consecutive blocks.) This means that no space on the disk will be wasted. The disadvantage is that disk I/O operations are performed more slowly than Apple Pascal, for example, because it takes longer to position the disk read/write head over the blocks of a highly fragmented file.

There are actually three variants of this general indexing scheme that are used by ProDOS; the one that is used depends on the size of the file being dealt with. The following woodsy classifications are used to describe the three basic file sizes:

Seedling file	1 to 512 bytes (1 data block only)
Sapling file	513 to 131,072 (128K) bytes (up to 256 data blocks)
Tree file	131,073 (128K+1) to 16,777,215 (16M−1) bytes (up to 32768 data blocks)

The indexing scheme used by a non-directory file can be determined by examining the storage type code number stored in the high-order four bits of the 0th entry in its directory entry. The number is 1 for a seedling file, 2 for a sapling file, and 3 for a tree file. If the number is 0, the file has been deleted. (Directory files use storage type codes of $D, $E, or $F; code $D is used to identify a directory entry for a subdirectory file, code $E to identify the header for a subdirectory, and code $F to identify the header for a volume directory.)

As we have just seen, a file's key pointer (relative bytes $11 and $12 of its directory entry) points to an index block (also called the *key block*) used by the file. Let's take a look at how ProDOS interprets the index block for each of the three types of files.

Seedling File. A seedling file cannot, by definition, exceed 512 bytes, so it uses only one block on the disk for data storage. This is the block number stored in the key pointer. This means that this block is not really an index block at all; it simply holds the contents of the file.

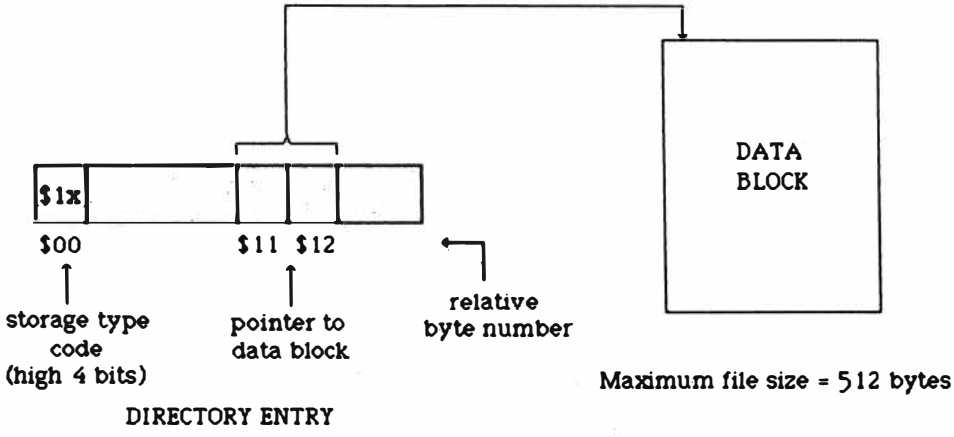


Figure 2-9. The structure of a seedling file

Sapling File. The key pointer for this type of file holds the block number of a standard index block that contains an ordered list of the block numbers on the disk that are used to store that file's data. Table 2-5 shows what an index block for a sapling file looks like. Since block numbers can exceed 255, two bytes are needed to store each block number. The low part of the block number is always stored in the first half of the block and the high part is stored 256 bytes further into the block. The maximum size of a sapling file is 128K; it cannot be larger

than this since only 256 blocks (of 512 bytes each) can be pointed to by the index block.

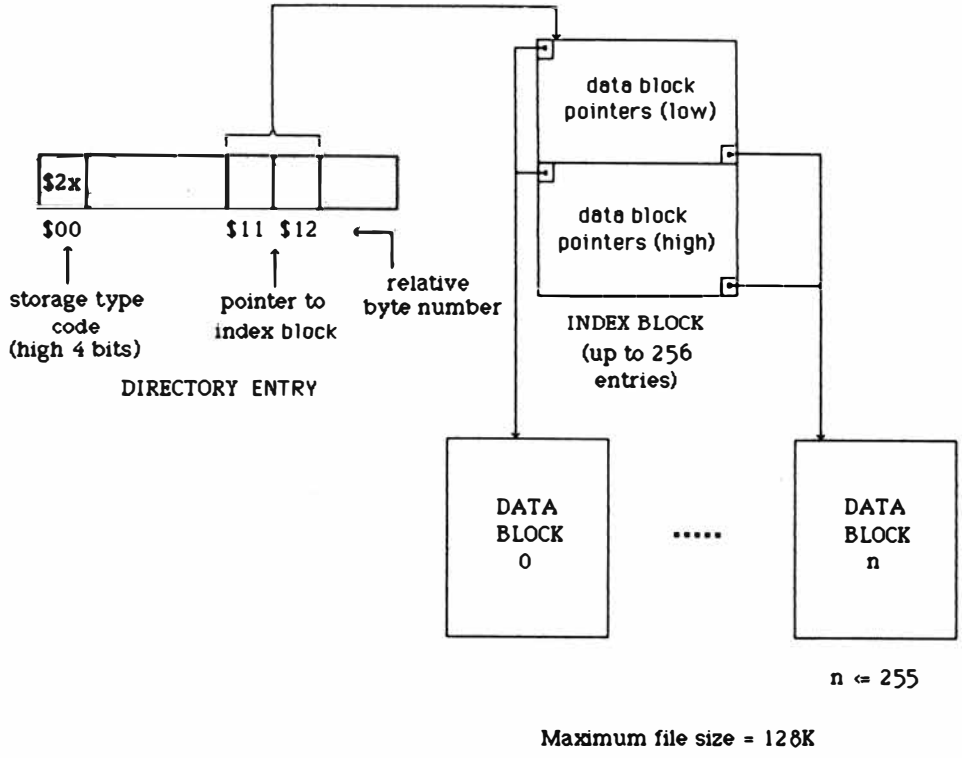


Figure 2-10. The structure of a sapling file

Table 2-5. Map of the ProDOS index block for a sapling file.

Byte Number	*Meaning*
$000	Block number of 0th data block (low)
$001	Block number of 1st data block (low)
$002	Block number of 2nd data block (low)
,	,
,	,
,	,
$0FF	Block number of 255th data block (low)
$100	Block number of 0th data block (high)
$101	Block number of 1st data block (high)
$102	Block number of 2nd data block (high)
,	,
,	,
,	,
$1FF	Block number of 255th data block (high)

Tree File. If the file is a tree file, then the key pointer holds the block number of a master index block which contains an ordered list of the block numbers of up to 128 standard sapling file index blocks. The structure of a master index block is shown in Table 2-6. Just as for sapling files, each of the index blocks pointed to by the master index block contains an ordered list of block numbers on the disk that the file uses to store its data. The maximum size of a tree file is 16 megabytes (less one byte, which is reserved for an end-of- file marker).

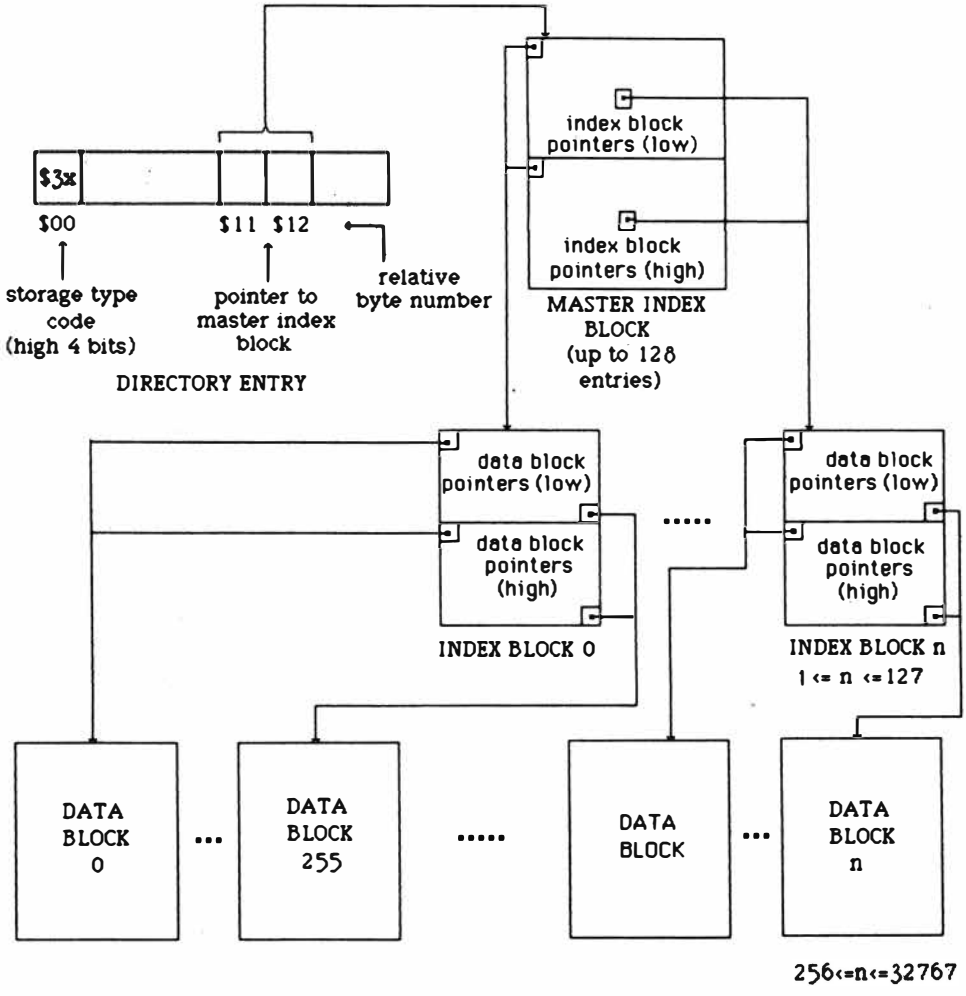


Figure 2-11. The structure of a tree file

Table 2-6. Map of the ProDOS master index block for a tree file.

Byte Number	*Meaning*
$000	Block number of 0th index block (low)
$001	Block number of 1st index block (low)
$002	Block number of 2nd index block (low)
,	,
,	,
,	,
$07F	Block number of 127th index block (low)
$100	Block number of 0th index block (high)
$101	Block number of 1st index block (high)
$102	Block number of 2nd index block (high)
,	,
,	,
,	,
$17F	Block number of 127th data block (high)

ProDOS determines the storage type of an existing file by examining the four highest bits of relative byte $00 in the directory entry for the file; the number stored here is $1 for a seedling file, $2 for a sapling file, and $3 for a tree file.

ProDOS takes care of all conversions that might become necessary if a file changes its type because it has either grown or shrunk. All this happens invisibly to an applications program and it is not necessary to know what type of file is being dealt with unless special programs are being used that do not use the standard ProDOS commands to access files.

ProDOS uses these three different indexing structures to reduce the amount of space needed to manage a file to the absolute minimum. This permits ProDOS to access a file as quickly as possible and frees up valuable disk space for the storage of other files.

Sparse Files

As we saw earlier in the discussion of TXT files, it is possible to create and use ProDOS files that are not sequential. That is, information can be written to *any* position within a file, even if that position is far away from any other previously used part of the file. To save valuable disk space, ProDOS does not actually allocate disk space for any totally unused blocks of the file that may appear in gaps such as this. Instead, it inserts $0000 placeholders in the index block to indicate that the part of the file to which the index entry corresponds has not yet been used. An actual block number will be stored at this entry at the time when that part of the file is actually written to.

Such a file is called a *sparse* file because it is really not fully populated even though it appears to be much larger file than it is.

Let's look at an actual example of a sparse file. Suppose you have created a random-access textfile with a record length of 128 bytes and you have written to record 2 and record 64 only. The structure of such a file is shown in Figure 2-12. Record 2 will be stored beginning at position $100 (2x128) in the file; this corresponds to position $100 of the first block allocated to the file (index block entry 0). Record 64 will begin at position $2000 (128x64) in the file; this corresponds to position $000 of the 16th index block entry. There are fifteen unused blocks between these two records that will be stored as $0000 entries in the index block. Thus, even though the file is logically seventeen blocks long, only three data blocks are needed to store it on the disk (one for the index block and two for the data blocks).

The file created by this program . . .

```
10 F$ = "RANSOM"
30 PRINT CHR$ (4);"OPEN";F$;",L128"
40 PRINT CHR$ (4);"WRITE";F$;",R2"
50 PRINT "RECORD 2"
60 PRINT CHR$ (4);"WRITE";F$;",R64"
70 PRINT "RECORD 64"
80 PRINT CHR$ (4);"CLOSE"
```

Is stored as follows . . .

```
Index Block (stored in the file's key pointer
entry):
0000: 8C 00 00 00 00 00 00 00 (This indicates
0008: 00 00 00 00 00 00 00 00 that data blocks
0010: 8E 00 00 00 00 00 00 00 0 and 16 are
  ,                           stored at blocks
  ,                           $008C and$008E on
0100: 00 00 00 00 00 00 00 00 this disk.)
0108: 00 00 00 00 00 00 00 00
0110: 00 00 00 00 00 00 00 00
  ,
  ,
01F8: 00 00 00 00 00 00 00 00
Data Block 0 (disk block $008C):
0000: 00 00 00 00 00 00 00 00
  ,
  ,
0100: 52 45 43 4F 52 44 20 32 RECORD 2
0108: 0D                    [carriage return]
0109: 00 00 00 00 00 00 00 00
  ,
  ,
01F8: 00 00 00 00 00 00 00 00
Data Block 16 (disk block $008E):
0000: 52 45 43 4F 52 44 20 36 RECORD 6
```

```
0008: 34                                        4
0009: 0D                                        [carriage return]
000A: 00 00 00 00 00 00 00 00
  ,
  ,
01F8: 00 00 00 00 00 00 00 00
```

Figure 2-12. The structure of a sparse file

READ.BLOCK Program

Table 2-7 shows an extremely useful program called READ.BLOCK that can be used to examine any of the blocks of data on the medium for any ProDOS disk device, to edit the contents of a block, and to write a modified block back to the disk.

With READ.BLOCK, you can easily look at real examples of the types of blocks we have been discussing in this chapter: the volume bit map, the directory blocks, the index blocks, and even a file's data blocks. You should be careful when writing a block to the disk, however, as it is easy to render the disk unreadable accidentally; you should always experiment on a back- up copy of the original disk.

Table 2-7. READ.BLOCK, a program to read any block on a ProDOS disk.

```
0     REM "READ.BLOCK"
90    HM = PEEK (115) + 256 * PEEK (116)
100   FOR I = HM TO HM + 124: READ X: POKE I,X:
      NEXT
105   POKE HM + 5, PEEK (116)
110   DEF FN MD(X) = X - 16 * INT (X / 16)
120   DEF FN M2(X) = X - 256 * INT (X / 256)
130   D$ = CHR$ (4)
150   TEXT : PRINT CHR$ (21): HOME : PRINT TAB(
      16);: INVERSE : PRINT "READ BLOCK": NORMAL :
      PRINT TAB( 10);"COPR. 1985 GARY LITTLE"
155   VTAB 8: CALL - 958: INPUT "ENTER SLOT (1-7):
      ";A$:SL = VAL (A$): IF SL < 1 OR SL > 7 THEN
      155
156   VTAB 9: CALL - 958: INPUT "ENTER DRIVE (1-
      2): ";A$:DR = VAL (A$): IF DR < 1 OR DR > 2
      THEN 156
157   POKE HM + 11,16 * SL + 128 * (DR = 2)
160   VTAB 10: CALL - 958: INPUT "ENTER BASE BLOCK
      NUMBER: ";T$: IF T$ = "" THEN 160
170   BL = INT ( VAL (T$)): IF BL = 0 AND T$ < > "
      0" THEN 160
```

Table 2-7. READ.BLOCK, a program to read any block on a ProDOS disk (continued).

```
180   IF BL < 0 THEN 160
190   RW = 128
200   POKE HM + 14, FN M2(BL): REM BLOCK # (LOW)
210   POKE HM + 15, INT (BL / 256): REM BLOCK#
      (HIGH)
220   POKE HM + 3,RW: REM READ=128 / WRITE=129
230   CALL HM
240   IF PEEK (8) < > 0 THEN PRINT : INVERSE :
      PRINT "DISK I/O ERROR": NORMAL : PRINT "
      PRESS ANY KEY TO CONTINUE: ";: GET A$: PRINT
      A$: GOTO 150
1000  VTAB 4: CALL - 958: PRINT TAB( 11);"CONTENTS
      OF BLOCK";BL: PRINT : POKE 34,5
1010  Q = 1
1020  HOME : GOSUB 2000: CALL HM+ 26:Q = Q + 1: IF
      Q = 5 THEN 1050
1030  IF PR = 0 THEN GET A$: IF A$ = CHR$ (27)
      THEN 1050
1040  GOTO 1020
1050  Q = Q - 1:PR = 0: PRINT D$;"PR#0":B = 0
1060  HTAB 1: VTAB 23: CALL - 958: PRINT "ENTER
      COMMAND (B,C,D,E,N,P,Q,W,HELP): ";: GET A$:
      IF A$ = CHR$ (13) THEN A$ = " "
1070  PRINT A$
1080  IF A$ < > "D" THEN 1110
1090  Q = Q - 1: IF Q = 0 THEN Q = 4
1100  HOME : GOSUB 2000: CALL HM + 26: GOTO 1060

1110  IF A$ = "H" THEN 5000
1120  IF A$ = "Q" THEN 1260
1130  IF A$ = "E" THEN 1270
1140  IF A$ = "P" THEN 1220
1150  IF A$ = "N" THEN 1240
1160  IF A$ = "B" THEN 150
1170  IF A$ = "C" THEN VTAB 23:CALL - 958: PRINT
      TAB( 6);: INVERSE : PRINT "TURN ON PRINTER
      IN SLOT #1": NORMAL :PR = 1: PRINT D$;"PR#1"
      : PRINT : GOTO 1000
1180  IF A$ < > "W" THEN 1210
1190  POKE HM + 15, INT (BL / 256): POKE HM +
      14,BL: POKE HM+ 3,129: VTAB 23: CALL - 958:
      PRINT "PRESS 'Y' TO VERIFY WRITE: ";: GET
      A$: IF A$ = CHR$ (13) THEN A$ = " "
1200  PRINT A$: IF A$ = "Y" THEN CALL HM:RW = 128:
      VTAB 23: CALL - 958: PRINT "WRITE COMPLETED.
      PRESS ANY KEY: ";: GET A$: GOTO 1060
1210  GOTO 5000
1220  BL = BL - 1: IF BL < 0 THEN BL = 0
```

(continued)

Table 2-7. READ.BLOCK, a program to read any block on a ProDOS disk (continued).

```
1230 GOTO 190
1240 BL = BL + 1: GOTO 190
1260 TEXT : HOME : END
1270 V = 8:H = 3: VTAB 5: PRINT TAB( 6);: INVERSE
     : PRINT "I=UP M=DOWN J=LEFT K=RIGHT":NORMAL
1280 HTAB 1: VTAB 23: CALL - 958: PRINT TAB( 6);"
     PRESS ";: INVERSE : PRINT "ESC";: NORMAL :
     PRINT " TO LEAVE EDITOR"
1290 REM
1300 GOSUB 1500: GET A$
1310 LC = 16384 + 128 * (Q - 1) + 8 * V + H:Y =
     PEEK (LC):X = ASC (A$)
1320 IF A$ = CHR$ (27) THEN HTAB 1: VTAB 5: CALL
     - 868: GOTO 1060
1330 IF A$ < > "I" THEN 1370
1340 B = 0:V = V - 1: IF V > = 0 THEN 1300
1350 V = 15:Q = Q - 1: IF Q < 1 THEN Q = 4
1360 GOSUB 2000: HOME : CALL HM+ 26: GOTO 1300
1370 IF A$ = "J" THEN B = 0:H = H - 1: IF H = - 1
     THEN H = 7
1380 IF A$ = "K" THEN B = 0:H = H + 1: IF H = 8
     THEN H = 0
1390 IF A$ < > "M" THEN 1430
1400 B = 0:V = V + 1: IF V < 16 THEN 1300
1410 V = 0:Q = Q + 1: IF Q = 5 THEN Q = 1
1420 GOTO 1360
1430 IF B = 0 THEN Y = FN MD(Y) + 16 * (X - 48) *
     (X < = 57) + 16 * (X - 55) * (X > = 65)
1440 IF B = 1 THEN Y = 16 * INT (Y / 16) + (X -
     48) * (X < = 57) + (X - 55) * (X > = 65)
1450 X = ASC (A$): IF (X > = 48 AND X < = 57) OR
     (X > = 65 AND X < = 70) THEN PRINT A$;: POKE
     ( PEEK (40) + 256 * PEEK (41) + 31 + H),Y:
     POKE LC,Y: IF B = 0 THEN CALL 64500:B = 1
1460 IF X = 8 AND B = 1 THEN B = 0
1470 IF X = 21 AND B = 0 THEN B = 1
1480 GOTO 1300
1490 CALL - 167
1500 VTAB V + 6: HTAB 3 * H + 7 + B: RETURN
2000 IF Q = 1 THEN POKE HM + 27,0: POKE HM +
     31,64
2010 IF Q = 2 THEN POKE HM + 27,128: POKE HM +
     31,64
2020 IF Q = 3 THEN POKE HM + 27,0: POKE HM +
     31,65
2030 IF Q = 4 THEN POKE HM + 27,128: POKE HM +
     31,65
2040 RETURN
```

Table 2-7. READ.BLOCK, a program to read any block on a ProDOS disk (continued).

```
5000 HOME : PRINT TAB( 10);"SUMMARY OF COMMANDS":
     PRINT TAB( 10);"==================": PRINT
5010 PRINT "B -- RESET BASE BLOCK"
5020 PRINT "C -- COPY BLOCK CONTENTS TO PRINTER"
5030 PRINT "D -- DISPLAY PREVIOUS 1/4 BLOCK"
5040 PRINT "E -- EDIT THE CURRENT BLOCK"
5050 PRINT "N -- READ THE NEXT BLOCK"
5060 PRINT "P -- READ THE PREVIOUS BLOCK"
5070 PRINT "Q -- QUIT THE PROGRAM"
5080 PRINT "W -- WRITE THE BLOCK TO DISK"
5090 PRINT : PRINT "PRESS ANY KEY TO CONTINUE: "
     ;: GET A$: PRINT A$: GOTO 1100
8000 DATA 32,0,191,128,10,3,144,
     8,176,11,3,96,0,64,0,0,169,
     0,133,8,96,169,1,133,8,96,169,0,133,6
8010 DATA 169,64,133,7,162,0,160,
     0,56,165,7,233,64,32,218,2
     53,165,6,32,218,253,169,186,
     32,237,253,169,160,32,237
8020 DATA 253,177,6,32,218,253,
     169,160,32,237,253,200,192,8,
     208,241,169,160,32,237,253,
     160,0,177,6,9,128,201,160,176
8030 DATA 2,169,174,32,237,253,
     200,192,8,208,238,169,141,32,
     237,253,24,165,6,105,8,133,
     6,165,7,105,0,133,7,232
8040 DATA 224,16,208,168,96
```

When READ.BLOCK is first run, you will be asked to enter the slot and drive numbers for the disk drive you want to access (this will be slot 3, drive 2 for the /RAM volume), and a base block number. Then the block will be read into memory and displayed on the screen in a special format. Because of 40-column screen size limitations, only one-quarter of the block can be represented at once (you have to press the "D" key to display the other quarters).

The contents of a block are displayed in 64 rows, each of which contains an offset address from the beginning of the block followed by the hexadecimal representations of the eight bytes stored from that location onward. At the far right of each row are the ASCII representations of each of these eight bytes. Note that only 16 rows are displayed at any one time.

After the entire block has been displayed, you will be asked to enter one of nine commands:

- B reset the base block number
- C copy the contents of the block to the printer (The printer must be in slot 1.)

D display the next quarter of the current block
E edit the current block
N read and display the next block on the disk
P read and display the previous block on the disk
Q quit the program
W write the block back to the disk

The functions that most of these commands perform are obvious. The only "tricky" one is the "E" (Edit) command. When the Edit command is entered, the cursor will move into the middle of the 8x16 array of hexadecimal digits that represent the contents of one-quarter of the block. To change any of these digits, use the I, J, K, and M keys to move the cursor up, left, right, and down, respectively, and then enter the new two-digit hexadecimal entry for that position. You can leave editing mode at any time by pressing the ESC key. Once you have left editing mode, you can save the changes to the disk using the "W" (Write) command.

Chapter 3

LOADING AND INSTALLING ProDOS

As far as 8-bit microcomputer operating systems go, ProDOS is relatively large. ProDOS proper occupies a total of 13K bytes of memory and that's just for the disk I/O subroutines and low- level command interpreter—it does not include the 1K buffer space that is required for each open file. If you want to add the special Applesoft disk commands, you must also install a system program called BASIC.SYSTEM, which occupies another 10.25K of space. (See Chapter 5 for a discussion of system programs.) Fortunately, all but 256 bytes of ProDOS is loaded into an area of memory (called bank-switched RAM) that is normally not used by applications programs.

In this chapter we will describe the booting process that loads ProDOS into memory in the first place, where ProDOS resides in memory, and how ProDOS makes use of certain areas in the first three pages of memory. We will also describe in detail the ProDOS global page, a 256-byte area of memory that resides from $BF00-$BFFF. A good understanding of the global page is absolutely necessary if you want to write a program that can communicate properly with ProDOS and if you want to be able to install your own device drivers for disks or clocks.

The ProDOS Booting Process

The first two blocks (0 and 1) of every standard ProDOS-formatted disk contain an assembly language program, called the *boot record*, that is automat-

ically loaded into memory at $800 and executed (by a call to $801) when the disk is booted. (The boot record is placed on the disk by the program that formatted the disk.) The program in ROM on the disk controller card performs this first part of the booting process.

The boot record program is capable of loading ProDOS on an Apple II or SOS (Sophisticated Operating System) on an Apple ///. In the following discussion we will focus on the how it reacts on an Apple II system only.

When the boot record program starts executing, it first loads the volume directory blocks (it assumes that these are located at their standard positions: blocks 2–5) into the 2K memory area from $C00–$13FF. It then scans the directory entries looking for a system program called PRODOS. If the file is not found, the message

```
*** UNABLE TO LOAD PRODOS ***
```

will be displayed, the system will halt, and you will have to boot another disk instead.

If the PRODOS program file is found, however, then the boot record will load it into memory beginning at location $2000 and run it by executing a "JMP $2000" instruction.

The PRODOS program file contains a copy of the code for the ProDOS operating system itself, as well as the code necessary to initialize various system parameters (the number of disk drives, the amount of memory, whether a clock is installed, and so on) stored in a special data area called the *ProDOS global page*. When PRODOS gets control, one of the first things it does is to relocate the ProDOS image to its execution position in bank- switched RAM. (We'll describe this RAM area in detail in the next section.)

The last thing PRODOS does is to scan the volume directory for the first entry that is a system file having a name of the form "xxxxxxxx.SYSTEM." (The file is usually a language interpreter that allows you to write other programs, but it could be any other executable program.) If one isn't found, the message

```
** UNABLE TO FIND A ".SYSTEM" FILE **
```

will be displayed and the system will have to be restarted. If such a file is found, it will be loaded into memory beginning at $2000 and executed.

If an Applesoft programming environment is to be used, this system file must be BASIC.SYSTEM (it is found on the ProDOS master diskette). As we will see in Chapter 5, BASIC.SYSTEM contains the subroutines that add the disk commands to the standard Applesoft programming language. It also takes care of parsing these commands, checking syntax, and calling the low-level ProDOS I/O subroutines when required.

It should be clear from this discussion that ProDOS is really nothing without a system program like BASIC.SYSTEM to act as a software interface between the user and the low-level ProDOS operating system. It just won't operate without such a program. For this reason, the ProDOS-BASIC.SYSTEM environment is commonly referred to as "ProDOS" even though this is technically not

the case. In the remainder of this chapter we will be examining ProDOS proper; a detailed discussion of BASIC.SYSTEM (and system programs in general) will be deferred to Chapter 5.

ProDOS Memory Usage

Bank-Switched RAM

After ProDOS has been loaded as described, it will occupy the following memory locations (as shown in Figure 3-1):

- $E000-$FFFF in main bank-switched RAM
- $D000-$DFFF in $Dx bank1 of main bank-switched RAM
- $D100-$D3FF in $Dx bank2 of main bank-switched RAM (This is the *dispatcher* code.)
- $BF00-$BFFF in main RAM (This is the *ProDOS global page*.)

The remaining space in bank-switched RAM, from $D400–$DFFF in $Dx bank2, has been reserved for future use by ProDOS and must not be used by applications programs.

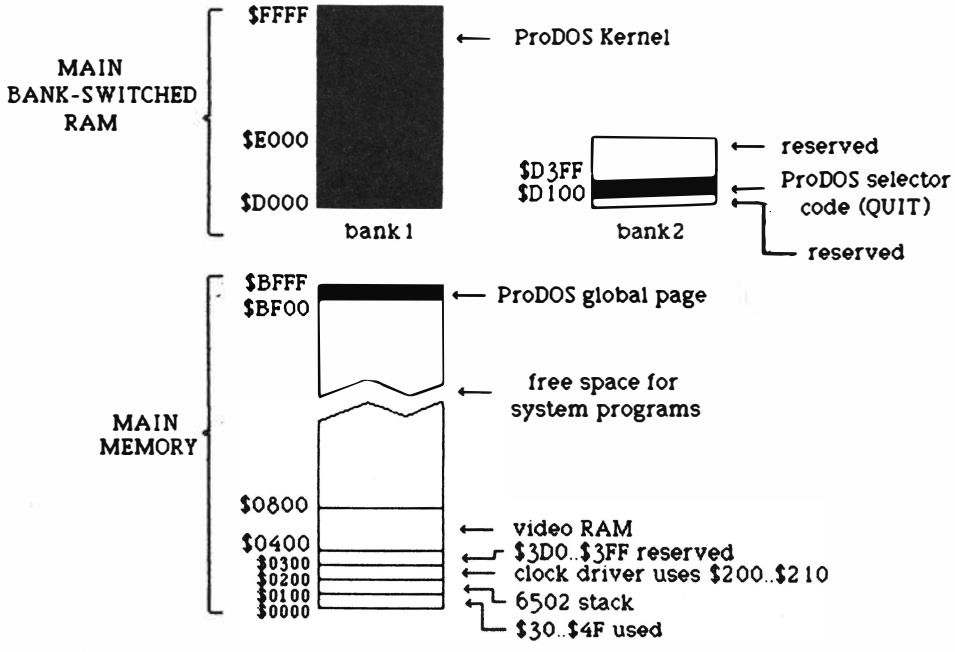


Figure 3-1. ProDOS Memory Map.

You may well be wondering what the terms "bank-switched RAM," "$Dx bank1," and "$Dx bank2" mean. An Apple II with a 16K memory card installed in slot #0 (or an Apple //e or Apple //c) has 64K of main RAM memory that is normally used by Applesoft and ProDOS. This memory, however, is not mapped to one area encompassing the entire 64K space that the 6502 microprocessor is capable of addressing. The first 48K of this memory space corresponds to the block of memory from $0000 to $BFFF but the remaining 16K of memory, the bank-switched RAM, corresponds to one 8K region of memory from $E000 to $FFFF and two 4K regions of memory from $D000 to $DFFF (called "$Dx bank1" and "$Dx bank2," respectively). The address space used by bank-switched RAM is exactly the same as that used by the Applesoft and system monitor ROM, so only one space or the other can be active for read or write operations at any one time.

As shown in Table 3-1, the Apple II uses eight I/O memory locations (*soft switches*) to control whether bank-switched RAM or the corresponding ROM space is to be active and whether $Dx bank1 or bank2 is to be used. We can even set these switches in such a way that the RAM area can be read from but not written to, or so that it will be active for write operations at the same time that the corresponding ROM area is active for read operations. This means that you can write data to the RAM area while running a program that uses subroutines in the ROM that occupies the same memory locations (that is, subroutines in Applesoft and the system monitor).

Table 3-1. Bank-switched RAM soft switches.

Address			*Active*	*Read*	*Write*
Hex	*(Dec)*	*Symbolic Name*	*$Dx Bank*	*From*	*to RAM?*
$C080	(49280)	READBSR2	2	RAM	No
$C081	(49281)	WRITEBSR2	2	ROM	Yes (*)
$C082	(49282)	OFFBSR2	2	ROM	No
$C083	(49283)	RDWRBSR2	2	RAM	Yes (*)
$C088	(49288)	READBSR1	1	RAM	No
$C089	(49289)	WRITEBSR1	1	ROM	Yes (*)
$C08A	(49290)	OFFBSR1	1	ROM	No
$C08B	(49291)	RDWRBSR1	1	RAM	Yes (*)

A location must be read from to perform the indicated function.
(*) Read twice in succession to write-enable bank-switched RAM.

To activate the particular mode of operation desired, it is necessary to select the appropriate soft switch address and then perform any kind of read operation at that address: an LDA, LDY, LDX, or BIT instruction in assembly language or a PEEK from Applesoft.

ProDOS takes care of managing the bank-switched RAM switches whenever it is called upon to perform some command. In general, it will save the state of bank-switched RAM when it gets control and then it will read- and write-enable

bank1 of bank-switched RAM before passing control to a subroutine residing there. When it relinquishes control, it will restore bank-switched RAM to its original state.

Auxiliary Memory

An Apple //e, with an extended 80-column text card installed, and an Apple //c both have a 64K *auxiliary* memory space that is mapped to addresses in the same way that the main 64K memory space is. Since this space is normally not used by applications programs, ProDOS uses it to store disk files in much the same way it stores files to a real disk drive. The name of the volume for this so-called "RAMdisk" is /RAM and we'll be investigating its characteristics in Chapter 7.

Page Zero Usage

ProDOS uses 21 locations in page zero for temporary data storage: $3A–$4E. The first 6 locations ($3A–$3F) are used only by the internal ProDOS disk device drivers for the Disk][and the /RAM volume. This means that if a disk I/O operation is performed, the existing contents of $3A–$3F will be overwritten. This is not too serious since these locations are usually used only by the Apple II's system monitor command interpreter. However, if they are used by an applications program, an irreconcilable conflict will occur and the program could bomb. Don't use them.

The other 15 locations ($40–$4E) are used by the ProDOS machine language interpreter (MLI) subroutine. Unlike the situation for the $3A–$3F area, however, when control passes to the MLI, the current contents of $40–$4E are saved in a safe data area within ProDOS and then are restored just before control returns to the caller.

Page Two Usage

One of the most useful features of ProDOS is its ability to date-stamp its files. ProDOS can do this easily because date and time fields are reserved in each directory entry and there is a special internal subroutine, called a clock driver, that ProDOS can call to read the current time and date. See Chapter 8.

The standard internal ProDOS clock driver works with the Thunderclock clock card (or equivalent) only. One of the quirks of the Thunderclock is that it uses the first part of the Apple II's line input buffer, from $200–$210, to store its time data string whenever the time is requested. This means that an applications program must not use this area for any purpose; if it does, then it will probably not work properly after the clock driver is called.

Note that other parts of page two may well be used by any system program (such as BASIC.SYSTEM) used with ProDOS. For example, BASIC.SYSTEM uses most of page two as a temporary data buffer area when it executes its disk

commands. This is another good reason to avoid using page two for program data storage.

Page Three Usage

The block of memory at the top end of page 3 of memory, from $3D0–$3FF, is used for special purposes on the Apple II. First of all, ProDOS reserves the area from $3D0–$3EC for use by any system program (such as BASIC.SYSTEM) that may be loaded. The specific use of this area is dictated by the system program itself, but it is normally used to store short, fixed-position subroutines that pass control to important subroutines in the main body of the system program. For example, BASIC.SYSTEM stores a three-byte "JMP $BE00" instruction beginning at $3D0; this is the warm entry point to BASIC.SYSTEM (that is, it reinstalls BASIC.SYSTEM without destroying the Applesoft program in memory). We'll investigate BASIC.SYSTEM's use of page 3 in more detail in Chapter 5.

The rest of page 3 is reserved for storage of a set of user-installable vectors and subroutines that are used to service interrupt conditions or special commands:

- XFER (main/auxiliary memory data transfer) vector at $3ED/$3EE (//e and //c only)
- BRK (6502 break instruction) vector at $3F0/$3F1
- RESET (reset interrupt) vector at $3F2/$3F3 and its enabling byte at $3F4 (called the *powered-up byte*)
- & (Applesoft ampersand command) vector at $3F5-$3F7
- [control-Y] (system monitor USER command) vector at $3F8-$3FA
- NMI (non-maskable interrupt) vector at $3FB-$3FD
- IRQ (interrupt request) vector at $3FE/$3FF

(Refer to Appendix IV of *Inside the Apple //e* for a detailed discussion of the meaning of each of these vectors and subroutines.)

ProDOS initializes the IRQ vector at $3FE/$3FF by storing the address of its internal interrupt-handling subroutine there. The vectors for RESET, &, [control-Y], and NMI are set equal to $FF59, the cold start entry point to the system monitor. Note, however, that BASIC.SYSTEM stores other values in these vectors (except BRK and IRQ) when it is first loaded. See Chapter 5 for a description of how these vectors are initialized by BASIC.SYSTEM.

The ProDOS Global Page: $BF00–$BFFF

The page of memory from $BF00 to $BFFF is called the *ProDOS global page* and it acts as the gateway to ProDOS proper (that is, the part that resides in bank-switched RAM). It contains several fixed-position jump vectors to standard

ProDOS subroutines (the machine language interpreter, the clock driver, the error handler, and so on) and several important data areas that contain information defining the state of the system. These data areas may be inspected, or changed, to facilitate communication between a system program (like BASIC.SYSTEM) and ProDOS.

The global page also contains the bank-switching subroutines needed to transfer control to and from the parts of the ProDOS machine language interpreter and interrupt handler that reside in bank-switched RAM. It should never be necessary for you to use these subroutines directly so their addresses, and the code itself, are not guaranteed to stay the same from one ProDOS version to another.

The System Bit Map

One important area in the ProDOS global page is the system bit map; it occupies the area from $BF58 to $BF6F. This map is used to indicate which RAM areas have been reserved for use and which are free. Before ProDOS (or BASIC.SYSTEM) performs any loading or buffer allocation operations, this map is examined to see if there will be a conflict with reserved areas. If there will be, the command is not executed and an error condition will be flagged.

Each bit in the map corresponds to one of the 192 pages of memory in the Apple II's main RAM area (pages $00 through $BF). If a bit is set to 1, the corresponding page has been reserved. The relative byte number (counting from zero) within the system bit map in which the bit for a given page number resides is the whole number generated by dividing the page number by 8; the bit number within this byte is seven minus the remainder generated by the division. For example, the bit for page 190 ($BE) is bit 1 of relative byte 23: 190 divided by 8 is 23 (the relative byte number) and the remainder is 6 (meaning that the bit number is 7-1 = 6).

ProDOS initially marks page zero, the stack page (1), the video RAM area (pages 4–7) and its global page (page $BF) as being reserved. Other pages can be protected as desired by system and applications programs. For example, BASIC.SYSTEM also reserves pages $9A-$B9 and page $BE; these are the pages where the actual BASIC.SYSTEM code is stored.

Short utility programs are often stored in the lower part of page 3 ($300–$3CF) because that area is not otherwise used by Applesoft, ProDOS, or the system monitor. Such a program can prevent itself from being overwritten by setting the appropriate bit in the system bit map to 1 (this will be bit 4 of $BF58).

The Machine Identification Byte

There is a byte in the ProDOS global page called MACHID ($BF98) that can be examined to determine the nature of the hardware environment in which ProDOS is executing. It contains information on the type of Apple being used (II, II Plus, //e, or //c), the amount of RAM memory (48K, 64K, or 128K), and whether an 80-column card or clock card is installed.

The bits in MACHID have the following meanings:

bits 7,6 (if bit 3 = 0)	00 = Apple II
	01 = Apple II Plus
	10 = Apple //e
	11 = Apple /// in Apple II emulation mode

bits 7,6 (if bit 3 = 1)	00 = [reserved]
	01 = [reserved]
	10 = Apple //c
	11 = [reserved]

bits 5,4	00 = [reserved] 01 = 48K RAM 10 = 64K system 11 = 128K system (//e, //c only)
bit 3	determines how bits 7,6 are to be interpreted
bit 2	[reserved]
bit 1	1 = 80-column card is installed 0 = no 80-column card is installed
bit 0	1 = clock card is installed 0 = no clock card is installed

Note that it is also possible to determine what type of Apple system is being used by examining two identification bytes in the system monitor ROM. For example, only an Apple //e or //c has a 6 stored at $FBB3. Location $FBC0 can be read to distinguish between a //e and a //c: this location holds a $00 if a //c is being used or a $EA if a //e with standard ROMs is being used; if the Apple //e upgrade ROMs (the *icon ROMs*) are being used, this location will hold $E0.

Source Listing of ProDOS Global Page

All of the other important areas in the ProDOS global page will be analyzed in detail in later chapters. Until then, a commented source listing of the code for the global page describing the usage of each of its 256 bytes is shown in Table 3-2.

Table 3-2. Source listing for ProDOS system global page.

```
                    1   ******************************
                    2   * ProDOS System Global Page *
                    3   *       $BF00..$BFFF       *
                    4   * for ProDOS version 1.1.1  *
                    5   ******************************
                    6
                    7   * NOTE: The addresses of the following subroutines
                    8   *       may change in future versions of ProDOS.
                    9
                    10  CLOCKDR  EQU  $D742    ;ProDOS clock driver
                    11  NODEVICE EQU  $DEA2    ;NO DEVICE CONNECTED addr
                    12  SYSERR1  EQU  $DFE5    ;System error handler
                    13  SYDEATH1 EQU  $DFF1    ;Critical error handler
                    14  ENTRYMLI EQU  $DE00    ;ProDOS MLI handler
                    15  IRQRECEV EQU  $DF3A    ;ProDOS interrupt handler
                    16  MLIQUIT  EQU  $FCE5    ;QUIT subroutine
                    17
                    18           ORG  $BF00
                    19
                    20  * MLI is the primary entry point to the ProDOS
                    21  * machine language interpeter. This interpreter
                    22  * supports a number of commands that can be used
                    23  * to access files. See Chapter 4.
                    24
BF00: 4C B7 BF      25  MLI      JMP  MLIENT1  ;The gateway to MLI commands
                    26
                    27  * QUIT is called whenever the MLI QUIT command is
                    28  * requested. (This is normally done when transferring
                    29  * control from one system program to another.)
                    30  * The standard subroutine asks the user to
```

(continued)

Table 3-2. Source listing for ProDOS system global page (continued).

```
                        31  * enter a new prefix and system file name and
                        32  * then executes the program specified. See Chapter 4.
                        33
BF03: 4C E5 FC          34  QUIT     JMP  MLIQUIT  ;Execute QUIT command
                        35
                        36  * DATETIME is called whenever the MLI GETTIME
                        37  * command is executed. If a clock card is installed,
                        38  * it will read the clock and place the date and
                        39  * time in DATE ($BF90) and TIME ($BF92). For
                        40  * details on how this is done, see Chapter 8.
                        41
BF06: 60                42  DATETIME RTS           ;JMP ($4C) if clock installed
BF07: 42 D7             43           DA   CLOCKDR  ;Address of clock driver
                        44
                        45  * ProDOS calls SYSERR if an error occurs during
                        46  * an MLI call. SYSERR takes the error code (that
                        47  * is in the accumulator) and stores it in SERR.
                        48  *
                        49  * SYSDEATH is called whenever a critical error
                        50  * occurs (example: important ProDOS data areas
                        51  * are overwritten). The system will have to be
                        52  * restarted if a critical error occurs.
                        53  *
                        54  * See Chapter 4 for a discussion of MLI system
                        55  * errors and critical errors.
                        56
BF09: 4C E5 DF          57  SYSERR   JMP  SYSERR1  ;System error handler
BF0C: 4C F1 DF          58  SYSDEATH JMP  SYDEATH1 ;Critical error handler
BF0F: 00                59  SERR     DFB  $00      ;MLI error code (0 if no error)
                        60
                        61  * Disk driver vector table. Each entry in the table
```

```
                        62  * corresponds to a unique drive/slot combination
                        63  * as shown. If an entry is unused, its vector
                        64  * points to the ProDOS ''no device connected''
                        65  * subroutine. The /RAM device that is available
                        66  * on an Apple //e or //c will be mapped to the
                        67  * slot 3/drive 2 device. The entries in the
                        68  * following table are for a 128K Apple //e
                        69  * with a Disk ][ controller card in slot 6.
                        70
BF10: A2 DE             71  DEVADR01 DA   NODEVICE ;"No device connected" vector
BF12: A2 DE             72  DEVADR11 DA   NODEVICE ;Slot 1/Drive 1 vector
BF14: A2 DE             73  DEVADR21 DA   NODEVICE ;Slot 2/Drive 1 vector
BF16: A2 DE             74  DEVADR31 DA   NODEVICE ;Slot 3/Drive 1 vector
BF18: A2 DE             75  DEVADR41 DA   NODEVICE ;Slot 4/Drive 1 vector
BF1A: A2 DE             76  DEVADR51 DA   NODEVICE ;Slot 5/Drive 1 vector
BF1C: 00 D0             77  DEVADR61 DA   $D000    ;Slot 6/Drive 1 vector (Disk ][)
BF1E: A2 DE             78  DEVADR71 DA   NODEVICE ;Slot 7/Drive 1 vector
                        79
BF20: A2 DE             80  DEVADR02 DA   NODEVICE ;"No device connected" vector
BF22: A2 DE             81  DEVADR12 DA   NODEVICE ;Slot 1/Drive 2 vector
BF24: A2 DE             82  DEVADR22 DA   NODEVICE ;Slot 2/Drive 2 vector
BF26: 00 FF             83  DEVADR32 DA   $FF00    ;Slot 3/Drive 2 vector (/RAM)
BF28: A2 DE             84  DEVADR42 DA   NODEVICE ;Slot 4/Drive 2 vector
BF2A: A2 DE             85  DEVADR52 DA   NODEVICE ;Slot 5/Drive 2 vector
BF2C: 00 D0             86  DEVADR62 DA   $D000    ;Slot 6/Drive 2 vector (Disk ][)
BF2E: A2 DE             87  DEVADR72 DA   NODEVICE ;Slot 7/Drive 2 vector
                        88
                        89  * DEVNUM contains the slot/drive code for the
                        90  * last disk device that was accessed. The bit
                        91  * format for this code is as follows:
                        92  *
                        93  *     D S S S 0 0 0 0
```

(continued)

Table 3-2. Source listing for ProDOS system global page (continued).

```
                        94  *
                        95  * where D is the drive number (0 for drive 1
                        96  * and 1 for drive 2) and SSS is the slot number
                        97  * (from 1 to 7).
                        98
BF30: 60                99  DEVNUM   DFB  $60      ;Slot/drive of last access
                       100
                       101 * DEVCNT holds the number of active disk devices
                       102 * installed in the system, less 1.
                       103
BF31: 02               104 DEVCNT   DFB  $02
                       105
                       106 * DEVLST contains a list of the drive/slot
                       107 * codes for each of the active disk devices
                       108 * (14 maximum).
                       109 * The codes are in the same format as used
                       110 * for DEVNUM except that the low-order four
                       111 * bits contain device ID information:
                       112 *
                       113 *          $x0 = Disk ][
                       114 *          $x4 = ProFile
                       115 *          $xF = /RAM
                       116 *
                       117 * Other ID values may be defined by ''foreign''
                       118 * disk devices (see Chapter 7).
                       119
BF32: E0               120 DEVLST   DFB  $E0      ;Disk ][ in slot 6/drive 2
BF33: 60               121          DFB  $60      ;Disk ][ in slot 6/drive 1
BF34: BF               122          DFB  $BF      ;/RAM in slot 3/drive 2
BF35: 00               123          DFB  $00
BF36: 00               124          DFB  $00
```

```
BF37: 00          125           DFB  $00
BF38: 00          126           DFB  $00
BF39: 00          127           DFB  $00
BF3A: 00          128           DFB  $00
BF3B: 00          129           DFB  $00
BF3C: 00          130           DFB  $00
BF3D: 00          131           DFB  $00
BF3E: 00          132           DFB  $00
BF3F: 00          133           DFB  $00
                  134
BF40: 43 4F 50    135           ASC  'COPR. APPLE,1983'
                  136
                  137 * This tiny bit of code is used by the ProDOS
                  138 * interrupt-handling subroutine.
                  139
BF50: 8D 8B C0    140           STA  $C08B    ;Turn on RAMcard
BF53: 4C D8 FF    141           JMP  $FFD8
BF56: 00          142 SAVE45    DFB  $00      ;Contents of $45 upon interrupt
BF57: 00          143 SAVEDX    DFB  $00      ;ID code for $Dx bank
                  144
                  145 * The system bit map. Each bit in this 24-byte
                  146 * (192-bit) table corresponds to a unique page
                  147 * from $00 through $BF. Page $00 corresponds
                  148 * to bit 7 of the first byte and page $BF
                  149 * corresponds to bit 0 of the last byte.
                  150 * If the page is in use, the corresponding
                  151 * bit will be set to 1. The configuration of
                  152 * the bit map after BASIC.SYSTEM has been
                  153 * loaded is shown.
                  154
BF58: CF 00 00    155 BITMAP    DFB  $CF,$00,$00 ;Pages 0,1,4-7 in use
BF5B: 00 00 00    156           DFB  $00,$00,$00
```

(continued)

Table 3-2. Source listing for ProDOS system global page (continued).

```
BF5E: 00 00 00 157           DFB   $00,$00,$00
BF61: 00 00 00 158           DFB   $00,$00,$00
BF64: 00 00 00 159           DFB   $00,$00,$00
BF67: 00 00 00 160           DFB   $00,$00,$00
BF6A: 00 3F FF 161           DFB   $00,$3F,$FF ;Pages $9A-$A7 in use
BF6D: FF FF C3 162           DFB   $FF,$FF,$C3 ;Pages $A8-$B9,$BE,$BF in use
               163
               164 * File buffer table. The buffer addresses for each
               165 * open file (a maximum of 8 are allowed) are stored
               166 * in this table. A buffer address must be changed
               167 * by using the MLI SET_BUF command.
               168
BF70: 00 00    169 BUFFER1  DA    $0000     ;Buffer address for file 1
BF72: 00 00    170 BUFFER2  DA    $0000     ;Buffer address for file 2
BF74: 00 00    171 BUFFER3  DA    $0000     ;Buffer address for file 3
BF76: 00 00    172 BUFFER4  DA    $0000     ;Buffer address for file 4
BF78: 00 00    173 BUFFER5  DA    $0000     ;Buffer address for file 5
BF7A: 00 00    174 BUFFER6  DA    $0000     ;Buffer address for file 6
BF7C: 00 00    175 BUFFER7  DA    $0000     ;Buffer address for file 7
BF7E: 00 00    176 BUFFER8  DA    $0000     ;Buffer address for file 8
               177
               178 * Interrupt vector table. This is where the
               179 * addresses of the user-installed interrupt
               180 * handling subroutines are stored (4 maximum).
               181 * They are installed using the MLI ALLOC_INTERRUPT
               182 * command and removed using the DEALLOC_INTERRUPT
               183 * command. Following the vector table is the
               184 * data area that ProDOS uses to store registers
               185 * and bank switching information when an
               186 * interrupt occurs. See Chapter 6.
               187
```

```
BF80: 00 00     188 INTRUPT1 DA   $0000     ;Interrupt vector #1
BF82: 00 00     189 INTRUPT2 DA   $0000     ;Interrupt vector #2
BF84: 00 00     190 INTRUPT3 DA   $0000     ;Interrupt vector #3
BF86: 00 00     191 INTRUPT4 DA   $0000     ;Interrupt vector #4
BF88: 00        192 INTAREG  DS   1         ;Accumulator
BF89: 00        193 INTXREG  DS   1         ;X register
BF8A: 00        194 INTYREG  DS   1         ;Y register
BF8B: 00        195 INTSREG  DS   1         ;Stack pointer register
BF8C: 00        196 INTPREG  DS   1         ;Processor status register
BF8D: 00        197 INTBNKID DS   1         ;ID code for $Dx bank
BF8E: 00 00     198 INTADDR  DA   $0000     ;Address where IRQ occurred
                199
                200 * The system date and time are stored in the
                201 * following two words in a special packed
                202 * format:
                203 *
                204 *     DATE:    year    = bits 15-9 (0..99)
                205 *              month   = bits 8-5 (1..12)
                206 *              day     = bits 4-0 (1..31)
                207 *
                208 *     TIME:    hours   = bits 12-8 (0..23)
                209 *              minutes = bits 5-0 (0..59)
                210 *
                211 * See Chapter 8 for more on DATE and TIME.
                212
BF90: 00 00     213 DATE     DW   $0000
BF92: 00 00     214 TIME     DW   $0000
                215
                216 * LEVEL indicates the level of the files to
                217 * be acted on by the ProDOS OPEN, FLUSH, and
                218 * CLOSE commands.
                219
```

(continued)

Table 3-2. Source listing for ProDOS system global page (continued).

```
BF94: 00        220 LEVEL     DFB  $00       ;Current file level
BF95: 00        221 BUBIT     DFB  $00       ;Backup bit flag
                222
BF96: 00 00     223 SPARE1    DW   $0000     ;Unused/reserved
                224
                225 * MACHID identifies the type of Apple being used,
                226 * the amount of memory available, and whether
                227 * an 80-column card or clock card (Thunderclock)
                228 * is installed. Here is the meaning of the bits
                229 * in MACHID:
                230 *
                231 *      bits 7,6 (if bit 3 = 0):  00 = Apple II
                232 *                                01 = Apple II Plus
                233 *                                10 = Apple //e
                234 *                                11 = Apple /// emul.
                235 *
                236 *      bits 7,6 (if bit 3 = 1):  00 = [reserved]
                237 *                                01 = [reserved]
                238 *                                10 = Apple //c
                239 *                                11 = [reserved]
                240 *
                241 *      bits 5,4:  00 = [reserved]
                242 *                 01 = 48K system
                243 *                 10 = 64K system
                244 *                 11 = 128K system (//e, //c only)
                245 *
                246 *      bit 3: determines how bits 7,6 are to be
                247 *             interpreted
                248 *
                249 *      bit 2: [reserved]
                250 *
```

```
                251 *       bit 1: 1 = 80-column card is installed
                252 *              0 = no 80-column card is installed
                253 *
                254 *       bit 0: 1 = Thunderclock clock card is installed
                255 *              0 = no Thunderclock clock card is
                    installed
                256 *
                257 * The example given is for a 128K Apple //e
                258 * with an 80-column card but no clock.
                259
BF98: B2        260 MACHID   DFB  $B2       ;//e + 80-columns + 128K
                261
                262 * The high seven bits of SLTBYT are used
                263 * as flags to indicate whether there is a
                264 * peripheral card with ROM on it in
                265 * slot (bit #). In the following example,
                266 * a byte is used that indicates ROM in
                267 * slots 1, 2, 3, 4, and 6.
                268
BF99: 5E        269 SLTBYT   DFB  $5E       ;Binary 01011110
                270
                271 * PFIXPTR is a flag that indicates whether
                272 * a filename prefix has yet been defined.
                273 * If it hasn't, then PFIXPTR will be 0 and
                274 * full pathnames (rather than partial
                275 * pathnames) must be specified when a
                276 * ProDOS command is requested.
                277
BF9A: 00        278 PFIXPTR  DFB  $00       ;Prefix flag (0 if no prefix)
                279
                280 * The following four parameters are set up
                281 * whenever an MLI call (''JSR $BF00'') is made.
```

(continued)

Table 3-2. Source listing for ProDOS system global page (continued).

```
                        282
BF9B: 00                283 MLIACTV  DFB   $00       ;MLI flag (bit 7=1 if active)
BF9C: 00 00             284 CMDADR   DA    $0000     ;Address+6 of last JSR to MLI
BF9E: 00                285 SAVEX    DFB   $00       ;X register when MLI called
BF9F: 00                286 SAVEY    DFB   $00       ;Y register when MLI called
                        287
                        288 * All calls to the MLI eventually exit by
                        289 * calling this subroutine with A = BNKBYT1.
                        290 * and bank1 of bank-switched RAM read enabled.
                        291 * EXIT restores the original state of the RAM
                        292 * switches and returns control to the address
                        293 * stored in CMDADR ($BF9C) via a ''simulated'' RTI.
                        294
BFA0: 4D 00 E0          295 EXIT     EOR   $E000     ;$E000 same as on entry?
BFA3: F0 05             296          BEQ   EXIT1     ;Yes, so RAM must be active
BFA5: 8D 82 C0          297          STA   $C082     ;No, so enable ROM
BFA8: D0 0B             298          BNE   EXIT2     ;(always taken)
BFAA: AD F5 BF          299 EXIT1    LDA   BNKBYT2   ;Get $Dx bank code
BFAD: 4D 00 D0          300          EOR   $D000     ;Same as on entry?
BFB0: F0 03             301          BEQ   EXIT2     ;Yes, so bank1 RAM active
BFB2: AD 83 C0          302          LDA   $C083     ;Read-enable bank2 RAM
BFB5: 68                303 EXIT2    PLA
BFB6: 40                304          RTI             ;(returns to CMDADR)
                        305
                        306 * The standard "JSR $BF00" MLI call is
                        307 * routed to this secondary entry point.
                        308 * It sets MLIACTV, saves the status of
                        309 * bank-switched RAM, and then enables
                        310 * bank1 of bank-switched RAM before
                        311 * passing control to ENTRYMLI.
                        312
```

```
BFB7: 38          313 MLIENT1  SEC
BFB8: 6E 9B BF    314          ROR  MLIACTV  ;Set "MLI active" flag (bit 7)
BFBB: AD 00 E0    315          LDA  $E000
BFBE: 8D F4 BF    316          STA  BNKBYT1  ;Save RAM/ROM code
BFC1: AD 00 D0    317          LDA  $D000
BFC4: 8D F5 BF    318          STA  BNKBYT2  ;Save $Dx bank code
BFC7: AD 8B C0    319          LDA  $C08B
BFCA: AD 8B C0    320          LDA  $C08B    ;Read/Write bank1 RAM
BFCD: 4C 00 DE    321          JMP  ENTRYMLI ;Go to RAM to do the rest
                  322
                  323 * This is the tail end of the special ProDOS
                  324 * interrupt handling subroutine.
                  325
BFD0: AD 8D BF    326 IRQXIT   LDA  INTBNKID ;Get RAMcard status
BFD3: F0 0D       327 IRQXIT3  BEQ  IRQXIT2  ;Branch if bank1 $Dx enabled
BFD5: 30 08       328          BMI  IRQXIT1  ;Branch if bank2 $Dx enabled
BFD7: 4A          329          LSR           ;Is there a RAM card?
BFD8: 90 0D       330          BCC  ROMXIT   ;No, so branch
BFDA: AD 81 C0    331          LDA  $C081    ;Yes, so enable ROM
BFDD: B0 08       332          BCS  ROMXIT   ;(always taken)
BFDF: AD 83 C0    333 IRQXIT1  LDA  $C083    ;Read-enable bank2 $Dx
BFE2: A9 01       334 IRQXIT2  LDA  #1
BFE4: 8D 8D BF    335          STA  INTBNKID ;Set flag for ROM
BFE7: AD 88 BF    336 ROMXIT   LDA  INTAREG  ;Restore accumulator
BFEA: 40          337          RTI           ;and finish up.
                  338
                  339 * The IRQ vector at $3FE/$3FF points here.
                  340 * This code simply read/write enables bank1
                  341 * of bank-switched RAM before passing control
                  342 * to the ProDOS interrupt handler that resides
                  343 * there.
                  344
BFEB: 2C 8B C0    345 IRQENT   BIT  $C08B    ;Read and ...
BFEE: 2C 8B C0    346          BIT  $C08B    ;... write-enable bank1 RAM
```

(continued)

Table 3-2. Source listing for ProDOS system global page (continued).

```
BFF1: 4C 3A DF 347          JMP  IRQRECEV ;Go to IRQ handler
              348
BFF4: 00      349 BNKBYT1   DFB  $00       ;RAM/ROM status stored here
BFF5: 00      350 BNKBYT2   DFB  $00       ;$Dx RAM bank status stored here
              351
              352 * The following code is called to force the
              353 * user to reboot the system. It causes the message
              354 *
              355 *          INSERT SYSTEM DISK AND RESTART
              356 *
              357 * to be displayed.
              358
BFF6: 2C 8B C0 359 RESTART  BIT  $C08B     ;Read-enable RAM
BFF9: 4C EF DF 360          JMP  $DFEF     ;System restart code
              361
              362 * IBAKVER is the earliest version number of the
              363 * ProDOS kernel (MLI) that can be used by the
              364 * currently active system program (interpreter).
              365 * IVERSION is the version number of the system
              366 * program. When a system program is first
              367 * executed it must set up these two parameters.
              368 * See Appendix II for IVERSION numbers.
              369
BFFC: 00      370 IBAKVER   DFB  $00       ;Earliest compatible kernel
BFFD: 01      371 IVERSION  DFB  $01       ;Current interpreter version
              372
              373 * KBAKVER is the earliest version number of the
              374 * ProDOS kernel (MLI) that is compatible with
              375 * the current version number stored in KVERSION.
              376 * See Appendix II for KVERSION numbers.
              377
BFFE: 00      378 KBAKVER   DFB  $00       ;Earliest compatible version
BFFF: 01      379 KVERSION  DFB  $01       ;Current ProDOS version
```

Chapter 4

THE MACHINE LANGUAGE INTERFACE

The major module contained within ProDOS is a low-level command interpreter called the machine language interface (MLI). The MLI interpreter supports a total of 26 commands, most of which are used to perform fundamental operations on disk files (creating, opening, closing, reading, writing, and so on). Each of these commands is accessed in the same way, using a standard protocol defined by the software designers at Apple Computer, Inc.

In this chapter we're going to take a close look at each of the MLI commands and show you how to take advantage of them in your own 6502 assembly language programs. In particular, we'll see how to

- call specific MLI commands
- set up MLI command parameter tables
- locate results returned by the MLI commands
- identify MLI error conditions
- interpret MLI error codes

Along the way we'll present short programming examples that will serve to clarify how the MLI commands are to be used.

Using the MLI Commands

It's really not that difficult to execute an MLI command. A typical program looks something like this:

```
          ,
          [place values in parameter table
          before calling MLI]
          ,
          JSR $BF00      ;$BF00 is the MLI entry
                          point
          DFB DOSCMD     ;MLI command number goes
                          here
          DA PARMTBL     ;Address of parameter
                          table
          BCS ERROR      ;Carry is set if error
                          occurred
          ,
          [continue on with your
          program]
          ,
          RTS

ERROR     ,
          [error handler
          goes here]
          ,
          RTS

PARMTBL   DFB NPARMS     ;NPARMS = # of parameters
                          in table
          ,
          [the rest of the parameters
          are placed here in the order
          defined by the MLI command.]
          ,
```

The key instruction here is "JSR $BF00." $BF00 is the address of the entry point to the ProDOS MLI interpreter. This interpreter is in charge of determining what MLI command has been requested and with passing control to the appropriate ProDOS subroutine to handle the request.

Figure 4-1 is a flowchart showing what occurs after a "JSR $BF00" instruction is executed. As soon as the MLI takes control, four locations in the ProDOS global page are modified: MLIACTV ($BF9B), CMDADR ($BF9C/$BF9D), SAVEX ($BF9E), and SAVEY ($BF9F). First, bit 7 of MLIACTV is changed from 0 to 1; this is done so that an interrupt-handling subroutine can easily determine if an interrupt condition has occurred during an MLI operation. (We'll see why it's important to know this information in Chapter 6.) Then the current values of the X and Y registers are stored in SAVEX and SAVEY. Finally, the address of

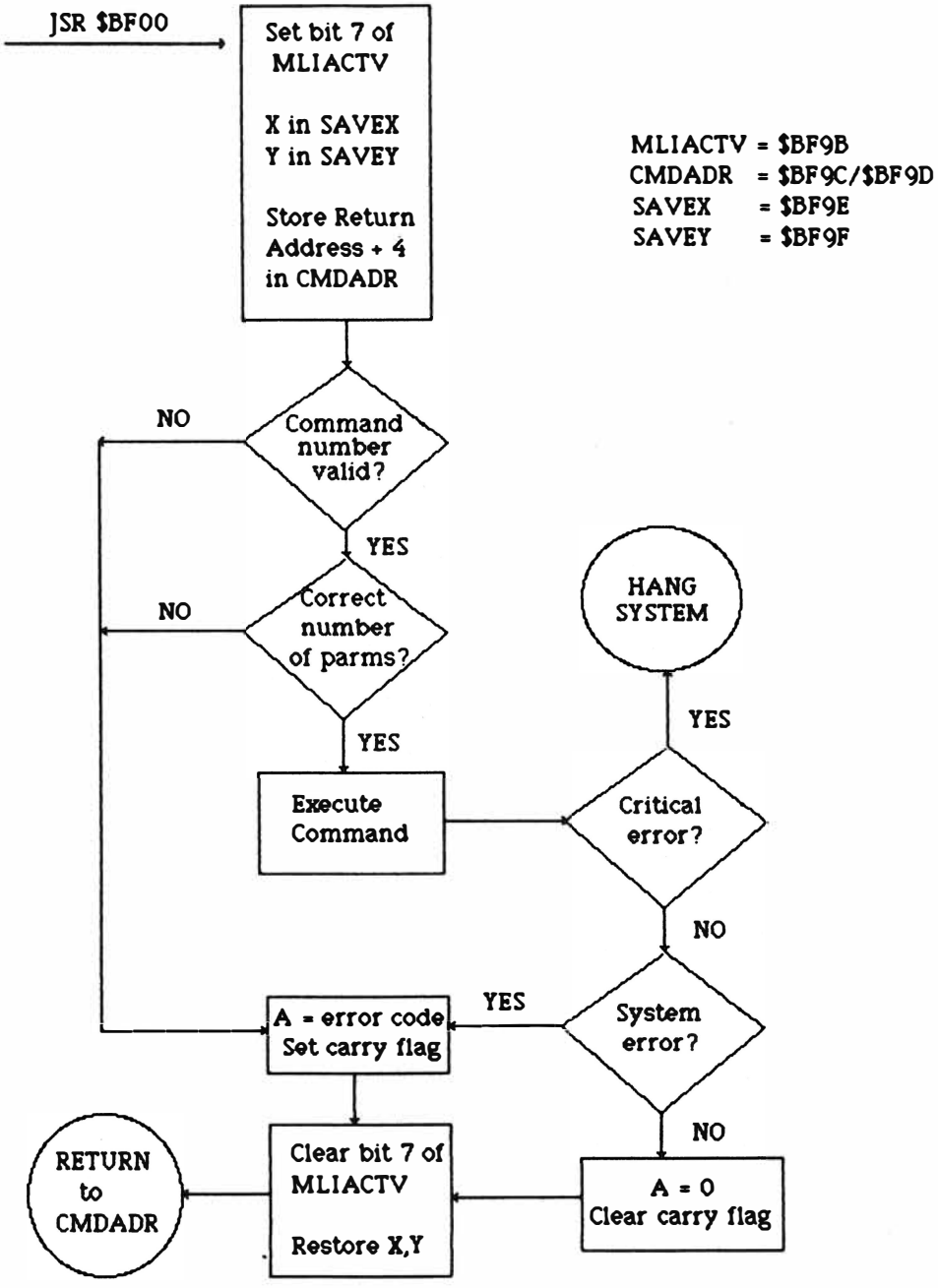


Figure 4-1. A flowchart of MLI operations

the instruction immediately following the three data bytes that follow the "JSR $BF00" instruction is stored at CMDADR. Control passes to this address after the MLI command has been executed. (The MLI modifies the return address that the JSR places on the 6502 stack to ensure that control passes to this address rather than to the address following the "JSR $BF00" instruction.)

The MLI determines which command has been requested by examining the value stored in the byte immediately following the "JSR $BF00" instruction. This byte must contain the unique identifier code (or *command number*) associated with the MLI command being requested. If an unknown command number is encountered, an MLI system error occurs. (We'll see how to handle such errors in a moment.)

The two bytes following the command number contain the address (low-order byte first) of a parameter table to be used by the MLI command. The parameter table begins with a byte that contains the number of parameters in the table; the rest of the table holds data that the MLI command requires in order to process your request. After the command is executed, the table will also hold any results that are generated. As we will see in the following sections, the contents of the parameter table associated with each MLI command have been carefully defined.

Parameters that are passed to an MLI subroutine are of two types: *pointers* and *values*. A pointer is a two-byte quantity that holds the address (low-order byte first) of a data structure to which it is said to be pointing. (Typical data structures are an I/O buffer or an ASCII pathname preceded by a length byte.) A value is a one-, two-, or three-byte quantity that holds a binary number. Multibyte values are always stored with the low-order bytes first.

The parameters returned by an MLI subroutine are called results. A result is usually a one-, two-, or three-byte numeric quantity (with the low-order bytes first), but it can also be a two-byte pointer, depending on the command that produced it.

If the number at the start of the parameter table does not correspond to the parameter count expected by the command, then a system error will occur. Otherwise, the MLI will execute the command.

While a command is being executed, a *critical* error condition may occur. Critical errors are very rare and only occur if ProDOS data areas have been overwritten by a runaway program or if a 6502 interrupt occurs and there is no interrupt handler installed to handle it. You cannot recover from such errors without rebooting the system. When a critical error occurs, the MLI executes a "JSR $BF0C" instruction. The subroutine at $BF0C (SYSDEATH) causes the following message to appear

```
INSERT SYSTEM DISK AND RESTART -ERR xx
```

where "xx" represents a two-digit hexadecimal error code. Four error conditions are possible:

01	Unclaimed interrupt error.
0A	Volume control block damaged.

0B File control block damaged.
0C Allocation block damaged.

The volume control, file control, and allocation blocks are internal ProDOS data structures used to enable it to handle disk volumes efficiently and to open files.

In the normal course of events, the MLI command will simply finish up by restoring the values of the X and Y registers (from SAVEX and SAVEY) and then, if a system error has occurred (see below), by executing a "JSR $BF09" instruction. The subroutine at $BF09 (SYSERR) causes an error code to be stored in SERR ($BF0F).

Note that since the X and Y registers are preserved by the MLI, there is no need to save them before calling the MLI.

Finally, control passes to the instruction immediately following the pointer to the address of the parameter table ("BCS ERROR" in the above example). Recall that this address was stored at CMDADR ($BF9C/$BF9D) when the MLI interpreter first took over.

MLI System Error Handling

Any error that is not a critical error is called a system error. Such errors can result for many reasons: specifying an illegal pathname, writing to a write-protected disk, opening a non-existent file, and so on.

If no system error occurred during execution of an MLI command, the 6502 accumulator will be 0, the carry flag will be clear (0), and the zero flag will be set (1).

If an error did occur, however, then the accumulator will hold the error code number, the carry flag will be set (1), and the zero flag will be clear (0). This means that you can use a BCS or a BNE instruction to branch to the error-handling portion of your code (we've used a BCS instruction in the example above).

You should always make it a point of handling error conditions after an MLI command ends. If you don't, you will undoubtedly have a program that won't always work properly. (For example, think of the consequences of writing to a file that could not be opened because it did not exist!)

For debugging purposes it's often handy to have a special subroutine available that can be called to print out helpful status information when an MLI error occurs. Such a subroutine is shown in Table 4-1. When it is called, the message

```
MLI ERROR $xx OCCURRED AT LOCATION $yyyy
```

is displayed, where "xx" is the two-digit hexadecimal error code and "yyyy" is the address that the MLI interpreter stored in CMDADR before trying to execute the command. This address is located six bytes after the "JSR $BF00" instruction that caused the error.

Table 4-1. A standard MLI error-handling subroutine.

```
                        1  **********************
                        2  *  General Purpose  *
                        3  * MLI Error Handler *
                        4  **********************
                        5
                        6  CMDADR  EQU  $BF9C      ;Return address for MLI call
                        7
                        8  CROUT   EQU  $FD8E      ;Print a CR
                        9  PRHEX   EQU  $FDDA      ;Print byte as two hex digits
                        10 COUT    EQU  $FDED      ;Standard output subroutine
                        11
                        12         ORG  $300
                        13
0300: 48                14 ERROR   PHA             ;Save error code on stack
                        15
0301: A0 00             16         LDY  #0
0303: B9 2E 03          17 E1      LDA  ERRMSG,Y
0306: F0 06             18         BEQ  E2
0308: 20 ED FD          19         JSR  COUT       ;Print first part of message
030B: C8                20         INY
030C: D0 F5             21         BNE  E1         ;(always taken)
                        22
030E: 68                23 E2      PLA             ;Get error code back
030F: 20 DA FD          24         JSR  PRHEX      ; and print it.
                        25
0312: A0 00             26         LDY  #0
0314: B9 3B 03          27 E3      LDA  ERRMSG1,Y
0317: F0 06             28         BEQ  E4
0319: 20 ED FD          29         JSR  COUT       ;Print second part of message
031C: C8                30         INY
031D: D0 F5             31         BNE  E3         ;(always taken)
```

```
                        32
031F: AD 9D BF  33 E4       LDA  CMDADR+1
0322: 20 DA FD  34          JSR  PRHEX      ;Print high part of address
0325: AD 9C BF  35          LDA  CMDADR
0328: 20 DA FD  36          JSR  PRHEX      ;Print low part of address
032B: 4C 8E FD  37          JMP  CROUT
                        38
032E: 8D        39 ERRMSG   DFB  $8D
032F: CD CC C9  40          ASC  "ML I ERROR $"
0332: A0 C5 D2 D2 CF D2 A0 A4
033A: 00        41          DFB  0
033B: A0 CF C3  42 ERRMSG1  ASC  " OCCURRED AT LOCATION $"
033E: C3 D5 D2 D2 C5 C4 A0 C1
0346: D4 A0 CC CF C3 C1 D4 C9
034E: CF CE A0 A4
0352: 00        43          DFB  0
```

The system error codes that are used by the MLI interpreter are summarized in Table 4-2. For convenience, Table 4-2 also indicates the Applesoft error messages that are displayed when an MLI error is returned to the BASIC.SYSTEM command that called it.

Table 4-2. MLI Error Codes.

Error Code	*BASIC.SYSTEM Error Message*	*Description*
$00	[not applicable]	No error occurred.
$01	I/O ERROR	An invalid MLI command number was specified.
$04	I/O ERROR	An incorrect "number of parms" value was specified in the parameter table.
$25	I/O ERROR	The ProDOS internal interrupt vector table is full (that is, four interrupt vectors have already been installed).
$27	I/O ERROR	A disk I/O error occurred that prevented the proper transfer of data. Such errors can be caused by a damaged disk or by the improper insertion of a disk.
$28	NO DEVICE CONNECTED	The specified disk drive device is not present.
$2B	WRITE PROTECTED	A write operation failed because the disk is write-protected.
$2E	I/O ERROR	An ON _ LINE operation failed because a disk containing an open file has been removed from its drive.
$40	SYNTAX ERROR	The pathname syntax is invalid because one of the filenames or directory names specified does not follow the ProDOS naming rules or because a partial pathname was specified and a prefix is not active.
$42	NO BUFFERS AVAILABLE	An attempt was made to open a ninth file. ProDOS allows only eight files to be open at once.
$43	FILE NOT OPEN	The file reference number is invalid. This error will occur if the wrong reference number is specified for an open file or if the reference number for a closed file is used.
$44	PATH NOT FOUND	The specified path was not found. This means that one of the subdirectory names, in an otherwise valid pathname, does not exist.

Table 4-2. MLI Error Codes (continued).

Error Code	*BASIC.SYSTEM* Error Message	*Description*
$45	PATH NOT FOUND	The specified volume directory was not found. This means that the volume directory name, in an otherwise valid pathname, does not exist. A common reason for this error is changing a disk without changing the active prefix.
$46	I/O ERROR	The specified file was not found. This means that the last filename, in an otherwise valid pathname, does not exist.
$47	DUPLICATE FILE NAME	The specified filename already exists. This error occurs when a file is being renamed or created and the new name specified is already in use.
$48	DISK FULL	The disk is full. This error can occur during a write operation when there are no free blocks on the disk to hold the data.
$49	DIRECTORY FULL	The volume directory is full. Only 51 files can be stored in the volume directory.
$4A	I/O ERROR	The format of the file specified is unknown or is not compatible with the version of ProDOS being used.
$4B	FILE TYPE MISMATCH	The "storage type code" for the file is invalid. Valid codes are $1 (seedling), $2 (sapling), $3 (tree), $D (subdirectory file), $E (volume directory header), and $F (subdirectory header).
$4C	END OF DATA	An end-of-file condition was encountered during a read operation.
$4D	RANGE ERROR	The specified value for MARK is out of range. When MARK (the "position-in-file") pointer is being changed, it cannot be set higher than EOF.
$4E	FILE LOCKED	The file cannot be accessed. This error occurs when the action prohibited by the "access code" byte is requested (RENAME, DESTROY, READ, or WRITE).
$50	FILE BUSY	The command is invalid because the file is open. The OPEN, RENAME, and DESTROY commands only operate on closed files.

(continued)

Table 4-2. MLI Error Codes (continued).

Error Code	*BASIC.SYSTEM* Error Message	*Description*
$51	I/O ERROR	The directory count is wrong. This error occurs if the file counter stored in the directory header is different than the actual number of files.
$52	I/O ERROR	This is not a ProDOS or SOS disk. This error occurs if the MLI senses a directory structure inconsistent with ProDOS or SOS.
$53	INVALID PARAMETER	A parameter is invalid because it is out of the allowable range.
$55	I/O ERROR	The volume control block table is full. This error occurs if eight files on eight separate disk drives are open and the ON _ LINE command is called for a drive having no open files.
$56	NO BUFFERS AVAILABLE	The buffer address is invalid because it conflicts with memory areas marked as "in use" by the ProDOS system bit map or because it does not start on a page boundary.
$57	I/O ERROR	Disks are on line that have the same volume directory name.
$5A	I/O ERROR	The volume bit map indicates that a block beyond the number available on the disk device is free for use. This error occurs if the volume bit map has been damaged.

MLI Command Descriptions

In the following sections, we will examine, in alphabetical order, each of the MLI commands that are supported by ProDOS. The command numbers follow the command names in parentheses.

Before we begin, here is a brief summary of each of the commands supported by the ProDOS MLI interpreter (this time in numeric order):

ALLOC _ INTERRUPT ($40)	Install an interrupt-handling subroutine.
DEALLOC _ INTERRUPT ($41)	Remove an interrupt-handling subroutine.
QUIT ($65)	Transfer control to another system program (interpreter).

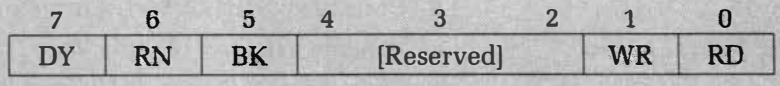
READ _ BLOCK ($80)	Read a data block from the disk.
WRITE _ BLOCK ($81)	Write a data block from the disk.
GET _ TIME ($82)	Read the current date and time.
CREATE ($C0)	Create a new file.
DESTROY ($C1)	Destroy an existing file or directory.
RENAME ($C2)	Rename a file.
SET _ FILE _ INFO ($C3)	Change the directory entry for a file.
GET _ FILE _ INFO ($C4)	Read the directory entry for a file.
ON _ LINE ($C5)	Determine the name of the volume directory for an installed disk.
SET _ PREFIX ($C6)	Set the pathname prefix.
GET _ PREFIX ($C7)	Read the pathname prefix.
OPEN ($C8)	Open a file for I/O operations.
NEWLINE ($C9)	Specify the character that will terminate a file read operation.
READ ($CA)	Read data from a file.
WRITE ($CB)	Write data to a file.
CLOSE ($CC)	Close a file.
FLUSH ($CD)	Flush a file buffer.
SET _ MARK ($CE)	Set the value of the MARK (position-in-file) pointer.
GET _ MARK ($CF)	Get the value of the MARK (position-in-file) pointer.
SET _ EOF ($D0)	Set the value of the EOF (end-of-file) pointer.
GET _ EOF ($D1)	Get the value of the EOF (end-of-file) pointer.
SET _ BUF ($D2)	Change the position of a file buffer.
GET _ BUF ($D3)	Get the position of a file buffer.

You should note that there is no pathname prefix in effect when ProDOS is first started up. This means that any pathname specified in an MLI command parameter list must be a full pathname and not a partial pathname or a bare filename. To simplify your code, it is a good idea to use the SET _ PREFIX ($C6) command to set the prefix string to the required pathname before calling other commands. If you simply want to set the prefix to the name of the volume directory on a given disk, you can use the ON _ LINE ($C5) command to get its name before using SET _ PREFIX ($C6). An example of a program that does this is given below in the discussion of the SET _ PREFIX command.

Three of the MLI commands, CREATE, GET _ FILE _ INFO and SET _ FILE _ INFO, use a parameter called *access code* that describes the read/write

status of a file. The meaning of each bit in the access code byte is explained in Figure 4-2.

Many of the MLI commands accept or return date and time values in their parameter lists. These values are stored in the same special packed form that is



DY = destroy-enable bit
RN = rename-enable bit
BK = backup-needed bit
WR = write-enable bit
RD = read-enable bit

If a bit is "1", then the function attributed to that bit is enabled; if it is "0", then it is disabled. The reserved bits must always be 0 (disabled).

If the DY, RN, and WR bits are all "1," then the file is said to be "unlocked;" if all three are "0," then the file is "locked." Any other combination means that the file is subject to "restricted access" limitations.

The backup-needed bit is always set to "1" whenever a file is written to. This permits the writing of a backup program that can perform incremental backups (that is, the backing up of only those programs that have been modified since the last backup). It is the responsibility of the backup program to clear the backup-needed bit to "0" after a copy of the file has been made.

Figure 4-2. Description of the "access code" byte.

used to store the ProDOS system global page TIME and DATE values. See Figure 8-1 in Chapter 8 for a description of this format.

Another popular MLI parameter is *file type code*. This is a number from 0 to 255 that serves to identify the general file type. The standard meanings of the ProDOS file type codes are given in Table 4-3.

Table 4-3. ProDOS file type codes.

File Type Code	*BASIC.SYSTEM CATALOG Mnemonic*	*Description*
$00		Typeless file (SOS too)
$01		Bad block file (SOS too)
$02		Pascal code file (SOS)
$03		Pascal text file (SOS)
$04	TXT	ASCII text file (SOS too)
$05		Pascal data file (SOS)
$06	BIN	Binary file (SOS too)
$07		Font file (SOS)
$08		Graphics screen file (SOS)
$09		Business BASIC program file (SOS)
$0A		Business BASIC data file (SOS)

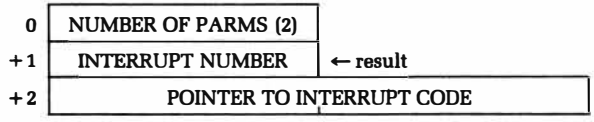
$0B		Word processor file (SOS)
$0C		System file (SOS)
$0D-$0E		Reserved (SOS)
$0F	DIR	Directory file (SOS too)
$10		RPS data file (SOS)
$11		RPS index file (SOS)
$12-$18		Reserved (SOS)
$19	ADB	AppleWorks database file
$1A	AWP	AppleWorks word processing file
$1B	ASP	AppleWorks spreadsheet file
$1C-$BF		Reserved (SOS)
$C0-$EE		ProDOS reserved
$EF	PAS	Pascal file
$F0	CMD	ProDOS added command file
$F1-$F8		ProDOS user-defined files
$F9		ProDOS reserved
$FA	INT	Integer BASIC program file
$FB	IVR	Integer BASIC variables file
$FC	BAS	Applesoft program file
$FD	VAR	Applesoft variables file
$FE	REL	EDASM relocatable code file
$FF	SYS	ProDOS system file

Note: SOS is the operating system for the Apple ///.

ALLOC_INTERRUPT ($40)

Purpose: To install the address of an interrupt-handling subroutine in the ProDOS internal interrupt vector table. The addresses of up to four such subroutines can be installed; earlier-installed subroutines will have higher priorities than later-installed ones.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 2).

INTERRUPT NUMBER — This is the identification number assigned to the interrupt handler. This number must be used when the interrupt handler is removed using the DEALLOC_INTERRUPT command.

POINTER TO INTERRUPT CODE — This is a pointer to the starting address of the interrupt-handling subroutine. Control passes to this address when an interrupt occurs.

Special Note: You must install an interrupt-handling subroutine *before* enabling interrupts on the hardware device. If you don't, then the system will crash as it tries to execute a subroutine that just isn't there.

Common Error Codes:

$25 — The interrupt vector table is full. **Solution:** remove one of the four active interrupt vectors (using DEALLOC_INTERRUPT) and then try again.

Other possible error codes are: $04.

Programming Example:

We'll look in greater detail at how ProDOS handles interrupts in Chapter 6. Meanwhile, to whet your appetite, here's how to install an interrupt handling subroutine that has been loaded into memory at location $300:

```
JSR MLI
DFB $40        ;ALLOC_INTERRUPT
```

```
          DA   PARMTBL ;Address of parameter table
          BCS  ERROR   ;Branch if error occurred
          ,
          ,
          ,
          RTS
PARMTBL   DFB  2       ;The # of parameters
          DS   1       ;Interrupt number is
                        returned here
          DA   $300    ;Address of interrupt
                        subroutine
```

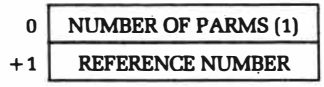
The returned "interrupt number" should be stored someplace safe because it's going to be needed when the interrupt handler is removed using the DEALLOC_INTERRUPT ($41) command.

CLOSE ($CC)

Purpose:

To close an open file. This process involves writing the contents of the data portion of the file's buffer to the file (if necessary) and then updating the file's directory entry. Once this is done, the 1024-byte file buffer is released to the system. Any one file, or all of them at or above a given level, can be closed at once.

Parameter Table:



LEVEL ($BF94) contains the current file level.

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 1).

REFERENCE — This is the reference number assigned to the file when it was opened.

If "reference number" is set equal to 0, then all files at the current level, or above it, will be closed. Suppose you have some files that were opened at level 3 and others at level 2. To close all the level 3 files, but not the level 2 files, store a 3 in LEVEL ($BF94) before using this command.

Common Error Codes:

$2B — The disk is write-protected. **Solution:** remove the write-protect sticker from the diskette.

$43 — The file reference number is invalid. **Solution:** only use reference numbers for open files.

Other possible error codes are: $04, $27.

Programming Example:

To close all open files, you can use the CLOSE command with "reference number" and LEVEL ($BF94) both equal to 0. Here's how to do it:

```
        LDA #0
        STA LEVEL     ;Set LEVEL to 0
        JSR MLI
        DFB $CC       ;CLOSE
        DA  PARMTBL   ;Address of parameter table
        BCS ERROR     ;Branch if error occurred
        '
        '
        '
        RTS

PARMTBL DFB 1         ;The # of parameters
        DFB 0         ;reference number = 0
                       (close all files)
```

CREATE ($C0)

Purpose:

To create a new disk file. This is done by placing an appropriate entry for the file in a specified directory. Every file, except a volume directory file, must be created using this subroutine. (Volume directory files are to be created by the program used to format a ProDOS disk.)

Parameter Table:

0	NUMBER OF PARMS (7)
+1	POINTER TO PATHNAME
+3	ACCESS CODE
+4	FILE TYPE CODE
+5	AUXILIARY TYPE CODE
+7	STORAGE TYPE CODE
+8	DATE OF CREATION
+10	TIME OF CREATION

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 7).

POINTER TO PATHNAME — A two-byte pointer to a byte string that describes the pathname of the file to be created. The first byte in this string represents the length of the pathname (64 characters maximum) and is followed by the pathname itself (in ASCII). If the pathname is not preceded by a slash ("/"), then ProDOS will automatically append it to the currently active prefix to create a full pathname.

ACCESS CODE — This byte contains five one-bit codes that define the access attributes of the file to be created. See Figure 4-2 for a description of these bits. The "backup-needed" bit of the access code is automatically set to 1 by this command.

FILE TYPE CODE — This byte contains a code that indicates the type of data the file holds. See Table 4-3 for a description of the ProDOS file type codes.

AUXILIARY TYPE CODE — These two bytes are used for special purposes by different interpreters (system programs). For example, BASIC.SYSTEM stores the default loading address

here for SYS files ($2000) and BIN files, the field length for TXT files, the loading address for BAS files (usually $801), or the starting address of a block of variables for VAR files.

STORAGE TYPE CODE — This byte indicates how the file is to be stored on the disk. Linked list directory files are stored with a storage type code = $0D; standard tree-structured data files are stored with a storage type code = $01.

DATE OF CREATION — These two bytes contain the date (year, month, day) on which the file was created. The format of these bytes is as shown in Figure 8-1. If these bytes are both zero, the current date will be automatically stored in the "creation date" field of the file's directory entry; otherwise, the date stored here will be used.

TIME OF CREATION — These two bytes contain the time (hour, minute) at which the file was created. The format of these bytes is shown in Figure 8-1. If these bytes are both zero, the current time will be automatically stored in the "creation time" field in the file's directory entry; otherwise, the time stored here will be used.

Common Error Codes:

$2B — The disk is write-protected. **Solution:** remove the write-protect sticker from the diskette.

$40 — The pathname contains invalid characters or a full pathname was not specified (and no prefix is active). **Solution:** make sure that the filenames and directory names specified in the pathname adhere to the naming rules described in Chapter 2 and, if a partial pathname was specified, that a prefix is active.

$44 — A directory in the pathname was not found. **Solution:** double-check the spelling of the pathname, insert the disk containing the correct directory, or change the active prefix.

$45 — The volume directory was not found. **Solution:** double-check the spelling of the volume directory name, insert the correct disk, or change the active prefix.

$47 — The filename specified already exists. **Solution:** give the file a name not used by any other file in the same subdirectory or destroy the old file before creating the new one.

$48	The disk is full. **Solution:** create the file on another disk or delete a file from the current disk and try again.
$49	The volume directory is full. **Solution:** create the file on another disk or delete a file from the volume directory on the current disk and try again. (Only 51 files can be stored in the volume directory.)

Other possible error codes are: $04, $27, $4B.

Programming Example:

Here is a short program segment that can be used to create a standard textfile. The filename for the textfile is JUPITER and the full pathname is /PLANETS/JUPITER.

```
            JSR  MLI
            DFB  $C0          ;CREATE code
            DA   PARMTBL      ;Address of parameter
                               table
            BCS  ERROR        ;Branch if error occurred
            ,
            ,
            ,
            RTS

PARMTBL     DFB  7            ;7 parameters
            DA   PATHNAME
            DFB  $E3          ;standard access code
                               (unlocked)
            DFB  $04          ;file type = 4 (textfile)
            DW   0            ;auxiliary type (0 for
                               sequential file)
            DFB  $01          ;storage type = 1
                               (standard file)
            DW   0            ;date of creation
                               (current date)
            DW   0            ;time of creation
                               (current time)

PATHNAME    DFB  16           ;Length of pathname
            ASC  '/PLANETS/JUPITER'  ;Pathname (in
                                      ASCII)
```

If you want to specify a creation date and time other than the current date and time, store the desired values (in the format shown in Figure 8-1) in the parameter list before calling CREATE.

DEALLOC _ INTERRUPT ($41)

Purpose: To remove the address of an interrupt-handling subroutine from the ProDOS internal interrupt vector table.

Parameter Table:

0	NUMBER OF PARMS (1)
+1	INTERRUPT NUMBER

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 1).

INTERRUPT NUMBER — This is the identification number for the interrupt handler. It was assigned when the handler was installed using the ALLOC _ INTERRUPT ($40) command.

Special Note: You must not remove an interrupt-handling subroutine until you have first disabled interrupts from the hardware device. If you don't, the system will crash.

Common Error Codes:

$53 — The "interrupt number" parameter is not in the valid 1-4 range. **Solution:** try again with an "interrupt number" of 1, 2, 3, or 4. (The number is assigned to an interrupt handler when it is first installed with ALLOC _ INTERRUPT.)

Other possible error codes are: $04.

Programming Example:

Here's how to remove the interrupt vector table entry for an interrupt-handling subroutine assigned the code number 1 when it was installed using the ALLOC _ INTERRUPT command:

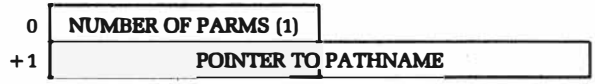
```
          JSR MLI
          DFB $41       ;DEALLOC_INTERRUPT
          DA  PARMTBL   ;Address of parameter table
          BCS ERROR     ;Branch if error occurred
          '
          '
          '
          RTS
PARMTBL   DFB 1         ;The # of parameters
          DFB 1         ;Interrupt code number
```

DESTROY ($C1)

Purpose:

To delete an existing file (provided that it is closed) from the disk. When a file is destroyed, all the blocks that it uses are freed up. All files except volume directory files can be destroyed; subdirectory files, however, must be empty before they can be destroyed.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 1).

POINTER TO PATHNAME — A two-byte pointer to a byte string that describes the pathname of the file to be destroyed. The first byte in this string represents the length of the pathname (64 characters maximum) and is followed by the pathname itself (in ASCII). If the pathname is not preceded by a slash ("/"), then ProDOS will automatically append it to the currently active prefix to create a full pathname.

Common Error Codes:

$2B — The disk is write-protected. **Solution:** remove the write-protect sticker from the diskette.

$40 — The pathname contains invalid characters or a full pathname was not specified (and no prefix is active). **Solution:** make sure that the filenames and directory names specified in the pathname adhere to the naming rules described in Chapter 2 and, if a partial pathname was specified, that a prefix is active.

$44 — A directory in the pathname was not found. **Solution:** double-check the spelling of the pathname, insert the disk containing the correct directory, or change the active prefix.

$45 — The volume directory was not found. **Solution:** double-check the spelling of the volume directory name, insert the correct disk, or change the active prefix.

$46	The file was not found. **Solution:** double-check the spelling of the filename and try again.
$4E	The file cannot be accessed. **Solution:** set the "destroy-enabled" bit of the access code to 1 using SET_FILE_INFO.
$50	The file is open. **Solution:** close the file first.

Other possible error codes are: $04, $27, $4A.

Programming Example:

Consider a situation in which the currently active prefix is /DEMOS/GAMES. To destroy a file that has a full pathname of /DEMOS/GAMES/TRIVIA.BLITZ, you could use the following program:

```
            JSR MLI
            DFB $C1         ;DESTROY code
            DA  PARMS       ;Address of parameter
                             table
            BCS ERROR       ;Branch if error occurred
            '
            '
            '
            RTS

PARMS       DFB 1           ;1 parameter
            DA  PATHNAME

PATHNAME    DFB 12          ;Length of pathname
            ASC 'TRIVIA.BLITZ' ;Pathname (in ASCII)
```

Notice that it was not necessary to specify the full pathname in this program. ProDOS automatically appends the name that is specified to the currently active prefix to create the full pathname that it acts on.

FLUSH ($CD)

Purpose: To force ProDOS to write the contents of the data portion of a file's buffer to the disk and to update the file's directory entry. This is done without closing the file. Any one file, or all of them at or above a given level, can be flushed at once.

Parameter Table

0	NUMBER OF PARMS (1)
+1	REFERENCE NUMBER

LEVEL ($BF94) contains the current file level.

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 1).

REFERENCE NUMBER — This is the reference number that was assigned to the file when it was opened.

If "reference number" is set equal to 0, then all files at the current level, or above it, will be flushed. Suppose you have some files that were opened at level 3 and others at level 2. To flush all the level 3 files, but not the level 2 files, store a 3 in LEVEL ($BF94) before using this command.

Common Error Codes:

$2B — The disk is write-protected. **Solution:** remove the write-protect sticker from the diskette.

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

Other possible error codes are: $04, $27.

Programming Example:

To flush all open files at or above level 2, use the FLUSH command with "reference number" equal to 0 and LEVEL ($BF94) equal to 2. Here's the code:

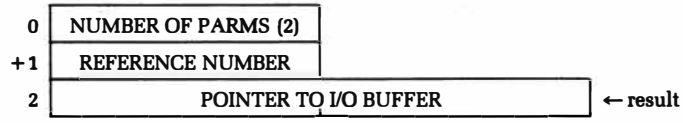
```
LDA #2
STA LEVEL    ;Set LEVEL to 2
JSR MLI
```

```
        DFB $CD       ;FLUSH
        DA  PARMTBL   ;Address of parameter table
        BCS ERROR     ;Branch if error occurred
        ,
        ,
        ,
        RTS

PARMTBL DFB 1         ;The # of parameters
        DFB 0         ;reference number = 0
                       (close all files)
```

GET_BUF ($D3)

Purpose: To determine the starting address of the 1024- byte buffer that is being used by an open file.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 2).

REFERENCE NUMBER — This is the reference number that was assigned to the file when it was opened.

POINTER TO I/O BUFFER — A two-byte pointer to the 1024-byte file buffer that is used by the open file. The low-order byte of this pointer is always $00 (that is, the buffer begins on a page boundary).

Common Error Codes:

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

Other possible error codes are: $04.

Programming Example:

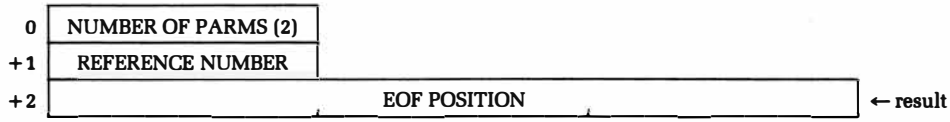
You can use the following program to determine the address of the file buffer for file #2. After the GET_BUF command is executed, the address will be stored at BUFFPTR.

```
            JSR MLI
            DFB $D3       ;GET_BUF
            DA  PARMTBL   ;Address of parameter table
            BCS ERROR     ;Branch if error occurred
            '
            '
            '
PARMTBL     DFB 2         ;The # of parameters
            DFB 2         ;File reference number
BUFFPTR     DS  2         ;Buffer address is returned
                           here
```

GET_EOF ($D1)

Purpose: To determine the value of the current end-of- file pointer (EOF) of an open file.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 2).

REFERENCE NUMBER — This is the reference number that was assigned to the file when it was opened.

EOF POSITION — This is a three-byte value that holds the current EOF position. This value is equal to the size of the file (in bytes).

Common Error Codes:

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

Other possible error codes are: $04.

Programming Example:

You can use the GET_EOF command to quickly determine how big an open file is. For example, after you call this program:

```
            JSR MLI
            DFB $D1       ;GET_EOF
            DA  PARMTBL   ;Address of parameter
                           table
            BCS ERROR     ;Branch if error occurred
            ,
            ,
            ,
            RTS

PARMTBL     DFB 2         ;The # of parameters
            DFB 1         ;File reference number
POSITION    DS  3         ;Current EOF position
```

The size of open file #1 will be stored at POSITION (low-order byte first).

GET_FILE_INFO ($C4)

Purpose:

To read the information that is stored in a file's directory entry. This includes the file's access code, file type code, auxiliary type code, and the date and time at which the file was last modified.

Parameter Table:

0	NUMBER OF PARMS (10)	
+1	POINTER TO PATHNAME	
+3	ACCESS CODE	← result
+4	FILE TYPE CODE	← result
+5	AUXILIARY TYPE CODE (*)	← result
+7	STORAGE TYPE CODE	← result
+8	BLOCKS USED (*)	← result
+10	DATE OF MODIFICATION	← result
+12	TIME OF MODIFICATION	← result
+14	DATE OF CREATION	← result
+16	TIME OF CREATION	← result

(*) When "pointer to pathname" points to a volume directory name, the volume size (in blocks) is returned in "auxiliary type code" and the number of blocks currently in use by all files is returned in "blocks used."

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 10).

POINTER TO PATHNAME — A two-byte pointer to a byte string that describes the pathname of the file to be used. The first byte in this string represents the length of the pathname (64 characters maximum) and is followed by the pathname itself (in ASCII). If the pathname is not preceded by a slash ("/"), then ProDOS will automatically append it to the currently active prefix to create a full pathname.

ACCESS CODE — This byte contains five one-bit codes that define the access attributes of the file to be created. See Figure 4-2 for a description of these bits. The "backup-needed" bit of the access code is automatically set to 1 by this command.

FILE TYPE CODE — This byte contains a code that indicates the type of data the file holds. See Table 4-3 for a description of the ProDOS file type codes.

AUXILIARY TYPE CODE — These two bytes are used for special purposes by different interpreters (system programs). For example, BASIC.SYSTEM stores the default loading address here for SYS files ($2000) and BIN files, the field length for TXT files, the loading address for BAS files (usually $801), or the starting address of a block of variables for VAR files.

Exception: If "pointer to pathname" points to a volume directory name, then the auxiliary type code will hold the volume size (in blocks).

STORAGE TYPE CODE — This byte describes the physical organization of the file on the disk:

$01 = seedling file
$02 = sapling file
$03 = tree file
$0D = directory file
$0F = volume directory file

BLOCKS USED — These two bytes contain the total number of blocks used by the file for data storage and for index blocks.

Exception: If "pointer to pathname" points to a volume directory name, then "blocks used" contains the number of blocks in use on the disk by all files.

DATE OF MODIFICATION — These two bytes contain the date (year, month, day) on which the file was last modified. The format of these bytes is as shown in Figure 8-1.

TIME OF MODIFICATION — These two bytes contain the time (hour, minute) at which the file was last modified. The format of these bytes is as shown in Figure 8-1.

DATE OF CREATION — These two bytes contain the date (year, month, day) on which the file was created. The format of these bytes is as shown in Figure 8-1.

TIME OF CREATION — These two bytes contain the time (hour, minute) at which the file was created. The format of these bytes is as shown in Figure 8-1.

Common Error Codes:

$40 — The pathname contains invalid characters or a full pathname was not specified (and no prefix is active).

	Solution: make sure that the filenames and directory names specified in the pathname adhere to the naming rules described in Chapter 2 and, if a partial pathname was specified, that a prefix is active.
$44	A directory in the pathname was not found. **Solution:** double-check the spelling of the pathname, insert the disk containing the correct directory, or change the active prefix.
$45	The volume directory was not found. **Solution:** double-check the spelling of the volume directory name, insert the correct disk, or change the active prefix.
$46	The file was not found. **Solution:** double-check the spelling of the filename and try again.

Other possible error codes are: $04, $27.

Programming Example:

See the example given for the SET _ FILE _ INFO ($C3) command.

GET_MARK ($CF)

Purpose: To determine the value of the current position- in-file pointer (MARK) of an open file. Subsequent read or write operations take place at this position.

Parameter Table:

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 2).

REFERENCE NUMBER — This is the reference number that was assigned to the file when it was opened.

MARK POSITION — This is a three-byte value that holds the current MARK position.

Common Error Codes:

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

Other possible error codes are: $04.

Programming Example:

Here's a program that will read and display the current MARK position of an open file:

```
JSR MLI
DFB $CF          ;GET_MARK
DA  PARMTBL      ;Address of parameter
                  table
BCS ERROR        ;Branch if error
                  occurred
LDA POSITION+2
JSR PRBYTE       ;Print high part
                  (PRBYTE=$FDDA)
LDA POSITION+1
JSR PRBYTE       ;Print mid part
LDA POSITION
JSR PRBYTE       ;Print low part
```

```
            LDA #$8D
            JSR COUT          ;Followed by CR
                               (COUT=$FDED)
            ,
            ,
            ,
            RTS

PARMTBL     DFB 2             ;The # of parameters
            DFB 1             ;File reference number
POSITION    DS  3             ;Current MARK position
```

The system monitor subroutine called PRBYTE ($FDDA) prints the byte in the accumulator as two hexadecimal digits.

GET_PREFIX ($C7)

Purpose: To determine the current system prefix.

Parameter Table:

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 1).

POINTER TO DATA BUFFER — These two bytes point to a buffer in which the name of the currently active prefix is returned preceded by a length byte. The prefix name is preceded and followed by a slash. The buffer must be 65 bytes long in order to accommodate the longest possible prefix that might be used (64 characters) plus the preceding length byte.

Common Error Codes:

$56 — The buffer address is invalid because it has been marked as "in use" in the system bit map. **Solution:** specify a buffer address that does not conflict with areas already used by ProDOS itself or by ProDOS file buffers. The system bit map can be examined to determine the free and protected areas.

Other possible error codes are: $04.

Programming Example:

This program will get the current prefix and store it in the buffer beginning at "PATHNAME" (preceded by a length byte):

```
JSR MLI
DFB $C7         ;GET_PREFIX
DA  PARMTBL
BCS ERROR       ;Branch if error occurred
,
,
,
RTS
```

```
PARMTBL   DFB 1         ;The # of parameters
          DA  PATHNAME  ;Pointer to the prefix

PATHNAME  DS  65        ;Prefix length + name
```

GET_TIME ($82)

Purpose:

To read the date and time from an installed clock card and to store the result in the ProDOS system global page at DATE ($BF90/$BF91) and TIME ($BF92/$BF93).

Parameter Table:

[no parameter table but a "dummy" table must be pointed to by the calling subroutine]

Common Error Codes:

[no errors defined]

Programming Example:

When this command is called, the current date (year, month, day) and time (hour, minute) are stored in a reserved area of the ProDOS global page from $BF90-$BF93. The date is stored in the DATE locations ($BF90 and $BF91) and the time is stored in the TIME locations ($BF92 and $BF93) in the special packed format that is described in Figure 8-1 of Chapter 8.

Note, however, that this command will return the time only if a Thunderclock interface card, or a compatible such as the Prometheus Versacard or the Applied Engineering Timemaster II, has been installed in one of the Apple's slots. When ProDOS is first activated, a special clock I/O driver is installed that allows ProDOS to read this particular clock. We'll see how you can install your own clock drivers in Chapter 8.

The subroutine to use to read the current date and time is very simple since no parameter table is required and no errors can occur. Here it is:

```
JSR MLI
DFB $82       ;GET_TIME
DA  $0000     ;Dummy parameter table
RTS
```

NEWLINE ($C9)

Purpose:

To enable or disable newline read mode. When newline read mode is enabled, subsequent read operations will automatically terminate once a specified character (the *newline character*) has been read. When newline read mode is disabled, read operations will terminate when the end-of- file position is reached or when the requested number of characters has been read.

Parameter Table:

0	NUMBER OF PARMS (3)
+1	REFERENCE NUMBER
+2	ENABLE MASK
+3	NEWLINE CHARACTER

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 3).

REFERENCE NUMBER — This is the reference number that was assigned to the file when it was opened.

ENABLE MASK — This value is logically ANDed with each byte subsequently read from the file. If the result of the AND operation is the same as "newline character", then the read request will terminate; otherwise, the read will continue normally.

Exception: if "enable mask" is zero, then newline read mode is disabled and read operations are not affected.

NEWLINE CHARACTER — This is the value of the newline character. Read requests will automatically terminate if the logical AND of "enable mask" and the character being read equals "newline character."

Common Error Codes:

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

Other possible error codes are: $04.

Programming Example:

A common situation is one in which you want to read one line at a time from a textfile. Since each line in an Apple textfile is usually terminated by $0D, the standard ASCII code for the carriage return character, you could simply set "enable mask" equal to $FF and "newline character" to $0D prior to executing the NEWLINE command. However, in some cases the negative ASCII code for the carriage return character, $8D, is used as the end- of-line character instead. If you want to terminate a read on input of either $0D or $8D, then you must set "newline character" equal to $0D and "enable mask" to $7F.

Here is a program segment that will set the $0D/$8D newline read mode for you:

```
          JSR MLI
          DFB $C9       ;NEWLINE
          DA  PARMTBL   ;Address of parameter table
          BCS ERROR     ;Branch if error occurred
          ,
          ,
          ,
          RTS

PARMTBL   DFB 3         ;3 parameters
          DFB 1         ;File reference number (#1
                         assumed)
          DFB $7F       ;"enable mask"
          DFB $0D       ;"newline character"
```

ON_LINE ($C5)

Purpose:

To determine the volume directory name for a disk in a specified slot and drive, or of all active ProDOS volumes.

Parameter Table:

0	NUMBER OF PARMS (2)
+1	UNIT NUMBER CODE
+2	POINTER TO DATA BUFFER

Description of Parameters:

NUMBER OF PARMS

The number of parameters for this command (always 2).

UNIT NUMBER CODE

This byte contains the slot and drive number for the disk drive to be examined. The format of this byte is as follows:

7	6	5	4	3	2	1	0
DR	SLOT			[Unused]			

Each Apple disk controller card can have one or two drives connected to it. DR = 0 for drive 1 and DR = 1 for drive 2. SLOT contains the slot number for the disk controller card (1-7). The possible values for "unit number" are:

Slot →	1	2	3	4	5	6	7
Drive 1 →	$10	$20	$30	$40	$50	$60	$70
Drive 2 →	$90	$A0	$B0	$C0	$D0	$E0	$F0

(On the Apple //c, "unit number" can be $60 (built in drive), $E0 (external drive), or $B0 (/RAM volume).)

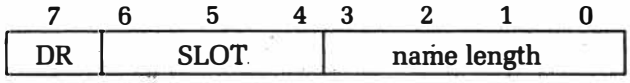
The "unit number" value for the /RAM volume, which can exist on an Apple //e or Apple //c only, is $B0 (that is, it is equivalent to a slot 3/drive 2 disk device).

Exception: If "unit number" is 0, then the volume names for all of the disk drives will be returned.

POINTER TO DATA BUFFER

This is a two-byte address that points to a buffer containing the volume name information returned for the specified disk drive. If "unit number" is 0, then the volume names for all the disk drives will be returned. Each volume name entry in the buffer is 16 bytes in length.

The first byte of each 16-byte record contains the drive and slot number for the disk drive and the length of its volume name in the following format:



DR and SLOT are defined in the same way described above for "unit number." "name length" contains the length of the volume name for the device defined by DR and SLOT. (If "name length" is zero, then an error occurred; in this case the error code is stored in the next byte. If the error code is $57 ("duplicate volume"), then the third byte of the record contains "unit number" for the duplicate.)

The next 15 bytes of the record contain the volume name (in standard ASCII). This name is not preceded by a slash ("/").

If "unit number" is 0, then the record after the last valid 16-byte record will begin with a $00 byte. A 256-byte buffer area must be reserved if ON_LINE is called with "unit number" set to 0.

Common Error Codes:

$27 Disk I/O error. **Solution:** try again with a diskette in the drive and with the drive door closed. If you have no luck, the disk is probably fatally damaged.

$28 The disk drive specified does not exist. **Solution:** try again with the "unit number" of a drive that does exist.

$2E A disk with an open file on it was removed from its drive prior to executing the command. **Solution:** close all files on the disk that is to be removed before executing the ON_LINE command.

$52 The disk in the drive specified by "unit number" is not a ProDOS disk. **Solution:** only use ProDOS-formatted disks with ProDOS!

$56 The buffer address is invalid because it has been marked as "in use" in the system bit map. **Solution:** specify a buffer address that does not conflict with areas already used by ProDOS itself or by ProDOS file buffers. The system bit map can be examined to determine the free and protected areas.

Other possible error codes are: $04, $55.

ON_LINE handles error conditions quite differently than the other MLI commands do. Generally speaking, if an error occurs, "name length" is set to 0 and the error code is stored in the second byte of the corresponding 16-byte record. THE ERROR CODE IS NOT STORED IN THE ACCUMULATOR AND THE CARRY FLAG IS NOT SET. Errors are handled in the standard way, however, when errors $55 (Volume Control Block full), $56 (buffer address invalid), and $04 (incorrect number of parameters) occur, however.

Programming Example:

This program will read the volume directory name of a disk that is in the slot 6, drive 2 disk device.

```
          JSR MLI
          DFB $C5       ;ON_LINE
          DA  PARMTBL   ;Address of parameter table
          BCS ERROR     ;Branch if error occurred
          ,
          ,
          ,
          RTS

PARMTBL   DFB 2         ;The # of parameters is
                         stored here
          DFB $E0       ;"unit number" = slot 6,
                         drive 2
          DA  BUFFER    ;Pointer to 16-byte buffer

BUFFER    DS  1         ;Slot/drive (bits 4-7) and
                         length of volume name
                         (bits 0-3)
          DS  15        ;Volume name (in ASCII)
```

If the volume directory name was ASM.FILES, then the byte stored at BUFFER would be $E9 and the bytes stored beginning at BUFFER+1 would be

```
41 53 4D 2E 46 49 4C 45 53
```

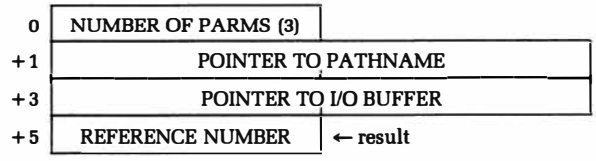
These are the ASCII codes for the characters in ASM.FILES.

OPEN ($C8)

Purpose:

To prepare a file for subsequent read or write operations. When a file is opened, the position-in-file pointer (MARK) points to the start of the file (that is, it has the value of 0) and its level is set equal to the number stored in LEVEL ($BF94). Up to 8 files can be open at one time.

Parameter Table:



LEVEL ($BF94) contains the current file level.

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 3).

POINTER TO PATHNAME — A two-byte pointer (low-order byte first) to a byte string that describes the pathname of the file to be opened. The first byte in this string represents the length of the pathname (64 characters maximum) and is followed by the pathname itself (in ASCII). If the pathname is not preceded by a slash ("/"), then ProDOS will automatically append it to the currently active prefix to create a full pathname.

POINTER TO I/O BUFFER — A two-byte pointer to a 1024-byte file buffer that is to be used by the open file. The low-order byte of this pointer must be $00 (that is, the buffer must begin on a page boundary).

The first half of the file buffer for a standard file contains a copy of the current file data block being accessed; the second half contains the current file index block. Only the first half of the buffer is used for a directory file; it contains the current directory file block.

REFERENCE NUMBER — This byte contains the reference number that ProDOS associates with the file after it has been opened. All file operations on open files use this reference number (instead of a pathname) to identify the file. The level

value associated with the file is set equal to value stored in LEVEL ($BF94) when the file is opened.

Common Error Codes:

$40	The pathname contains invalid characters or a full pathname was not specified (and no prefix is active). **Solution:** make sure that the filenames and directory names specified in the pathname adhere to the naming rules described in Chapter 2 and, if a partial pathname was specified, that a prefix is active.
$42	An attempt was made to open a ninth file. **Solution:** Close one of the open files before attempting to open another file.
$44	A directory in the pathname was not found. **Solution:** double-check the spelling of the pathname, insert the disk containing the correct directory, or change the active prefix.
$45	The volume directory was not found. **Solution:** double-check the spelling of the volume directory name, insert the correct disk, or change the active prefix.
$46	The file was not found. **Solution:** double-check the spelling of the filename and try again.
$50	The file is open. **Solution:** close the file first.
$56	The buffer address is invalid because it has been marked as "in use" in the system bit map. **Solution:** specify a buffer address that does not conflict with areas already used by ProDOS itself or by ProDOS file buffers. The system bit map can be examined to determine the free and protected areas.

Other possible error codes are: $04, $27, $4A.

Programming Example:

The following program will open (at level 3) a file called SESAME that resides in the currently active subdirectory. The disk buffer is located from $2000-$23FF.

```
LDA #3
STA LEVEL      ;Set level to 3 (LEVEL =
                $BF94)
JSR MLI
DFB $C8        ;OPEN
DA  PARMTBL    ;Address of parameter
                table
```

```
          BCS ERROR     ;Branch if error occurred
          ;
          ;
          ;
          RTS

PARMTBL   DFB 3         ;The # of parameters
          DA  PATHNAME  ;Pointer to pathname
          DA  $2000     ;Address of 1024-byte I/O
                         buffer
          DS  1         ;Reference number
                         returned here

PATHNAME  DFB 6         ;Length of file name
          ASC 'SESAME'  ;File name
```

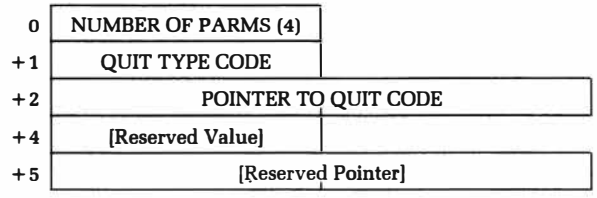
Once you run this program SESAME will open up at level 3.

QUIT ($65)

Purpose:

To transfer control from a ProDOS system program or interpreter (like BASIC.SYSTEM) to a special selector program that is used to load and execute another system program. Most system programs will automatically call the QUIT command when their exit commands are entered.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS	The number of parameters for this command (always 4).
QUIT TYPE CODE	This is the quit type code. The only quit type currently defined is 0.
POINTER TO QUIT CODE	This pointer is reserved for future use.
[Reserved Value]	This value is reserved for future use.
[Reserved Pointer]	This pointer is reserved for future use.

Note: the reserved pointers and values must be set to $0000 or 0 before executing this command.

Common Error Codes:

$04	The specified number of parameters is incorrect. **Solution:** Change the number of parameters to 4.

Programming Example:

The QUIT command is called by all well-designed system programs (like BASIC.SYSTEM, FILER, and CONVERT) when control is to be passed to another system program. Here is the calling sequence used to do this:

```
JSR MLI
DFB $65      ;QUIT
```

```
        DA   PARMTBL ;Address of parameter table
        BCS  ERROR   ;Branch if error occurred
        ,
        ,
        ,
        RTS

PARMTBL DFB  4       ;The number of parameters
        DFB  0
        DA   $0000
        DFB  0
        DA   $0000
```

When the QUIT command is executed, the code residing at $D100-$D3FF in the second 4K bank of bank-switched RAM (called the *selector code* or *dispatcher code*) is moved to location $1000 in main memory and then a "JMP $1000" instruction is executed.

When the standard ProDOS selector (the one that is installed on bootup) takes over, it performs the following steps:

- It asks you to enter the prefix and name of the next system program to be executed.
- It stores the length of the name of the system program at $280, followed by the ASCII-encoded name itself.
- It closes all open files.
- It clears the ProDOS system bit map and marks as "in use" zero page, the stack (page 1), the video RAM area (pages 4-7) and the ProDOS global page (page $BF).
- It enables the 40-column screen and connects the standard input (keyboard) and output (video) subroutines. (In your own selector program this can be done by executing the following group of instructions:

  ```
  LDA $C082    ;Read-enable monitor ROM
  JSR SETKBD   ;$FE89: connect keyboard
  STA $C00C    ;Turn off 80 columns (//e, //c)
  JSR SETVID   ;$FE93: connect 40-column video
  ```

  Note that the monitor ROM must be read-enabled before calling the SETKBD and SETVID monitor subroutines because it will be disabled when the selector takes over.)
- It loads the specified system program at $2000 and starts executing it by jumping to that location.

You can also install your own selector code if you wish. If you do, it must begin with a CLD instruction and it must perform the steps indicated above.

An alternative selector program that demonstrates how the above steps can be performed is shown in Table 4-4; to install the program at $D100 (bank2), BRUN it from disk. This selector is not at all interactive since it always passes control to the same system program: BASIC.SYSTEM in the volume directory

of the slot 6/drive 1 disk device. However, this is the program that most users of ProDOS will always want to select after leaving other system programs like the ProDOS utilities FILER (to copy disk volumes and files) and CONVERT (to transfer files between the DOS 3.3 and ProDOS environments). If you want to transfer control from, say, FILER to CONVERT or vice versa with this selector in effect, you will be forced to enter BASIC.SYSTEM first. Once there, you can use the "-" command to execute the other system program.

Depending on the system program to which control is being passed, your selector code can even specify that a particular applications program is to be run by the system program right away. In order for this to be done, the system program must adhere to a special "auto-run" protocol that we'll describe in Chapter 5.

Table 4-4. A ProDOS selector program.

```
1   ****************************************
2   *       ProDOS SELECTOR PROGRAM        *
3   *                                      *
4   * When this selector is called using   *
5   * the QUIT ($65) command, the system   *
6   * file called BASIC.SYSTEM in the      *
7   * slot 6/drive 1 device will be        *
8   * automatically executed.              *
9   ****************************************
10
11  PATHNAME EQU  $280       ;Full pathname stored here
12
13  FILEBUFF EQU  $1100      ;1K file buffer
14
15  SYSLOC   EQU  $2000      ;Start addr of system program
16
17  MLI      EQU  $BF00      ;Gateway to MLI
18  BITMAP   EQU  $BF58      ;System bit map
19
20  COL80OFF EQU  $C00C      ;Turn off 80 cols (//e,//c)
21
22  SSPACE   EQU  $D100      ;Selector space (in bank2)
23
24  HOME     EQU  $FC58      ;Clear the screen
25  SETKBD   EQU  $FE89      ;Connect keyboard
26  SETVID   EQU  $FE93      ;Connect video (40)
27
28           ORG  $2000
29
30  * Store selector code at $D100 in bank2 of
```

(continued)

Table 4-4. A ProDOS selector program (continued).

```
                        31  * bank-switched RAM:
                        32
2000: AD 81 C0          33          LDA   $C081
2003: AD 81 C0          34          LDA   $C081       ;Write-enable bank2 BSR
                        35
2006: A2 00             36          LDX   #0
2008: BD 15 20          37  MOVECODE LDA  SELECTOR,X ;Move the new code
200B: 9D 00 D1          38          STA   SSPACE,X    ;.. to its proper place.
200E: E8                39          INX
200F: D0 F7             40          BNE   MOVECODE
                        41
2011: AD 82 C0          42          LDA   $C082       ;Write-protect BSR
                        43
2014: 60                44          RTS
                        45
                        46  SELECTOR EQU  *
                        47
                        48  * Here is the actual selector code:
                        49
                        50          ORG   $1000
                        51
1000: D8                52          CLD               ;(Required by ProDOS)
                        53
1001: 20 00 BF          54          JSR   MLI
1004: CC                55          DFB   $CC         ;CLOSE any open files
1005: AD 10             56          DA    CPARMS
                        57
1007: 20 00 BF          58          JSR   MLI
100A: C5                59          DFB   $C5         ;ONLINE for slot 6/drive 1
100B: 8B 10             60          DA    OLPARMS
100D: B0 79             61          BCS   ERROR
```

```
                        62
100F: AD 8F 10          63              LDA   NAMESIZE    ;Get returned length
1012: 29 0F             64              AND   #$0F        ;Strip slot/drive bits
1014: F0 72             65              BEQ   ERROR       ;If zero, then error
1016: 8D 8F 10          66              STA   NAMESIZE    ;Store length
                        67
                        68  * Put prefix at $281:
                        69
1019: A9 2F             70              LDA   #'/         ;Start prefix with slash
101B: 8D 81 02          71              STA   PATHNAME+1
                        72
101E: A2 00             73              LDX   #0
1020: BD 90 10          74  PUTNAME     LDA   VOLNAME,X
1023: 9D 82 02          75              STA   PATHNAME+2,X
1026: E8                76              INX
1027: EC 8F 10          77              CPX   NAMESIZE
102A: D0 F4             78              BNE   PUTNAME
                        79
102C: A9 2F             80              LDA   #'/         ;End prefix with slash
102E: 9D 82 02          81              STA   PATHNAME+2,X
1031: E8                82              INX
                        83
                        84  * ... and then tack on the BASIC.SYSTEM filename:
                        85
1032: A0 00             86              LDY   #0
1034: B9 AF 10          87  PUTSYS      LDA   SYSNAME,Y
1037: F0 07             88              BEQ   SAVELEN     ;Done if zero
1039: 9D 82 02          89              STA   PATHNAME+2,X
103C: E8                90              INX
103D: C8                91              INY
103E: D0 F4             92              BNE   PUTSYS      ;(Always taken)
                        93
```

(continued)

Table 4-4. A ProDOS selector program (continued).

```
1040: E8          94   SAVELEN  INX                ;Add 1 for initial slash
1041: 8E 80 02    95            STX  PATHNAME     ;Store length before pathname
                  96
1044: 20 00 BF    97            JSR  MLI
1047: C8          98            DFB  $C8          ;OPEN BASIC.SYSTEM
1048: 9F 10       99            DA   OPARMS
104A: B0 3C       100           BCS  ERROR
                  101
104C: AD A4 10    102           LDA  REFNUM
104F: 8D A6 10    103           STA  REFNUM1      ;Store ref # in READ table
                  104
1052: 20 00 BF    105           JSR  MLI
1055: CA          106           DFB  $CA          ;READ BASIC.SYSTEM
1056: A5 10       107           DA   RPARMS
1058: B0 2E       108           BCS  ERROR
                  109
105A: 20 00 BF    110           JSR  MLI
105D: CC          111           DFB  $CC          ;CLOSE BASIC.SYSTEM
105E: AD 10       112           DA   CPARMS
1060: B0 26       113           BCS  ERROR
                  114
                  115 * Restore I/O links to their standard values:
                  116
1062: AD 82 C0    117           LDA  $C082        ;Read-enable monitor ROM
                  118
1065: 20 89 FE    119           JSR  SETKBD
                  120
1068: 8D 0C C0    121           STA  COL80OFF     ;Want 40-column screen
106B: 20 93 FE    122           JSR  SETVID
                  123
106E: 20 58 FC    124           JSR  HOME         ;Clear the screen
```

```
                    125
                    126 * Initialize the system bit map:
                    127
1071: A9 CF         128           LDA  #$CF        ;Pages 0,1,4..7 in use
1073: 8D 58 BF      129           STA  BITMAP
                    130
1076: A9 00         131           LDA  #0
1078: A2 16         132           LDX  #22
107A: 9D 58 BF      133 INITMAP   STA  BITMAP,X    ;Pages 9..$BE free
107D: CA            134           DEX
107E: D0 FA         135           BNE  INITMAP
                    136
1080: A9 01         137           LDA  #1          ;Page $BF in use
1082: 8D 6F BF      138           STA  BITMAP+23
                    139
1085: 4C 00 20      140           JMP  SYSLOC      ;Execute BASIC.SYSTEM
                    141
1088: 4C 88 10      142 ERROR     JMP  ERROR
                    143
                    144 * ONLINE parameter table:
                    145
108B: 02            146 OLPARMS   DFB  2
108C: 60            147           DFB  $60         ;Slot 6/drive 1
108D: 8F 10         148           DA   NAMESIZE    ;Pointer to len+name
                    149
108F: 00            150 NAMESIZE  DS   1           ;Length (bits 0..3)
1090: 00 00 00      151 VOLNAME   DS   15          ;Volume name
1093: 00 00 00 00 00 00 00 00
                    152
                    153 * OPEN parameter table:
                    154
109F: 03            155 OPARMS    DFB  3
```

(continued)

Table 4-4. A ProDOS selector program (continued).

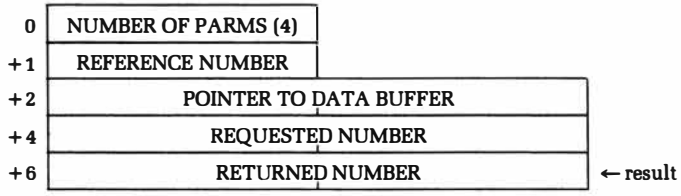
```
10A0: 80 02     156           DA   PATHNAME    ;Pointer to pathname
10A2: 00 11     157           DA   FILEBUFF
10A4: 00        158 REFNUM    DS   1           ;File reference number
                159
                160 * READ parameter table:
                161
10A5: 04        162 RPARMS    DFB  4
10A6: 00        163 REFNUM1   DS   1
10A7: 00 20     164           DA   SYSLOC      ;Start of load buffer
10A9: FF FF     165           DW   $FFFF       ;(Enough for entire file)
10AB: 00 00     166           DW   $0000
                167
                168 * CLOSE parameter table:
                169
10AD: 01        170 CPARMS    DFB  1
10AE: 00        171           DFB  0           ;All files
                172
10AF: 42 41 53  173 SYSNAME   ASC  'BASIC.SYSTEM' ;Name of system program
10B2: 49 43 2E 53 59 53 54 45
10BA: 4D
10BB: 00        174           DFB  0           ;... followed by zero
354
```

READ ($CA)

Purpose:

To read bytes of data from an open file. Reading begins at the current MARK position. After the read operation, MARK is increased by the number of bytes that were read from the file.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 4).

REFERENCE Number — This is the reference number that was assigned to the file when it was opened.

POINTER TO DATA BUFFER — This is a two-byte pointer to the beginning of a block of memory into which file data is to be read. The size of the buffer must be "requested number" characters.

REQUESTED NUMBER — This is the number of characters to be read from the file and placed in the buffer pointed to by "pointer to data buffer"

RETURNED NUMBER — This returned result contains the number of characters actually read from the file and will usually equal "requested number." However, it will be lower than "requested number" if an EOF condition was encountered or if newline read mode is active and the newline character was encountered. (See the discussion of the NEWLINE command.)

Common Error Codes:

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

$4C — The end-of-file position has been reached. **Solution:** stop reading from the file. Note that this error will be flagged only if no bytes were read from the file during

the last execution of the READ command (that is, "returned number" is 0).

$4E — The file cannot be accessed. **Solution:** set the read-enabled bit of the file's access code to 1 using SET_FILE_INFO.

$56 — The buffer address is invalid because it has been marked as "in use" in the system bit map. **Solution:** specify a buffer address that does not conflict with areas already used by ProDOS itself or by ProDOS file buffers. The system bit map can be examined to determine the free and protected areas.

Other possible error codes are: $04, $27.

Programming Example:

The following program will read up to $1000 bytes from open file #1 into the block of memory beginning at BUFFER. As usual, the reading operation begins at the current MARK position in the file. By making repeated calls to the program, further $1000-byte blocks of the file can be read.

```
            JSR MLI
            DFB $CA       ;READ
            DA  PARMTBL   ;Address of parameter
                           table
            BCS ERROR     ;Branch if error occurred
            ,
            ,
            ,
            RTS

PARMTBL     DFB 4         ;The # of parameters
            DFB 1         ;File reference number
            DA  BUFFER    ;Pointer to data buffer
            DW  $1000     ;Number of bytes to read
TRANSCNT    DS  2         ;# of bytes actually read

BUFFER      DS  $1000     ;Data buffer
```

After every call to this subroutine, you must examine the two-byte number at TRANSCNT to determine exactly how many bytes were actually read. This number may be less than $1000 if the end-of-file was reached part way through the reading operation, or if the newline character was encountered (see the discussion of the NEWLINE command for information on this character).

If error code $4C (end-of-file) is generated by the READ command, then no bytes were read and the file can be closed.

READ _ BLOCK ($80)

Purpose:

This command is used to transfer the contents of any disk block from the disk to a 512-byte buffer in memory.

Parameter Table:

0	NUMBER OF PARMS (3)
+1	UNIT NUMBER CODE
+2	POINTER TO DATA BUFFER
+4	BLOCK NUMBER

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 3).

UNIT NUMBER CODE — This byte contains the slot and drive number for the disk drive to be examined. The format of this byte is as follows:

7	6	5	4	3	2	1	0
DR	SLOT			[Unused]			

Each Apple disk controller card can have one or two drives connected to it. DR = 0 for drive 1 and DR = 1 for drive 2. SLOT contains the slot number for the disk controller card (1-7).

POINTER TO DATA BUFFER — This is a two-byte pointer to the beginning of a 512-byte block of memory that will hold the contents of the specified block after the command is executed.

BLOCK NUMBER — This is the block number to be accessed. The permitted values for "block number" depend on the disk device:

- 0-279 for the Disk][
- 0-9727 for the ProFile
- 0-127 for the /RAM volume

The volume size for any other device can be determined by using the GET _ FILE _ INFO ($C4) command on the volume directory for the disk used by the device. The size (in blocks) is returned at relative positions $5,$6 (auxiliary type code) in the parameter table.

Common Error Codes:

$27	The disk is unreadable. **Solution:** a portion of the disk may be permanently damaged. This error will also occur if the drive door is open, so make sure it is shut.
$28	The disk drive specified does not exist. **Solution:** change "unit number code" so that it refers to an existing drive.

Other possible error codes are: $04, $56.

Programming Example:

READ _ BLOCK is one of two direct disk access commands provided by ProDOS (WRITE _ BLOCK is the other). READ _ BLOCK allows you to read any block on the disk, be it a file data block, index block, directory block, or a boot record block. It doesn't matter.

It is also possible to use READ _ BLOCK to read any sector on a DOS 3.3-formatted diskette. See Appendix III for instructions on how to do this.

Here's a short program that reads the volume bit map block (block 6) on a Disk][diskette in slot 6, drive 1 into memory and then calculates the number of free blocks on the diskette.

```
            JSR MLI
            DFB $80          ;READ_BLOCK
            DA  PARMTBL      ;Address of parameter
                              table
            BCS ERROR        ;Branch if error
                              occurred

            LDA #0
            STA COUNTER
            STA COUNTER+1    ;Zero the counter

            LDY #34          ;Bit map bytes from 0-34
NEXTBYTE    LDA BLKBUFF,Y    ;Get next bit in volume
                              bit map

            LDX #8           ;8 bits to test
TESTBIT     LSR              ;Put next bit into carry
            BCC NEXTBIT      ;Branch if block not
                              free
            INC COUNTER
            BNE NEXTBIT      ;Branch if not past 255
            INC COUNTER+1    ; ... else bump high
                              part
NEXTBIT     DEX              ;Decrement bit counter
            BNE TESTBIT      ;Branch if not done
```

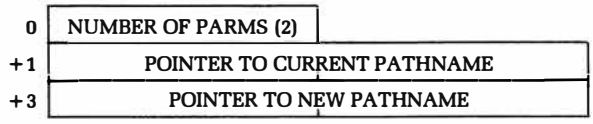
```
            DEY             ;Move to next byte
            BPL NEXTBYTE    ;Branch if not done
            RTS

PARMTBL     DFB 3           ;The # of parameters
            DFB $60         ;unit number code (slot
                             6/drive 1)
            DA  BLKBUFF     ;Pointer to 512-byte
                             buffer
            DW  6           ;Block number for volume
                             bit map

BLKBUFF     DS  512         ;This is the block
                             buffer
COUNTER     DS  2           ;# of free blocks stored
                             here
```

Recall from Chapter 2 that the first 280 bits (35 bytes) in the volume bit map act as usage flags for the 280 blocks on a standard diskette. If the bit is 1, then the block is not in use; if it is 0, then it is. This program simply scans through these 35 bytes and counts the number of "1" bits. The two-byte result is stored in COUNTER and COUNTER+1.

RENAME ($C2)

Purpose: To change the name of a specified file in a specified directory.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 2).

POINTER TO CURRENT PATHNAME — A two-byte pointer to a byte string that describes the current pathname of the file. The first byte in this string represents the length of the pathname (64 characters maximum) and is followed by the pathname itself (in ASCII). If the pathname is not preceded by a slash ("/"), then it will automatically be affixed to the currently active prefix to create a full pathname.

POINTER TO NEW PATHNAME — This is the pointer to the new pathname for the file. The format of the byte string to which it points is the same as the one to which the pointer to the current pathname points. The new pathname must be the same as old one except for the filename itself (that is, it must describe a file in the same directory). For example, you can rename a file called /FOOTBALL/CANADIAN/BC.LIONS as /FOOTBALL/CANADIAN/VANCOUVER.LIONS but not as /FOOTBALL/AMERICAN/DETROIT.LIONS.

Common Error Codes:

$2B — The disk is write-protected. **Solution:** remove the write-protect sticker from the diskette.

$40 — The pathname contains invalid characters or a full pathname was not specified (and no prefix is active). **Solution:** make sure that the filenames and directory names specified in the pathname adhere to the naming rules described in Chapter 2 and, if a partial pathname was specified, that a prefix is active.

$44	A directory in the pathname was not found. **Solution:** double-check the spelling of the pathname, insert the disk containing the correct directory, or change the active prefix.
$45	The volume directory was not found. **Solution:** double-check the spelling of the volume directory name, insert the correct disk, or change the active prefix.
$46	The file was not found. **Solution:** double-check the spelling of the filename and try again.
$47	The new filename specified already exists. **Solution:** give the file a new name not used by any other file in the same subdirectory.
$4E	The file cannot be accessed. **Solution:** set the "rename-enabled" bit of the file's access code to 1 using SET_FILE_INFO.
$50	The file is open. **Solution:** close the file first.

Other possible error codes are: $04, $27, $4A.

Programming Example:

Here is a program that will change the name of a file called BATMAN in the /SUPER.HEROES volume directory, to a file called BRUCE.WAYNE in the same directory.

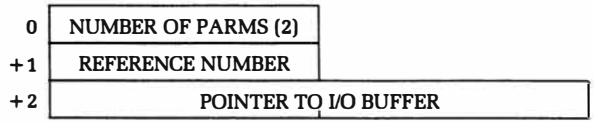
```
          JSR MLI
          DFB $C2       ;RENAME code
          DA  PARMS     ;Address of parameter table
          BCS ERROR     ;Branch if error occurred
          '
          '
          '
          RTS

PARMS     DFB 2         ;2 parameters
          DA  PATH1     ;Pointer to current
                         pathname
          DA  PATH2     ;Pointer to new pathname

PATH1     DFB 20        ;Length of pathname
          ASC '/SUPER.HEROES/BATMAN' ;Old pathname
                                      (in ASCII)

PATH2     DFB 25        ;Length of pathname
          ASC '/SUPER.HEROES/BRUCE.WAYNE'
                        ;New pathname (in ASCII)
```

Note that you cannot rename /SUPER.HEROES/BATMAN as /IDENTITIES/BRUCE.WAYNE because this violates the rule that the new file must reside in the same directory as the old one.

SET_BUF ($D2)

Purpose: To move the file buffer for an open file from its current position to another 1024-byte area.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 2).

REFERENCE NUMBER — This is the reference number assigned to the file when it was opened.

POINTER TO I/O BUFFER — A two-byte pointer to the new 1024-byte area to which the file's current buffer is to be transferred. The low-order byte of this pointer must be $00 (that is, the buffer must begin on a page boundary).

Common Error Codes:

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

$56 — The buffer address is invalid because it has been marked as "in use" in the system bit map. **Solution:** specify a buffer address that does not conflict with areas already used by ProDOS itself or by ProDOS file buffers. The system bit map can be examined to determine the free and protected areas.

Other possible error codes are: $04.

Programming Example:

The following program will move the file buffer for file #1 from its current position to $2000. You are responsible for ensuring that the area from $2000-$23FF will not be used for any other purpose.

```
JSR MLI
DFB $D2      ;SET_BUF
DA  PARMTBL  ;Address of parameter table
```

```
        BCS ERROR    ;Branch if error occurred
        ,
        ,
        ,
        RTS

PARMTBL DFB 2        ;The # of parameters
        DFB 1        ;File reference number
        DA  $2000    ;Pointer to new buffer
```

SET_EOF ($D0)

Purpose:

To change the current end-of-file pointer (EOF) of an open file. If EOF is reduced, all data blocks past the end of the new EOF will be freed up; however, if EOF is increased, new blocks for the file are not allocated until data is actually written to the new part of the file. If the new EOF is less than MARK, then MARK is set equal to the new EOF.

Parameter Table:

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 2).

REFERENCE NUMBER — This is the reference number assigned to the file when it was opened.

EOF POSITION — This is a three-byte value that holds the new EOF position.

Common Error Codes:

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

$4E — The file cannot be accessed. **Solution:** set the "write-enabled" bit of the file's access code to 1 using SET_FILE_INFO.

Other possible error codes are: $04, $27.

Programming Example:

Consider a situation in which you must read an entire file into memory, modify it, and then write it back to the same file. Unless you exercise some care, if the new file is smaller than the original, the tail end of the old file (the part that is not overwritten) will unexpectedly remain as part of the new file.

To avoid this you can do one of two things: delete the file before rewriting it or write to the file and then use the SET_EOF command to fix the new EOF position. The second method is faster and more convenient because it is not necessary to go to the trouble of first deleting, then re-creating, a file.

Suppose the new file length is $1534 bytes. To set the EOF for this file you would call a subroutine such as this:

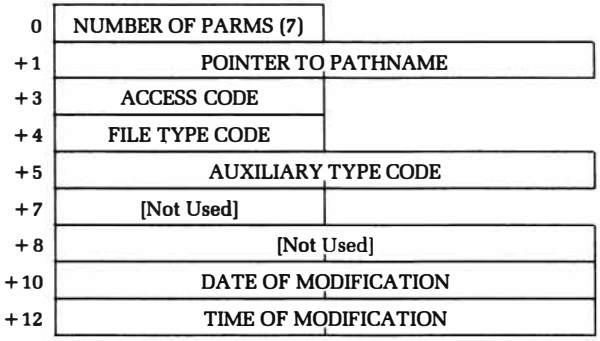
```
            LDA #$34
            STA POSITION
            LDA #$15
            STA POSITION+1
            LDA #0
            STA POSITION+2
            JSR MLI
            DFB $D0          ;SET_EOF
            DA  PARMTBL      ;Address of parameter
                              table
            BCS ERROR        ;Branch if error
                              occurred
            ,
            ,
            ,
            RTS

PARMTBL     DFB 2            ;The # of parameters
            DFB 1            ;File reference number
POSITION    DS  3            ;New EOF position
```

SET_FILE_INFO ($C3)

Purpose:

To modify the information that is stored in a file's directory entry. This includes the file's access code, file type code, auxiliary type code, and the date and time at which the file was last modified.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 7).

POINTER TO PATHNAME — A two-byte pointer to a byte string that describes the pathname of the file to be used. The first byte in this string represents the length of the pathname (64 characters maximum) and is followed by the pathname itself (in ASCII). If the pathname is not preceded by a slash ("/"), then ProDOS will automatically append it to the currently active prefix to create a full pathname.

ACCESS CODE — This byte contains five one-bit codes that define the access attributes of the file to be created. See Figure 4-2 for a description of these bits. The "backup-needed" bit of the access code is automatically set to 1 by this command.

FILE TYPE CODE — This byte contains a code that indicates the type of data the file holds. See Table 4-3 for a description of the ProDOS file type codes.

AUXILIARY TYPE CODE — These two bytes are used for special purposes by different interpreters (system programs). For example, BASIC.SYSTEM stores the default loading address

	here for SYS files ($2000) and BIN files, the field length for TXT files, the loading address for BAS files (usually $801), or the starting address of a block of variables for VAR files.
[Not Used]	These bytes are not used. They act as padding to preserve symmetry between this parameter list and the GET _ FILE _ INFO parameter list.
DATE OF MODIFICATION	These two bytes contain the date (year, month, day) on which the file was last modified. The current date should be stored here before executing the command. The format of these bytes is as shown in Figure 8-1.
TIME OF MODIFICATION	These two bytes contain the time (hour, minute) at which the file was last modified. The current time should be stored here before executing the command. The format of these bytes is as shown in Figure 8-1.

Common Error Codes:

$2B	The disk is write-protected. **Solution:** remove the write-protect sticker from the diskette.
$40	The pathname contains invalid characters or a full pathname was not specified (and no prefix is active). **Solution:** make sure that the filenames and directory names specified in the pathname adhere to the naming rules described in Chapter 2 and, if a partial pathname was specified, that a prefix is active.
$44	A directory in the pathname was not found. **Solution:** double-check the spelling of the pathname, insert the disk containing the correct directory, or change the active prefix.
$45	The volume directory was not found. **Solution:** double-check the spelling of the volume directory name, insert the correct disk, or change the active prefix.
$46	The file was not found. **Solution:** double-check the spelling of the filename and try again.
$4E	The access code specified for the file is not permitted. **Solution:** ensure that the reserved bits of the access code (bits 2, 3, and 4) are all zero.

Other possible error codes are: $04, $27.

Programming Example:

The following program will lock a file called PRISONER by changing the value of its access code byte. It is assumed that PRISONER is located in the currently active directory (the one specified by the prefix).

```
            LDA  #10
            STA  PARMTBL      ;Store # of parms for
                               GET_FILE_INFO
            JSR  MLI
            DFB  $C4          ;GET_FILE_INFO (see next
                               section)
            DA   PARMTBL      ;Address of parameter
                               table
            BCS  ERROR        ;Branch if error
                               occurred

            LDA  PARMTBL+3    ;Get current access
                               code
            AND  #$3D         ;Clear bits 1, 6, and 7
                               (write, rename, and
                               destroy bits)
            STA  PARMTBL+3    ;Store new access code
            LDA  #7
            STA  PARMTBL      ;Store # of parms for
                               SET_FILE_INFO
            JSR  MLI          ;Save new access code
                               to disk
            DFB  $C3          ;SET_FILE_INFO
            DA   PARMTBL      ;Address of parameter
                               table
            BCS  ERROR        ;Branch if error
                               occurred
            ,
            ,
            ,
            RTS

PARMTBL     DS   1            ;The # of parameters is
                               stored here
            DA   PATHNAME
            DS   1            ;access code
            DS   1            ;file type code
            DS   2            ;auxiliary type code
            DS   1            ;storage type code
            DS   2            ;blocks used
            DS   2            ;date of modification
            DS   2            ;time of modification
            DS   2            ;date of creation
            DS   2            ;time of creation
PATHNAME    DFB  8            ;Length of pathname
            ASC  'PRISONER'   ;Pathname (in ASCII)
```

There are two interesting things to note about this program. First, it uses the GET_FILE_INFO ($C4) command to read the file's current access code and other directory information. Since the parameter table for this command and for the SET_FILE_INFO command are symmetric, there is no need to create two tables; all that has to be done is store the proper parameter number at the head of the table before calling each command.

Second, notice how the file is locked. The existing access code is logically ANDed with $3D (binary 00111101) to clear bits 1, 6, and 7 to zero while leaving the others unaffected. As indicated in Figure 4-2, clearing these bits will disable write, rename, and destroy operations, respectively.

SET_MARK ($CE)

Purpose:

To change the current position-in-file pointer (MARK) of an open file. MARK can be set to any position within the file; subsequent read or write operations will take place at that position.

Parameter Table:

Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 2).

REFERENCE NUMBER — This is the reference number assigned to the file when it was opened.

MARK POSITION — This is a three-byte value that holds the new MARK position. This position must not exceed the EOF position for the file.

Common Error Codes:

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

$4D — The MARK position is larger than the EOF position. **Solution:** specify a MARK position that is less than or equal to the EOF position.

Other possible error codes are: $04, $27.

Programming Example:

Suppose you have created a large textfile in which information is arranged in 98-byte records and you want to directly access the 23rd such record. The easiest way to do this is to move the MARK pointer directly to the start of this record and then use the READ or WRITE command.

You can determine the proper value for MARK by multiplying the record length (98) by the record number (23); the result is 2254 (or $08CE). Here's how to move MARK to this position (assume that the file is open and has a reference number of 1):

```
            LDA #$CE
            STA POSITION
            LDA #$00
            STA POSITION+1
            LDA #0
            STA POSITION+2 ;(high-order byte is
                            zero)

            JSR MLI
            DFB $CE         ;SET_MARK
            DA  PARMTBL     ;Address of parameter
                            table
            BCS ERROR       ;Branch if error
                            occurred
            ,
            ,
            ,
            RTS

PARMTBL     DFB 2           ;The # of parameters
            DFB 1           ;File reference number
POSITION    DS  3           ;New MARK position
```

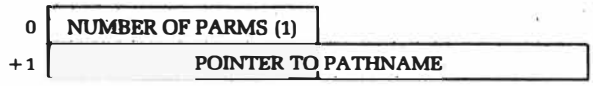
Remember that the MARK position is a three-byte quantity and that it cannot exceed the EOF position.

SET_PREFIX ($C6)

Purpose:

To set the current prefix to a specified directory path. When partial pathnames are specified in other ProDOS calls, they are automatically converted to full pathnames by appending them to the active prefix.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 1).

POINTER TO PATHNAME — A two-byte pointer to a byte string that describes the pathname of the prefix. The first byte in this string represents the length of the pathname (64 characters maximum) and is followed by the pathname itself (in ASCII). If the pathname is not preceded by a slash ("/"), then it will automatically be affixed to the currently active prefix to create the new prefix. An optional slash may be placed at the end of the pathname.

Common Error Codes:

$40 — The pathname contains invalid characters or a full pathname was not specified (and no prefix is active). **Solution:** make sure that the filenames and directory names specified in the pathname adhere to the naming rules described in Chapter 2 and, if a partial pathname was specified, that a prefix is active.

$44 — A directory in the pathname was not found. **Solution:** double-check the spelling of the pathname, insert the disk containing the correct directory, or change the active prefix.

$45 — The volume directory was not found. **Solution:** double-check the spelling of the volume directory name, insert the correct disk, or change the active prefix.

$46 — The file was not found. **Solution:** double-check the spelling of the filename and try again.

$4B	A non-directory name was specified in the prefix string. **Solution:** try again with a prefix string that contains only directory names.

Other possible error codes are: $04, $27.

Programming Example:

It is often convenient to be able to set the ProDOS prefix to the name of the volume directory on a disk in a specific disk drive. If this is done, all files in the volume directory can be referred to by filename alone, rather than by full pathname.

This can be done in two simple steps: first, use the ON_LINE command to determine the volume name for that disk and, second, use SET_PREFIX to assign that name to the prefix. One complication does arise, however: the name returned by ON_LINE is not quite in the format required by SET_PREFIX. Fortunately, we can easily overcome this deficiency.

```
            JSR MLI
            DFB $C5         ;ON_LINE
            DA  PARMTBL     ;Address of parameter
                             table
            BCS ERROR       ;Branch if error occurred
            LDA BUFFER      ;Get length byte
            AND #$0F        ;Strip off slot/drive
                             bits
            STA PFXNAME     ;Store length for
                             SET_PREFIX
            LDA #'/'
            STA BUFFER      ;Put slash in front of
                             volume name
            JSR MLI
            DFB $C6         ;SET_PREFIX
            DA  PARMTBL1
            BCS ERROR1      ;Branch if error occurred
            ,
            ,
            ,
            RTS

PARMTBL     DFB 2           ;The # of parameters
            DFB $E0         ;unit number = slot 6,
                             drive 2
            DA  BUFFER      ;Pointer to 16-byte
                             buffer

PARMTBL1    DFB 1           ;The # of parameters
            DA  PFXNAME     ;Pointer to volume name
```

```
PFXNAME   DS   0          ;Length of name for
                           SET_PREFIX
BUFFER    DS   1          ;Slot/drive (bits 4-7)
                           and length of volume
                           name (bits 0-3)
          DS   15         ;Volume name (in ASCII)
```

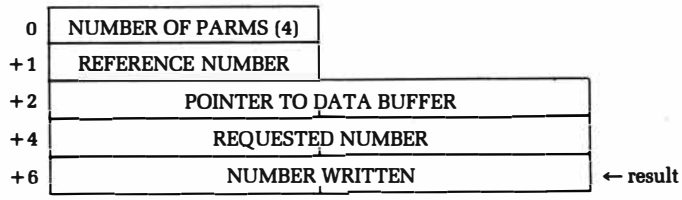
The ON _ LINE command returns a volume name that is NOT preceded by the slash required by SET _ PREFIX. This problem is fixed by reading the name length byte returned by SET _ PREFIX, storing it at the previous memory location (PFXNAME), and then overwriting the name length byte with the slash. After this has done, the data structure beginning with PFXNAME is in the format required by SET _ PREFIX.

WRITE ($CB)

Purpose:

To write bytes of data to an open file. Writing begins at the current MARK position. After the data is written to the file, the MARK position is increased by the number of bytes written. If the new MARK position is greater than EOF, then EOF is set equal to MARK.

Parameter Table:



Description of Parameters:

NUMBER OF PARMS — The number of parameters for this command (always 4).

REFERENCE NUMBER — This is the reference number assigned to the file when it was opened.

POINTER TO DATA BUFFER — This is a two-byte pointer to the beginning of a block of memory that contains the data to be written to the file.

REQUESTED NUMBER — This is the number of characters to be written to the file from the buffer pointed to by "pointer to data buffer."

NUMBER WRITTEN — This returned result contains the number of characters actually written to the file and will usually equal "requested number." However, it will be less than "requested number" if the disk becomes full part way through a write operation or if some other disk error occurs that prevents the file from being written to.

Common Error Codes:

$2B — The disk is write-protected. **Solution:** remove the write-protect sticker from the diskette.

$43 — The file reference number is invalid. **Solution:** only use reference numbers for files that are still open.

$48	The disk is full. **Solution:** create the file on another disk or delete a file from the current disk and try again.
$4E	The file cannot be accessed. **Solution:** set the "write-enabled" bit of the file's access code to 1 using SET_FILE_INFO.
$56	The buffer address is invalid because it has been marked as "in use" in the system bit map. **Solution:** specify a buffer address that does not conflict with areas already used by ProDOS itself or by ProDOS file buffers. The system bit map can be examined to determine the free and protected areas.

Other possible error codes are: $04, $27.

Programming Example:

This program writes 256 bytes to file #2. The data buffer begins at location BUFFER.

```
            JSR MLI
            DFB $CB       ;WRITE
            DA  PARMTBL   ;Address of parameter
                           table
            BCS ERROR     ;Branch if error occurred
            ,
            ,
            ,
            RTS

PARMTBL     DFB 4         ;The # of parameters
            DFB 2         ;File reference number
                           (assume #2)
            DA  BUFFER    ;Pointer to data buffer
            DW  256       ;Number of bytes to write
TRANSCNT    DS  2         ;# of bytes actually
                           written

BUFFER      DS  256       ;Data buffer
```

If no error occurred, the number stored at TRANSCNT should be equal to 256, the "requested number." If the disk becomes full during the write, however, TRANSCNT will be less than 256.

WRITE _ BLOCK ($81)

Purpose:

To transfer the contents of a 512-byte buffer from memory to a block on the disk.

Parameter Table:

0	NUMBER OF PARMS (3)
+1	UNIT NUMBER CODE
+2	POINTER TO DATA BUFFER
+4	BLOCK NUMBER

Description of Parameters:

NUMBER OF PARMS

The number of parameters for this command (always 3).

UNIT NUMBER CODE

This byte contains the slot and drive number for the disk drive to be written to. The format of this byte is as follows:

7	6	5	4	3	2	1	0
DR	SLOT			[Unused]			

Each Apple disk controller card can have one or two drives connected to it. DR = 0 for drive 1 and DR = 1 for drive 2. SLOT contains the slot number for the disk controller card (1-7).

POINTER TO DATA BUFFER

This is a two-byte pointer to the beginning of a 512-byte block of memory that is to be written to the disk.

BLOCK NUMBER

This is the block number to be accessed. The permitted values for "block number" depend on the disk device:

- 0-279 for the Disk][
- 0-9727 for the ProFile
- 0-127 for the /RAM volume

The volume size for any other device can be determined by using the GET _ FILE _ INFO ($C4) command on the volume directory for the disk used by the device. The size (in blocks) is returned at relative positions $5,$6 (auxiliary type code) in the parameter table.

Common Error Codes:

$27	The disk is unwriteable. **Solution:** a portion of the disk may be permanently damaged. This error will also occur if the drive door is open, so make sure it is shut.
$28	The disk drive specified does not exist. **Solution:** change "unit number code" so that it refers to an existing drive.
$2B	The disk is write-protected. **Solution:** remove the write-protect sticker from the diskette.

Other possible error codes are: $04, $56.

Programming Example:

WRITE_BLOCK is probably the most dangerous of all the ProDOS commands since it allows you to overwrite any block on the disk with any data you want. It is very useful, however, for trying to recover damaged files and for making backup copies of disks.

It is also possible to use WRITE_BLOCK to write to any sector on a DOS 3.3-formatted diskette. See Appendix III for instructions on how to do this.

Here's an interesting program that allows you to rename the volume directory of a disk in slot 6, drive 1 to AREA:

```
            JSR MLI
            DFB $80          ;READ_BLOCK
            DA  PARMTBL      ;Address of parameter
                              table
            BCS ERROR        ;Branch if error
                              occurred

            LDX #0
            LDY #5           ;Offset for volume name
MOVENAME    LDA NEWNAME,X
            BEQ SETLEN       ;Branch if at end
            STA BLKBUFF,Y    ;Move new name into
                              place
            INX
            INY
            BNE MOVENAME     ;(Always taken)
SETLEN      TXA              ;Get new name length
            ORA #$F0         ;Merge directory ID bits
            LDY #4
            STA BLKBUFF,Y    ;Save new name length

            JSR MLI
            DFB $81          ;WRITE_BLOCK
```

```
            DA   PARMTBL     ;Address of parameter
                              table
            BCS  ERROR       ;Branch if error
                              occurred
            RTS

PARMTBL     DFB  3           ;The # of parameters
            DFB  $60         ;unit number code (slot
                              6/drive 1)
            DA   BLKBUFF     ;Pointer to 512-byte
                              buffer
            DW   2           ;Block number for volume
                              directory

BLKBUFF     DS   512         ;This is the block
                              buffer

NEWNAME     ASC  'AREA'      ;New volume name
                              (<=15 chars)
            DFB  0           ;(Terminate with 0)
```

We saw in Chapter 2 that the volume directory of a disk always begins in block 2 and that the volume name is the first entry in that directory block (beginning at offset 5). This program simply reads in block 2 (using READ _ BLOCK), changes the volume name, and then writes the block back to diskette. The chore is simplified by the fact that the parameter tables for READ _ BLOCK and WRITE _ BLOCK are identical.

Chapter 5

SYSTEM PROGRAMS FEATURING BASIC.SYSTEM

Generally speaking, a system program is one that creates and defines a general-purpose environment in which other, more specific programs, called applications programs, can be created and, perhaps, executed in a ProDOS environment. It is primarily responsible for translating user-entered commands (that it defines and interprets) into the appropriate MLI commands and for properly managing system memory usage. System programs are very important to ProDOS because, as we saw in Chapter 3, ProDOS cannot operate without one installed. (ProDOS will hang if a system program having a name of the form "xxxxxxxx.SYSTEM" is not present in the volume directory of the disk.)

Common system programs are high-level programming language interpreters (for BASIC, Pascal, C, and so on), assemblers, or even word processors. (In the latter case, the applications program is simply the document that is created in the word processing environment.) Such system programs usually define a set of disk commands that can be used to communicate easily with files on a disk device. This serves to shield the user from the low-level MLI commands that only programmers should have to worry about.

From the point of view of ProDOS, however, the description of what constitutes a system program is much more specific: any file whose file type code is $FF (this code is stored at relative position $10 in the directory entry for the file). Such a program has a type mnemonic of SYS that appears when the disk

is cataloged in a BASIC.SYSTEM environment. It is the duty of the programmer who created the system program to anoint it with the properties described in the previous paragraph.

In this chapter we will describe the features of a well-designed system program (we'll also call such a program an "interpreter") and then describe one such program in particular, BASIC.SYSTEM. The discussion of BASIC.SYSTEM will be quite detailed; we will see how it installs itself in the system, how it makes calls to the MLI, how its command set can be extended, and how it handles errors. We will also take a close look at the *global page* that it uses to communicate with ProDOS and with applications programs.

The Structure of a System Program

A well-designed system program is an executable 6502 assembly language program that adheres to certain conventions and protocols regulating its internal structure and its performance characteristics.

Structure

By convention, a system program is to be loaded into memory beginning at location $2000, and is executed by performing a JMP $2000 instruction. This means that the code must be assembled for execution at this, and no other, position. Note, however, that the system program can later relocate itself anywhere within the RAM space from $800–$BEFF that is reserved for its use.

You can use the BASIC.SYSTEM "-" command to execute a system program. It is also possible to designate a system program that is to be automatically executed when ProDOS is first booted; this is done by giving the program a name of the form "xxxxxxxx.SYSTEM" and ensuring that it is the first entry in the volume directory that has such a name.

To create a system program you will probably want to use an assembler. However, most assemblers create object code binary (BIN) files rather than system (SYS) files. Fortunately, it is fairly easy to change the file type. First, use the CATALOG command to display the size of the binary file to be converted. Next, use the BASIC.SYSTEM BLOAD command to load the file into memory beginning at $2000,

```
BLOAD FILENAME,A$2000
```

then delete the file,

```
DELETE FILENAME
```

and save the "new" file back to the disk using the BSAVE command with a "TSYS" parameter suffix,

```
BSAVE FILENAME,A$2000,L$xxxx,TSYS
```

where "xxxx" represents the hexadecimal size of the program.

Some system programs follow a special *auto-run* protocol that has been defined to allow a ProDOS *selector* program to install the name of an applications program that is to be automatically executed as soon as the system program takes over. (Recall from Chapter 4 that a selector program gets control when the MLI QUIT ($65) command is executed.) The standard ProDOS selector program does not actually allow you to do this, but it could be an option in any alternative selector that you may design to replace it. The description of the QUIT ($65) command in Chapter 4 includes instructions on how to write such a selector.

The protocol is quite simple. If the first byte of the system program ($2000) is $4C (a JMP opcode) and the fourth and fifth bytes ($2003 and $2004) are both $EE, then the sixth byte ($2005) holds the size of a buffer that starts at the very next byte. This buffer begins with a name length byte and is followed by the standard ASCII codes for the characters in the name of the first applications program to be run. (The system program will usually store a default filename here.) Thus, if the selector program detects the presence of the three identification bytes, it could prompt you to enter the name of an applications program, load the system program, store the length and name of the applications program beginning at $2006, and then execute the system program by jumping to $2000.

The BASIC.SYSTEM system program adheres to this protocol. Here is what the beginning of that program looks like:

```
        JMP  START1     ;Must be a JMP instruction
        DFB  $EE        ;Identification byte 1
        DFB  $EE        ;Identification byte 2
        DFB  $41        ;Size of following buffer
        DFB  $07        ;Length of filename
        ASC  'STARTUP'  ;Name of auto-run program
        '
        '
START1                  ;Main program entry point
```

As you can see, BASIC.SYSTEM defines a default auto-run program called STARTUP. This program will be loaded and run whenever BASIC.SYSTEM is executed, unless the selector inserts a different filename.

The selector program ensures that when a system program gets control, its pathname or partial pathname is stored at $281; location $280 contains the length of the name. This permits the system program to deduce the precise directory in which it is located so that it can load any subsidiary programs that are located in the same directory.

Performance

In many cases, a system program will define an interpretive programming environment in which applications programs can be written and executed

(BASIC.SYSTEM is the best example of such a program). If this is the case, the code for the interpreter should be tucked away in a safe place that will not conflict with memory areas that can be used by the applications program. The best position for the code is in a contiguous block at the upper end of main RAM memory, just below the ProDOS global page at $BF00; this leaves the space from $800 to the start of the code free for use by the applications program. The code space can be protected by setting to 1 those bits in the system bit map that correspond to the pages used by the system program. If this is done, the ProDOS MLI interpreter will not permit these areas to be used as file buffers or I/O buffers. (See Chapter 3 for a discussion of the system bit map.)

When a system program first gets control, it should do several preliminary housekeeping chores:

- Initialize the 6502 stack pointer. To ensure that the maximum amount of stack space is available to the system program, the stack pointer should be set to the bottom of the stack. This can be done as follows:

```
LDX #$FF
TXS
```

  You should ensure that no more than ¾ of the stack is used at any given time.

- Initialize the reset vector. When reset is pressed on an Apple II, control will pass to the subroutine whose address is stored in the reset vector at SOFTEV ($3F2/$3F3) if the number stored at PWREDUP ($3F4) is the same as the number generated by logically exclusive-ORing the number stored at SOFTEV+1 with the constant $A5. If PWREDUP is not set up properly, the system will reboot when reset is pressed. To point the reset vector to a subroutine called RTRAP within the system program and fix up PWREDUP, execute the following code:

```
LDA #<RTRAP   ;Address low
STA SOFTEV    ;$3F2
LDA #>RTRAP   ;Address high
STA SOFTEV+1  ;$3F3
EOR #$A5      ;twiddle the bits
STA PWREDUP   ;$3F4
```

  A general-purpose RTRAP subroutine should close all files and then jump to the cold start entry point of the system program. To do anything else may not be safe because it is impossible for the reset subroutine to determine the state of the system just before the reset condition becomes active.

- Initialize the version numbers in the ProDOS global page. IBAKVER ($BFFC) must be set equal to the earliest version of ProDOS with which the system program will work; store 0 here if any version will do. IVERSION must be set equal to the version number of the system program being used. For example, here is the code needed if version 3 of the

system program is being used and it will only work with version 1 (or higher) of ProDOS:

```
LDA #1
STA IBAKVER   ;($BFFC)
LDA #3        ;
STA IVERSION  ;($BFFD)
```

See Appendix II for the internal version numbers used by ProDOS and BASIC.SYSTEM.

When these chores have been completed, the system program can begin its main duties. If the system program adheres to the auto-run protocol described above, then it must execute the program whose name (preceded by a length byte) is stored beginning at $2005. After that's done, the system program is free to do almost anything it wants as long as it does not overwrite the ProDOS system global page (page $BF) or data areas in other pages that are used by ProDOS or any system monitor subroutines that may be used by the system program. (See Chapter 3 for a discussion of ProDOS memory usage.)

If a system program wants to create special classes of files, it can use any of the user-definable file type codes, $F1–$F8. All other codes are reserved. See Table 4-3 in Chapter 4 for a description of how the other file type codes are used by ProDOS and other system programs released by Apple.

When a system program creates a file, it can store a two-byte auxiliary type code in its directory entry (at relative bytes $1F and $20; see Chapter 2) that holds miscellaneous information about the file. This code is assigned when the file is first created using the MLI CREATE ($C0) command. For example, here is the meaning of the auxiliary type code for each type of file used by BASIC.SYSTEM:

```
BIN  default loading address
SYS  default loading address ($2000)
TXT  record length (0 for sequential files)
BAS  default loading address (usually $0801)
VAR  starting address of a block of variables
```

The meaning of the auxiliary type codes for files created by other system programs is completely determined by them.

When the time comes to leave the system program, the PWREDUP byte should first be scrambled by decrementing it; this will cause the system to reboot if reset is pressed. All files should then be closed and /RAM should be reconnected if it was earlier disconnected (see Chapter 7 for instructions on how to do this). Finally, control should be passed to another system program by using the MLI QUIT ($65) command. As we saw in Chapter 4, this will cause the standard ProDOS selector program to be executed. Here is what the code will look like:

```
,
[close files]
[restore /RAM]
```

```
          DEC  $3F4          ;Scramble PWRDUP byte
          JSR  $BF00         ;Call the MLI
          DFB  $65           ;QUIT
          DA PARMTBL
          BCS ERROR
          RTS
PARMTBL   DFB  4             ;4 parameters
          DFB  0
          DA   0
          DFB  0
          DA   0
```

The selector code is responsible for passing control to another system program in an orderly manner. The standard ProDOS selector will ask you to enter the prefix and pathname of the system program that is to be loaded and executed.

The BASIC.SYSTEM Program

The most commonly used ProDOS system program is probably the BASIC.SYSTEM interpreter. This program is loaded whenever an Applesoft programming environment is going to be used because it extends the Applesoft command set by installing a group of 32 disk commands that are available for use within an Applesoft program (many can also be used from Applesoft direct mode). BASIC.SYSTEM installs itself by storing the addresses of its internal character input and output subroutines in the system monitor's input link (KSW: $38/$39) and output link (CSW: $36/$37). (The subroutines whose addresses are stored in these links are called whenever a character input or output operation is to be performed.)

The BASIC.SYSTEM input subroutine normally reads input from the current input device (usually the keyboard) and will identify and execute any valid disk commands that are entered while in direct mode. If a file has previously been opened for read operations, however, it will get its input from the file instead.

Similarly, the BASIC.SYSTEM output subroutine normally sends output to the current output device (usually the video screen) unless a file has been opened to receive the output instead. It is also always on the lookout for arguments of PRINT statements that begin with a [control-D] character; such arguments are assumed to be BASIC.SYSTEM disk commands and attempts will be made to interpret them as such. The output subroutine can spot such PRINT statements because BASIC.SYSTEM always operates with Applesoft trace mode on; this means that line numbers will be sent to the output subroutine before the line is actually executed, thus giving BASIC.SYSTEM a chance to check any PRINT statements on that line. (By the way, the line numbers generated in trace mode will not be displayed by BASIC.SYSTEM unless the Applesoft TRACE command has been executed.)

A BASIC.SYSTEM memory map is shown in Figure 5-1. When BASIC.SYSTEM is first loaded, it relocates its command interpreter to the high end of main RAM memory at $9A00–$BEFF (just below the ProDOS system global page), it reserves a 1K general-purpose file buffer from $9600–$99FF, and then it sets the Applesoft HIMEM pointer at $73/$74 to $9600. (HIMEM represents the upper limit for storage of Applesoft string variables.) This leaves the space from $800–$95FF free for Applesoft program and variable storage.

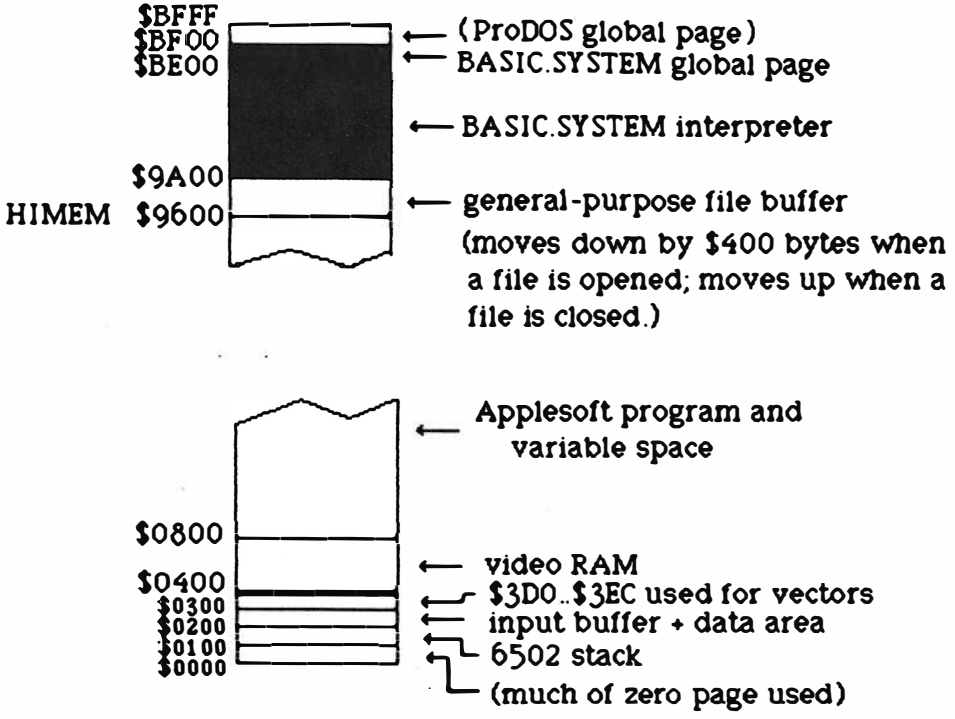


Figure 5-1. BASIC.SYSTEM memory map

BASIC.SYSTEM also uses the area between $3D0–$3EC for storage of position-independent vectors to some of its internal subroutines. We'll be examining the way in which BASIC.SYSTEM uses page 3 in more detail later on in this chapter.

The BASIC.SYSTEM interpreter, because of its intimate connection to the Applesoft ROM interpreter, can also be said to use all those RAM areas used by Applesoft itself. This includes the input buffer at $200–$2FF (BASIC.SYSTEM also uses most of this page as a temporary data buffer when it executes certain disk commands), the 6502 stack at $100–$1FF, and several locations in zero page. (See Chapter 4 of *Inside the Apple //e* for a detailed description of how Applesoft uses these areas.) Other areas, such as the video RAM area from $400–$7FF and the system vector area from $3ED–$3FF, are also reserved for use in a BASIC.SYSTEM environment.

The BASIC.SYSTEM Commands

Most of the BASIC.SYSTEM disk commands provide convenient access to files for I/O operations (OPEN, READ, POSITION, WRITE, APPEND, FLUSH, and CLOSE), or general file management (CAT, CATALOG, CREATE, DELETE, LOCK, PREFIX, RENAME, UNLOCK, and VERIFY), or program file loading and execution (-, BLOAD, BRUN, BSAVE, EXEC, LOAD, RUN, SAVE). There are also commands used to effect I/O redirection (IN#, PR#), to perform garbage collection of Applesoft string variables (FRE), to save and load Applesoft variables to and from files (STORE and RESTORE), to transfer control from one Applesoft program to another without destroying existing variables (CHAIN), and to disconnect BASIC.SYSTEM and run another ProDOS system program (BYE). One command (NOMON) is allowed but ignored; it is included to maintain compatability with programs running under DOS 3.3 that use NOMON to disable the display of disk commands and I/O operations.

To use a BASIC.SYSTEM command from within a program, you must use the PRINT statement to print a [control-D] character, the BASIC.SYSTEM command, the command parameters, and then a carriage return. For example, to list all the files in the /RAM volume on an Apple //c, you would execute a line that looks something like this:

```
100   PRINT CHR$(4);"CATALOG/RAM"
```

In this example, the CHR$(4) statement generates the [control-D] character, the BASIC.SYSTEM command is CATALOG, and the command parameter is /RAM (a pathname). The required carriage return is automatically generated by the PRINT statement.

If you're entering a BASIC.SYSTEM command directly from the keyboard in Applesoft direct mode, then you don't have to worry about the [control-D]. All that you have to do is type in the command followed by the command arguments. The keyboard equivalent of the CATALOG command is simply

```
CATALOG/RAM
```

You should be aware, however, that BASIC.SYSTEM does not permit all of its commands to be entered from the keyboard in this way.

Most BASIC.SYSTEM commands support, or require, several parameters that specify such things as the pathname for the file to be acted on, loading addresses, lengths, and so on. A brief description of each of the 13 different parameters recognized by BASIC.SYSTEM is given in Table 5-1.

The letter parameters shown in Table 5-1 (",A#", ",B#", and so on, where "#" represents the value of a parameter) can be specified in any order by appending them to the end of the command line. The "snum" and "pathname" parameters cannot appear in the same command line. When one of these parameters is specified, it must be placed immediately after the command name. The exception is the RENAME command that requires two pathnames; the second pathname must appear right after the first one.

Table 5-1. BASIC.SYSTEM command line parameters.

Parameter	*Standard Meaning*	*Permitted Values*	
pathname	the active file	See rules in Chapter 2	
snum	active I/O slot	0-7	(1)
,A#	starting address	$0000-$FFFF	
,B#	byte number	$0000-$FFFFF	
,D#	disk drive number	1-2	(2)
,E#	ending address	$0000-$FFFF	
,F#	field number	$0000-$FFFF	
,L#	length	$0000-$FFFF	
,@#	line number	$0000-$FFFF	
,R#	record number	$0000-$FFFF	
,S#	disk slot number	1-7	(2)
,T#	file type code	$00-$FF	(3)
,V#	volume number	$00-$FF	

The "#" in the parameter name represents the parameter's value.
The value can be specified in hexadecimal or decimal format (hexadecimal numbers must be preceded by "$").

Notes:

1. Hexadecimal values are not allowed for "snum".
2. In a command line that includes a pathname, the S and D parameters specified must correspond to an installed disk drive, or a "NO DEVICE CONNECTED" error will be generated.
3. A three-character file type mnemonic corresponding to a value can be specified with the "T" parameter instead. The mnemonics that are available are shown in Table 2-4 in Chapter 2.

Note that most BASIC.SYSTEM commands may be entered with slot (",S#") and drive (",D#") parameters that specify the physical location of the disk to be accessed. It is not necessary to use these parameters if the pathname specified is a full pathname or if a prefix is active, because BASIC.SYSTEM will automatically search all installed disk drives for the file. However, if a filename or partial pathname is specified, and no prefix has yet been defined or either the ",S#" or ",D#" parameter is used, BASIC.SYSTEM will automatically use the name of the volume directory specified by the slot and drive parameters (or their defaults) to create the full pathname. BASIC.SYSTEM's ability to use slot and drive parameters allows Applesoft programs to maintain greater compatibility with a DOS 3.3 environment where the slot and drive must be specified to access disks in the non- default drive.

Let's take a quick look at each of the 32 BASIC.SYSTEM commands right now. A summary of the command syntax for each of these commands is shown in Table 5-2. (You can refer to Apple's book, *BASIC Programming with ProDOS*, for detailed information on these commands; see the references in Appendix IV.) These commands can be divided into four distinct categories: file management commands, file loading and execution commands, file input/output commands, and miscellaneous commands.

Table 5-2. The syntax for each BASIC.SYSTEM command.

_	pathname [,S#] [,D#]
APPEND	pathname [,Ttype] [,L#] [,S#] [,D#]
BLOAD	pathname [,A#] [,B#] [,L# | ,E#] [,Ttype] [,S#] [,D#]
BRUN	pathname [,A#] [,B#] [,L# | ,E#] [,S#] [,D#]
BSAVE	pathname,A# ,L# | ,E# [,B#] [,Ttype] [,S#] [,D#]
BYE	
CAT	[pathname] [,S#] [,D#]
CATALOG	[pathname] [,S#] [,D#]
CHAIN	pathname [,@#] [,S#] [,D#]
CLOSE	[pathname]
CREATE	pathname [,Ttype] [,S#] [,D#]
DELETE	pathname [,S#] [,D#]
EXEC	pathname [,F# | ,R#] [,S#][,D#]
FLUSH	[pathname]
FRE	
IN#	snum | A# | snum,A#
LOAD	pathname [,S#] [,D#]
LOCK	pathname [,S#] [,D#]
NOMON	[anything]
OPEN	pathname [,L#] [,Ttype] [,S#] [,D#]
POSITION	pathname ,F# | ,R#
PR#	snum | A# | snum,A#
PREFIX	[pathname] [,S#] [,D#]
READ	pathname [,R#] [,F#] [,B#]
RENAME	pathname1,pathname2 [,S#] [,D#]
RESTORE	pathname [,S#] [,D#]
RUN	pathname [,@#] [,S#] [,D#]
SAVE	pathname [,S#] [,D#]
STORE	pathname [,S#] [,D#]
UNLOCK	pathname [,S#] [,D#]
VERIFY	[pathname] [,S#] [,D#]
WRITE	pathname [,R#] [,F#] [,B#]

Note: Brackets enclose optional parameters.
Vertical bars separate alternative parameters.

File Management Commands

CAT. This command is used to display a list of the names of the files on the disk. Only the names of the files in the directory specified in the pathname parameter following the CAT command will be displayed. (If no such parameter is specified, then the currently active directory will be used.) CAT will also display the type of each file (as a three-character mnemonic such as BAS, BIN, TXT, SYS, and so on; see Table 2-4), the number of blocks it occupies, and the date on which it was last modified. After the names of all files have been listed, the number of blocks free and blocks used on the disk will be displayed.

CATALOG. This command is very similar to CAT. It displays the very same information for each file as well as its time of last modification, its creation date

and time, its size (in bytes), and its "subtype" entry (this is the file's auxiliary type code; the entries displayed are the default loading address for a BIN file and the record length for a TXT file). It also displays the size of the disk, in blocks.

CREATE. This command is used to create a directory entry for a specified file. It is most often used to create subdirectory files since the other common types of ProDOS files (Applesoft programs, binary files, and textfiles) are automatically created by other BASIC.SYSTEM commands (SAVE, BSAVE, and OPEN). For example, if the volume directory is active and you want to create a subdirectory called DEMO.PROGRAMS, then you would enter the command

```
CREATE DEMO.PROGRAMS
```

from the keyboard. When you do this, the subdirectory will appear as a file entry when you catalog the directory in which it is contained. The file type mnemonic used to identify it in the catalog listing is DIR. Other types of files can be created using the ",Ttype" parameter.

DELETE. This command is used to delete a file by removing its entry from the directory and altering the volume bit map to free up the blocks that the file uses. Only unlocked files can be erased with the DELETE command. (See the description of the LOCK command.)

LOCK. This command is used to protect a ProDOS file from being accidentally or intentionally deleted, modified, or renamed. Once a file has been locked, it cannot be deleted, modified, or renamed unless it is first unlocked. You can tell which files are locked by cataloging the disk (using the CAT or CATALOG commands); if the name of the file is preceded by an asterisk ("*"), it is locked.

PREFIX. This command is used to define the chain of directory names to which any filename or partial pathname specified will automatically be appended to generate a full pathname. It is this full pathname on which the BASIC.SYSTEM commands will act. If the pathname parameter specified after the PREFIX command does not begin with a slash it will be appended to the current prefix.

RENAME. This command is used to change the name of any file on the disk from the first pathname parameter specified to the second.

UNLOCK. This command unlocks a locked file so that it can be deleted, modified, or renamed.

VERIFY. This command is used to check whether a file exists. If no error occurs, the file does exist. Entering VERIFY by itself (that is, without a pathname) will cause Apple's copyright notice to be printed.

File Loading and Execution Commands

- (dash). This is the BASIC.SYSTEM intelligent run command. Its parameter can be the pathname of an Applesoft program, a binary program, or a textfile, in which cases the - will behave exactly like a RUN, BRUN, or EXEC command,

respectively. Dash can also be used to load and execute ProDOS system (SYS) programs.

BLOAD. This command is used to transfer the data from a file to an area of memory. The most common form of this command is:

```
BLOAD MY.FILE,A#
```

where "#" represents the address of the beginning of the block to which the file is to be transferred. The default file type is binary (BIN), but you can override this by using the ",Ttype" parameter. The BLOAD command can also be used without the ",A#" parameter; in this case, the file will be loaded at the location from which it was originally saved to the disk using the BSAVE command (this address appears in the "subtype" column when the disk is cataloged using the CATALOG command). Any portion of a file can be loaded by using one or more optional parameters: ",B#" (the starting position within the file), ",L#" (the number of bytes to be transferred), and ",E#" (the last memory location to be transferred to).

BRUN. This command is the same as BLOAD except that after the file is loaded, it will automatically be executed. Execution begins at the loading address. The BRUN command can only be used with binary (BIN) files.

BSAVE. This command is used to save the contents of a range of memory to a file. (The default file type used is binary (BIN) but you can override this default with the ",Ttype" parameter.) For example, to save the contents of memory from $300-$3CF to a binary file called PAGE.THREE, you would enter the command:

```
BSAVE PAGE.THREE,A$300,E$3CF
```

or

```
BSAVE PAGE.THREE,A$300,L$D0
```

where the ",A$300" parameter indicates the starting address of the range, ",E$3CF" indicates the ending address, and ",L$D0" indicates the number of bytes to be saved. You can also use the ",B#" parameter to indicate the byte position in the file from which the write operation will take place.

EXEC. This command is used to redirect subsequent requests for input to a specified file instead of the keyboard until everything in the file has been read. For example, suppose you have defined a file called MY.STARTUP that contains the following two lines:

```
HOME
CATALOG
```

When you enter EXEC MY.STARTUP from direct mode, the screen will clear and then the disk will be cataloged, just as if you had entered the two

commands directly from the keyboard. You can use the ",F#" or ",R#" parameters to specify the number of the first line in the file to be executed.

LOAD. This command is used to load an Applesoft program into memory. (If LOAD is entered without a pathname, the program will be loaded from cassette tape.)

RUN. This command is the same as the LOAD command except that after the program is loaded, it will automatically be executed. The ",@#" parameter can be used to specify the Applesoft line number that is to be executed first; the default is the first line number. (If RUN is entered without a pathname, the program already in memory will be executed.)

SAVE. This command is used to save an Applesoft program to a file on the disk. The file type mnemonic for a program file is BAS. (If SAVE is entered without a pathname, the program will be saved to cassette tape.)

File Input/Output Commands

OPEN. This command is used to open a file (by default, a TXT file) for either reading or writing. If the pathname specified does not exist, then a new file will be created. A file must be opened before it can be accessed using the BASIC.SYSTEM READ, WRITE, FLUSH, and POSITION commands. Textfiles can be opened as one of two basic types: sequential or random-access. A sequential textfile is one in which in which lines of information are stored one after another, separated only by a carriage return code; usually, if you want to access information anywhere in the file, you have to read all the information that precedes it.

A random-access textfile is one that is organized as a series of fixed-length records that hold related groups of information; any record can be accessed *randomly* (that is, without reading all previous records first) simply by specifying its record number when using the READ command. The record length is assigned to a random-access textfile when it is first opened by using the ",L#" parameter; it is displayed in the subtype column of a CATALOG listing in the form "R=$xxxx." For example, if the record length is 127, then the subtype entry would be "R=$007F."

READ. This command is used to redirect subsequent requests for input to an open file instead of the keyboard. If a random-access textfile is being read, the record number to be read can be specified using the ",R#" parameter. You can also specify a field number (a field is a string of characters terminated by a carriage return code) by using the ",F#" parameter or a byte number using the ",B#" parameter. If more than one of these parameters is used, READ will first skip to the proper record number, then to the proper field number, and finally to the proper byte position. (That is, the byte position is relative to the current field position.)

POSITION. This command is used to adjust the position in the file at which subsequent read and write operations will operate. The number of fields to skip over is specified by the ",F#" or ",R#" parameter.

WRITE. This command is used to redirect subsequent output to an open file instead of to the video screen. It works much like the READ command except in the opposite direction.

APPEND. This command is used to open a file and redirect subsequent output to the end of the file. The default file type is a textfile, but this can be changed using the ",Ttype" parameter.

FLUSH. When BASIC.SYSTEM opens a file, a file buffer is allocated to it in memory. Data written to the file is stored in this buffer first and is normally stored on the disk only when the buffer becomes full or when another file block must be accessed. The FLUSH command is used to force any data stored in the buffer to be saved to the disk even if the buffer is not yet full. This minimizes the risk of data loss in the event of an unexpected exit from the program (caused by a loss of power, pressing RESET, and so on) but it slows down disk write operations considerably. FLUSH also causes the file's directory entry to be updated. If FLUSH is used without a pathname, all open files will be flushed.

CLOSE. This command is used to close a file that has been opened using the OPEN or APPEND command. When a file is closed, its buffer is automatically flushed and its directory entry is updated. If CLOSE is used without a pathname, all open files will be closed.

Miscellaneous Commands

BYE. This command is used to disconnect BASIC.SYSTEM and execute a ProDOS system program by calling the MLI QUIT ($65) command. When the command is entered, the ProDOS selector program that was discussed earlier in this chapter will be executed. The standard selector will prompt you to enter the prefix and partial pathname of the next system program; after you do this, this program will be executed.

CHAIN. This command is used to transfer control from one Applesoft program to another while maintaining the names and current values of all the variables in the program from which control is being passed. This allows very large programs to be executed by breaking them into separate modules and chaining them together. You can chain to any line number in the new program by using the ",@#" parameter.

FRE. This command is used to force garbage collection of Applesoft string variables. This garbage collection command is much faster than the one of the same name built into the Applesoft interpreter. (See Chapter 4 of *Inside the Apple //e* for more information on the garbage collection procedure.)

IN#. This command is used to redirect subsequent requests for input to a peripheral card subroutine at $Cn00 (where "n" is a slot number) or to any other specified subroutine. If a slot number of 0 is specified, the standard keyboard input subroutine at KEYIN ($FD1B) is used instead. IN# can also be used to associate the address of any input subroutine with any slot number by using the "snum,A#" construct; once this is done, a IN#n command can be used to direct later requests for input to this subroutine rather than to $Cn00.

NOMON. This command is allowed but is ignored. Under DOS 3.3 the command is used to disable the display of disk commands and I/O operations; under BASIC.SYSTEM, they are always disabled.

PR#. This command is used to redirect subsequent output to a peripheral card subroutine at $Cn00 or to any other specified subroutine. If a slot number of 0 is specified, the standard 40-column video output subroutine at COUT1 ($FDF0) is used instead. PR# can also be used to associate the address of any output subroutine with any slot number by using the "snum,A#" construct; once this is done, a PR#n command can be used to direct subsequent output to this subroutine rather than to $Cn00.

RESTORE. This command is used to initialize the names and values of the variables in an Applesoft program to those contained in the file specified in the argument. This file must be of type VAR (this is the type created by the STORE command).

STORE. This command is used to save the names and current values of all the variables in an Applesoft program to a file. The mnemonic for the file type code assigned to the file is VAR.

BASIC.SYSTEM and the Input and Output Links

It is often necessary for an Applesoft program to redirect input or output requests to a device in one of the Apple's expansion slots (called ports on the slotless Apple //c). This is most easily done by using the BASIC.SYSTEM IN# and PR# commands. For example, to redirect output to a printer in slot #1, you would execute this statement:

```
PRINT CHR$(4);"PR#1"
```

The confusingly similar Applesoft commands of the same names, must NOT be used to redirect I/O when using BASIC.SYSTEM.

You can also use a special form of the IN# and PR# commands to redirect I/O to a subroutine that is located anywhere in memory and not just to a program located in the ROM on a peripheral card. The only restriction on its use is that the first byte of the new input subroutine must be a 6502 "CLD" (clear decimal flag) instruction. To direct I/O to any such I/O subroutine, you must execute a statement like

```
PRINT CHR$(4);"IN# Aaddr"
```

or

```
PRINT CHR$(4);"PR# Aaddr"
```

where "addr" represents either the decimal starting address of the new I/O subroutine or, if preceded by "$", the hexadecimal starting address. For exam-

ple, if a new input subroutine begins at $300 (decimal 768), then you would execute either of the following two statements:

```
PRINT CHR$(4);"IN# A$300"
```

or

```
PRINT CHR$(4);"IN# A768"
```

to install the new subroutine.

Problems can arise when trying to redirect I/O in a BASIC.SYSTEM environment using assembly language techniques. Traditionally, I/O requests are redirected by storing the address of a new input routine in KSW ($38/$39) and the address of a new output routine in CSW ($36/$37); KSW and CSW are called the input and output links, respectively. However, as we saw earlier, this is exactly how BASIC.SYSTEM gets its hooks into the system. Thus, if we were to overwrite these links, we would interfere with the operation of BASIC.SYSTEM and we may even disconnect it. (If you accidentally disconnect BASIC.SYSTEM like this, you can reconnect it by executing a "JSR BIENTRY" instruction; BIENTRY is located at $BE00.)

We can get around this problem in one of two ways, however. First, we can use the BRUN command to load and execute any assembly language program that modifies the standard I/O links This works because just before the program that is BRUN ends, BASIC.SYSTEM checks to see whether the I/O links have changed. If they have, the new link addresses are moved into BASIC.SYSTEM's own internal I/O links and the addresses of its own I/O subroutines are restored. The BASIC.SYSTEM I/O links are used just like the standard ones and the subroutines whose addresses are stored in them are called when BASIC.SYSTEM wants to perform standard (non-disk) I/O operations.

Alternatively, we can install a new input or output subroutine by storing its address directly into the appropriate internal BASIC.SYSTEM link itself: the input link at VECTIN ($BE32/$BE33) or the output link at VECTOUT ($BE30/$BE31).

Note that any other method that is used to change the standard input links (such as POKEing new values from an Applesoft program or using CALL to execute a subroutine that stores new values) will not work properly.

Reserving Space Above the File Buffers

As shown in Figure 5-1, when BASIC.SYSTEM is loaded, its code occupies the entire space from $9A00 to $BEFF. BASIC.SYSTEM also sets up a $400-byte (1K) general-purpose buffer that initially sits just below this area, beginning at $9600. The entire space above $9600 is protected by setting the Applesoft HIMEM pointer to $9600. (HIMEM refers to the address stored in the Applesoft

end-of-string pointer at $73/$74.) This means that the Applesoft string variable storage space that ends at HIMEM will not conflict with BASIC.SYSTEM.

The general-purpose buffer always occupies the 1K area just above HIMEM, even if HIMEM is changed. It is used by BASIC.SYSTEM as a temporary storage area for directory blocks whenever the disk is CATALOGed.

BASIC.SYSTEM automatically changes HIMEM whenever files are opened or closed using its OPEN, APPEND, and CLOSE commands. It is not immediately obvious why this is necessary, so let's examine how BASIC.SYSTEM manages files in a bit more detail. Whenever a file is opened, BASIC.SYSTEM creates a $400-byte buffer for it by lowering HIMEM by that number of bytes (and moving the general-purpose buffer down with it) and then reserving the $400 byte area beginning at the original HIMEM position for use as the buffer. If another file is opened (up to eight files can be open at once), the process is repeated so that the new buffer fits in just below the first one. (*Exception*: if a file is opened using the EXEC command, its buffer is always positioned immediately above the highest- addressed active buffer.) When a file is closed, its buffer is "removed" by relocating the lowest-addressed active file buffer to the position of the closed buffer and then raising HIMEM by $400 bytes. Note that BASIC.SYSTEM takes all steps necessary to ensure that Applesoft's string variables cannot be overwritten despite the fluctuations in HIMEM.

It is often convenient to reserve a safe area of memory where assembly language programs can be stored without fear of being overwritten by either BASIC.SYSTEM or Applesoft itself. One such area is between $300–$3CF in page 3 but there is only room for very short programs there. In DOS 3.3 days, an alternative area could be reserved simply by lowering HIMEM and then storing the program between the new and old HIMEM addresses. You can't do this with BASIC.SYSTEM, however, because of the way that buffers are positioned when files are opened or closed. When BASIC.SYSTEM is being used, however, a safe area can be reserved that resides above the $400-byte directory buffer beginning at HIMEM. Follow these steps:

- Close all files using the BASIC.SYSTEM CLOSE command.
- Lower HIMEM by a multiple of $100 (256) bytes using the Applesoft HIMEM: command. (The HIMEM: command simply places the address specified in the argument of the command directly into the HIMEM pointer.)

These steps must be performed before any Applesoft string variables have been defined since the existing Applesoft string space will be overwritten. After these two steps have been completed, the area from HIMEM+$400 to $99FF can be used for storage of machine-language programs without danger of them being overwritten by BASIC.SYSTEM operations. Note that you must be very careful when using the Applesoft HIMEM: command because no checks are made to ensure that the address specified in the command is an integral multiple of 256. BASIC.SYSTEM will not operate properly if HIMEM does not rest on a page boundary.

Alternatively, you can, at any time, call the GETBUFR ($BEF5) subroutine from an assembly language program if you want to free up a space of contiguous

256-byte pages above HIMEM. This is done by placing the number of pages to be reserved in the accumulator and then calling GETBUFR; upon exit, the carry flag will be clear if there was enough free space available, or it will be set if there wasn't. If all went well, the first page reserved will be in the accumulator. We'll see an example of the use of GETBUFR later on in this chapter in the installation code for a user-defined command called ONLINE.

You can de-allocate space that was reserved using GETBUFR by calling the FREEBUFR ($BEF8) subroutine. This subroutine frees up *all* buffers that have been reserved using GETBUFR since bootup by setting HIMEM back to its original value stored at PAGETOP ($BEFB). (You can actually selectively free up the most recently allocated buffers by setting PAGETOP to the page number, less 4, of the start of the buffer that you don't want freed up.)

Whenever you reserve space above HIMEM it is a good idea to modify the system bit map to indicate that the memory pages reserved are in use. If this is done, the ProDOS MLI interpreter will not permit these pages to be used as buffer areas when MLI commands are requested. See Chapter 3 for a discussion of the system bit map.

BASIC.SYSTEM Page Three Usage

We saw in Chapter 3 that ProDOS reserves the area from $3D0–$3EC for use by system programs like BASIC.SYSTEM. As shown in Table 5-3, BASIC.SYSTEM only uses the first six locations; they are both used to reconnect BASIC.SYSTEM by jumping to the BASIC.SYSTEM warm-start entry point.

Table 5-3. ProDOS–BASIC.SYSTEM page 3 vectors.

Address	*Description of Vector*
$3D0-$3D2	A JMP instruction to the BASIC.SYSTEM warm-start entry point. A call to this vector will reconnect BASIC.SYSTEM without destroying the Applesoft program in memory. Use the "3DØG" command to move from the system monitor to Applesoft.
$3D3-$3D5	Another JMP instruction to the BASIC.SYSTEM warm-start entry point.

Note: Locations $3D6-$3EC are also reserved for use by a ProDOS system program.

BASIC.SYSTEM also initializes most of the system vectors from $3ED to $3FF when it is first installed. The contents of this area of page 3 are shown in Table 5-4.

The rest of page 3 (from $300 to $3CF) is unused and is a convenient spot to store short assembly language subroutines called from an Applesoft program.

Table 5-4. Initialization of page 3 system vectors by ProDOS and BASIC.SYSTEM.

Vector Name	*Address*	*Contents*	*Description*
XFERLOC	$3ED-$3EE	[not initialized]	Address to which controlpasses when XFER ($C314) is called (//e, //c only)
BRK	$3F0-$3F1	$FA59	Address of a subroutine that displays the 6502 registers and then enters the system monitor
RESET	$3F2-$3F3 $3F4	$BE00 $1B	Address of the BASIC.SYSTEM warm-start entry point (reconnects BASIC.SYSTEM) followed by "powered-up" byte
&	$3F5-$3F7	"JMP $BE03"	Jump to BASIC.SYSTEM's external entry point for command strings (see Chapter 5)
USER	$3F8-$3FA	"JMP $BE00"	Jump to BASIC.SYSTEM'S warm-start entry point
NMI	$3FB-$3FD	"JMP $FF59"	Jump to the system monitor's cold-start entry point
IRQ	$3FE-$3FF	$BFEB	Address of the special ProDOS interrupt handler (see Chapter 6)

Note: The addresses stored at each vector location are stored with the low-order byte first.

The BASIC.SYSTEM Global Page : $BE00–$BEFF

The BASIC.SYSTEM global page is located from $BE00 to $BEFF, just beneath the ProDOS global page. It contains a number of fixed-position subroutines and data areas that can be used by assembly language programs to communicate easily with BASIC.SYSTEM. For example, the global page contains entry points for connecting BASIC.SYSTEM to the system, for executing ASCII command strings, for handling user-installed commands, for handling errors, and for executing MLI commands. A commented source listing for the BASIC.SYSTEM global page is shown in Table 5-5.

The GOSYSTEM Subroutine

Most of the global page is used to support the GOSYSTEM ($BE70) subroutine that the BASIC.SYSTEM code calls whenever it needs to execute an MLI command. On entry, GOSYSTEM constructs a standard "JSR MLI" call by storing the MLI command number (that is passed in the accumulator) at SYSCALL ($BE85) and the address of the command's parameter table at SYSPARM ($BE86). (As shown in Table 5-5, each command used by BASIC.SYSTEM has

Table 5-5. Source listing for the BASIC.SYSTEM global page.

```
1    ******************************
2    * BASIC.SYSTEM Global Page   *
3    *                            *
4    *        $BE00..$BEFF        *
5    *    (ProDOS 1.1.1 only)     *
6    ******************************
7
8    * Note: these addresses are valid for ProDOS
9    *       version 1.1.1 only!
10
11   TXBUF2    EQU  $280
12   SYSOUT    EQU  $9A2F       ;Internal output subroutine
13   SYSIN     EQU  $9ABA       ;Internal input subroutine
14   NODEVERR  EQU  $9AEE
15   ERROR     EQU  $9AF0
16   PRTERR    EQU  $9F88
17   PAGEGET   EQU  $A2B5
18   PAGEFREE  EQU  $A301
19   SYNTAX    EQU  $A677
20   WARMDOS   EQU  $ABF1
21   DOSOUT    EQU  $B7F1      ;Character out intercept
22   DOSIN     EQU  $B7F4      ;Character in intercept
23   SYSCTBL   EQU  $B805      ;Table of parm table addresses
24   MLIERTBL  EQU  $B9EE      ;Table of MLI error codes
25   BIERRTBL  EQU  $BA01      ;Table of Applesoft error
                                codes
26   CALLX     EQU  $BCA8
27   TXBUF     EQU  $BCBD
28
29   MLI       EQU  $BF00
30
```

```
                        31  COUT1    EQU  $FDF0       ;Video output
                        32  KEYIN    EQU  $FD1B       ;Keyboard input
                        33
                        34           ORG  $BE00
                        35
BE00: 4C F1 AB          36  BIENTRY  JMP  WARMDOS     ;Connect BASIC.SYSTEM I/O
                                                       links
BE03: 4C 77 A6          37  DOSCMD   JMP  SYNTAX      ;Execute command string at
                                                       $200
BE06: 4C 9E BE          38  EXTRNCMD JMP  XRETURN     ;User command handler
                        39
                        40  ********************************************************
                        41  * ERROUT is called by BASIC.SYSTEM whenever a          *
                        42  * disk error condition is detected. (The error         *
                        43  * code--2..22--is stored in the accumulator.)          *
                        44  * ERROUT stores the error code in ERRCODE and in       *
                        45  * $DE (required by Applesoft) and then if ONERR        *
                        46  * is active it passes control to the Applesoft         *
                        47  * error handling subroutine; if it isn't, an           *
                        48  * error message is printed by calling PRINTERR.        *
                        49  ********************************************************
BE09: 4C F0 9A          50  ERROUT   JMP  ERROR      ;Applesoft error handler
BE0C: 4C 88 9F          51  PRINTERR JMP  PRTERR     ;Print error message
BE0F: 00                52  ERRCODE  DFB  0          ;Error code
                        53
                        54  ************************************************
                        55  * The following table holds the addresses to   *
                        56  * be placed in the output link whenever a      *
                        57  * PR#s command is entered. If a peripheral     *
                        58  * card is in a particular slot, the entry      *
                        59  * will be of the form "Cs00"; if no card is    *
                        60  * installed, the address of the subroutine     *
```

(continued)

Table 5-5. Source listing for the BASIC.SYSTEM global page (continued).

```
                        61  * that generates a "no device connected"     *
                        62  * error code is stored instead. Any address  *
                        63  * can be placed in the table using the       *
                        64  * Applesoft PRINT CHR$(4);"PR# s,A#"         *
                        65  * construct.                                 *
                        66  *********************************************
BE10: F0 FD             67  OUTVECT0 DA   COUT1      ;Standard video output
BE12: 00 C1             68  OUTVECT1 DA   $C100      ;(Assume printer card)
BE14: 00 C2             69  OUTVECT2 DA   $C200      ;(Assume modem card)
BE16: 00 C3             70  OUTVECT3 DA   $C300      ;(Assume 80-column card)
BE18: 00 C4             71  OUTVECT4 DA   $C400      ;(Assume mouse card)
BE1A: EE 9A             72  OUTVECT5 DA   NODEVERR
BE1C: 00 C6             73  OUTVECT6 DA   $C600      ;(Assume disk drive)
BE1E: EE 9A             74  OUTVECT7 DA   NODEVERR
                        75
                        76  *********************************************
                        77  * The following table holds the addresses to *
                        78  * be placed in the output link whenever a    *
                        79  * IN#s command is entered. If a peripheral   *
                        80  * card is in a particular slot, the entry    *
                        81  * will be of the form "Cs00"; if no card is  *
                        82  * installed, the address of the subroutine   *
                        83  * that generates a "no device connected"     *
                        84  * error code is stored instead. Any address  *
                        85  * can be placed in the table using the       *
                        86  * Applesoft PRINT CHR$(4);"IN# s,A#"         *
                        87  * construct.                                 *
                        88  *********************************************
BE20: 1B FD             89  INVECT0  DA   KEYIN      ;Standard keyboard input
BE22: 00 C1             90  INVECT1  DA   $C100      ;(Assume printer card)
BE24: 00 C2             91  INVECT2  DA   $C200      ;(Assume modem card)
```

```
BE26: 00 C3     92  INVECT3  DA   $C300       ;(Assume 80-column card)
BE28: 00 C4     93  INVECT4  DA   $C400       ;(Assume mouse card)
BE2A: EE 9A     94  INVECT5  DA   NODEVERR
BE2C: 00 C6     95  INVECT6  DA   $C600       ;(Assume disk drive)
BE2E: EE 9A     96  INVECT7  DA   NODEVERR
                97
                98  ************************************************
                99  * The BASIC.SYSTEM I/O links are stored here.  *
                100 * These are the addresses to which control     *
                101 * will pass if the input or output is not      *
                102 * handled internally.                          *
                103 ************************************************
BE30: F0 FD     104 VECTOUT  DA   COUT1       ;ProDOS output link
BE32: 1B FD     105 VECTIN   DA   KEYIN       ;ProDOS input link
                106
                107 * Miscellaneous internal BASIC.SYSTEM parameters:
                108
BE34: F1 B7     109 VDOSIO   DA   DOSOUT      ;Character out intercept
BE36: F4 B7     110          DA   DOSIN       ;Character in intercept
                111
BE38: 2F 9A     112 VSYSIO   DA   SYSOUT      ;Internal output subroutine
BE3A: BA 9A     113          DA   SYSIN       ;Internal input subroutine
                114
BE3C: 06        115 DEFSLT   DFB  6           ;Default slot #
BE3D: 01        116 DEFDRV   DFB  1           ;Default drive #
                117
BE3E: 00        118 PREGA    DFB  0           ;Temporary storage for A
BE3F: 00        119 PREGX    DFB  0           ;Temporary storage for X
BE40: 00        120 PREGY    DFB  0           ;Temporary storage for Y
                121
BE41: 00        122 DTRACE   DFB  0           ;bit 7=1 ==> Applesoft trace
                                               on
                123
```

(continued)

Table 5-5. Source listing for the BASIC.SYSTEM global page (continued).

```
BE42: 00        124 STATE     DFB  0          ;0=direct, <>0=in program
BE43: 00        125 EXACTV    DFB  0          ;bit 7=1 ==> EXEC file open
BE44: 00        126 IFILACTV  DFB  0          ;bit 7=1 ==> input file active
BE45: 00        127 OFILACTV  DFB  0          ;bit 7=1 ==> output file
                                               active
BE46: 00        128 PFXACTV   DFB  0          ;bit 7=1 ==> prefix input
                                               active
BE47: 00        129 DIRFLG    DFB  0          ;bit 7=1 ==> dir. file active
BE48: 00        130 EDIRFLG   DFB  0          ;bit 7=1 ==> end of directory
BE49: 00        131 STRINGS   DFB  0          ;Counter for free space calc.
BE4A: 00        132 TBUFPTR   DFB  0          ;Character count for WRITE
BE4B: 00        133 INPTR     DFB  0          ;Char. count for kbd input
BE4C: 00        134 CHRLAST   DFB  0          ;Last character printed
BE4D: 00        135 OPENCNT   DFB  0          ;Number of open files (not
                                               EXEC)
BE4E: 00        136 EXFILE    DFB  0          ;EXEC file close flag
BE4F: 00        137 CATFLAG   DFB  0          ;Directory input flag
                138
                139 ************************************************
                140 * The following three locations will be used if  *
                141 * you are adding user commands to BASIC.SYSTEM.  *
                142 ************************************************
BE50: 00 00     143 XTRNADDR DA   0           ;Address of user command
                                               handler
BE52: 00        144 XLEN      DFB  0          ;Length of user command - 1
BE53: 00        145 XCNUM     DFB  0          ;Command number in use
                                               (0=user)
                146
                147 * Notes on PBITS and FBITS:
                148 *
                149 * Once BASIC.SYSTEM has identified a valid command,
```

```
                150 * it stores a number in PBITS and PBITS+1 that
                151 * reflects the syntax of the command. It then calls
                152 * the command parser, which updates FBITS and
                153 * FBITS+1 to reflect the parameters that were
                154 * actually found.
                155 *
                156 * Meaning of bits in PBITS/FBITS:
                157 *
                158 * bit 7 : fetch prefix if pathname not specified
                159 * bit 6 : slot number required/found
                160 * bit 5 : command NOT valid in direct mode
                161 * bit 4 : pathname is optional (no names+parms)
                162 * bit 3 : create file if it doesn't exist
                163 * bit 2 : file type optional ("T" parameter)/found
                164 * bit 1 : second pathname required (for RENAME)/found
                165 * bit 0 : filename allowed/found
                166
                167 * Meaning of bits in PBITS+1/FBITS+1:
                168 *
                169 * bit 7 : "A" parameter allowed/found
                170 * bit 6 : "B" parameter allowed/found
                171 * bit 5 : "E" parameter allowed/found
                172 * bit 4 : "L" parameter allowed/found
                173 * bit 3 : "@" parameter allowed/found
                174 * bit 2 : "S"/"D" parameters allowed/found
                175 * bit 1 : "F" parameter allowed/found
                176 * bit 0 : "R" parameter allowed/found
                177
BE54: 00 00     178 PBITS    DW   0          ;Permitted parameter bits
BE56: 00 00     179 FBITS    DW   0          ;Found parameter bits
                180
```

(continued)

Table 5-5. Source listing for the BASIC.SYSTEM global page (continued).

```
                        181 ********************************************************
                        182 * The following table is where command parameters      *
                        183 * are stored during a parsing operation. The           *
                        184 * entries for unspecified parameters are not           *
                        185 * changed.                                             *
                        186 ********************************************************
BE58: 00 00             187 APARM    DA   0          ;"A" (address) parameter
BE5A: 00 00 00          188 BPARM    DS   3          ;"B" (byte #) parameter
BE5D: 00 00             189 EPARM    DA   0          ;"E" (end addr) parameter
BE5F: 00 00             190 LPARM    DW   0          ;"L" (length) parameter
BE61: 00                191 SPARM    DFB  0          ;"S" (slot) parameter
BE62: 00                192 DPARM    DFB  0          ;"D" (drive) parameter
BE63: 00 00             193 FPARM    DW   0          ;"F" (field #) parameter
BE65: 00 00             194 RPARM    DW   0          ;"R" (record #) parameter
BE67: 00                195 VPARM    DFB  0          ;"V" (volume #) parameter
BE68: 00 00             196 @PARM    DW   0          ;"@" (line #) parameter
BE6A: 00                197 TPARM    DFB  0          ;"T" (file type code)
                                                      parameter
BE6B: 00                198 SLPARM   DFB  0          ;"slot" (for IN#,PR#)
                                                      parameter
BE6C: BC BC             199 PATH1    DA   TXBUF-1    ;Pointer to first pathname
BE6E: 80 02             200 PATH2    DA   TXBUF2     ;Pointer to second pathname
                        201
                        202 ********************************************************
                        203 * All BASIC.SYSTEM MLI calls are routed to GOSYSTEM   *
                        204 * with the command number in the accumulator.         *
                        205 * Prior to calling GOSYSTEM, BASIC.SYSTEM             *
                        206 * sets up the appropriate parameter table in the      *
                        207 * global page as required by the call. GOSYSTEM       *
                        208 * handles all MLI calls from $C0..$D3 inclusive. If   *
                        209 * an error occurs, an Applesoft error code is         *
```

```
                    210 * returned in A with the carry flag set.            *
                    211 ************************************************
BE70: 8D 85 BE      212 GOSYSTEM STA  SYSCALL    ;Save MLI command number
BE73: 8E A8 BC      213          STX  CALLX      ;Save X register
BE76: 29 1F         214          AND  #$1F       ;# mod 32
BE78: AA            215          TAX
BE79: BD 05 B8      216          LDA  SYSCTBL,X  ;Get address of parm table
BE7C: 8D 86 BE      217          STA  SYSPARM    ; (low) and save it
BE7F: AE A8 BC      218          LDX  CALLX      ;Restore X
BE82: 20 00 BF      219          JSR  MLI        ;Do the MLI call
BE85: 00            220 SYSCALL  DFB  0          ;MLI command # stored here
BE86: 00            221 SYSPARM  DFB  0          ;Address of parm table (low)
BE87: BE            222          DFB  $BE        ;High address always $BE
BE88: B0 01         223          BCS  BADCALL    ;Branch if error
BE8A: 60            224          RTS
                    225
                    226 ******************************************
                    227 * The BADCALL subroutine converts the MLI *
                    228 * error code to a corresponding Applesoft *
                    229 * error code.                             *
                    230 ******************************************
BE8B: A2 12         231 BADCALL  LDX  #$12
BE8D: DD EE B9      232 MLIERR1  CMP  MLIERTBL,X ;Is it a "known" MLI error?
BE90: F0 05         233          BEQ  MLIERR2    ;Yes, so branch
BE92: CA            234          DEX             ;Check all 19 possibilities
BE93: 10 F8         235          BPL  MLIERR1
BE95: A2 13         236          LDX  #$13       ;Not known, so "I/O error"
237
BE97: BD 01 BA      238 MLIERR2  LDA  BIERRTBL,X ;Convert to Applesoft error
                                                  code
BE9A: AE A8 BC      239          LDX  CALLX      ;Restore X
BE9D: 38            240          SEC             ;==> error
BE9E: 60            241 XRETURN  RTS
```

(continued)

Table 5-5. Source listing for the BASIC.SYSTEM global page (continued).

```
BE9F: 00        242          DFB  $00         ;Unused byte
                243
                244 ********************************************************
                245 * The parameter tables for each of the MLI functions   *
                246 * supported by BASIC.SYSTEM follow. These tables       *
                247 * must be filled in before calling GOSYSTEM.           *
                248 ********************************************************
                249 * Parm table for CREATE:
BEA0: 07        250          DFB  $07         ;Number of parameters
BEA1: BC BC     251          DA   TXBUF-1     ;Pathname pointer
BEA3: C3        252          DFB  $C3         ;Access code
BEA4: 00        253          DFB  0           ;File type code
BEA5: 00 00     254          DW   $0000       ;Auxiliary type code
BEA7: 00        255          DFB  0           ;Storage type code (usually 1)
BEA8: 00 00     256          DW   0           ;Create date
BEAA: 00 00     257          DW   0           ;Create time
                258
                259 * Parm table for DESTROY, SET_PREFIX, GET_PREFIX:
BEAC: 01        260          DFB  $01         ;Number of parameters
BEAD: BC BC     261          DA   TXBUF-1     ;Pathname pointer
                262
                263 * Parm table for RENAME:
BEAF: 02        264          DFB  $02         ;Number of parameters
BEB0: BC BC     265          DA   TXBUF-1     ;Old pathname pointer
BEB2: 80 02     266          DA   TXBUF2      ;New pathname pointer
                267
                268 * Parm table for SET_FILE_INFO and GET_FILE_INFO:
BEB4: 00        269          DFB  $00         ;=7 (SFI) or =10 (GFI)
BEB5: BC BC     270          DA   TXBUF-1     ;Pathname pointer
BEB7: 00        271          DFB  $00         ;Access code
BEB8: 00        272          DFB  $00         ;File type code
```

```
BEB9: 00 00     273           DW   $0000       ;Auxiliary type code
BEBB: 00        274           DFB  $00         ;Storage type code (GFI only)
BEBC: 00 00     275           DW   $0000       ;Blocks used (GFI only)
BEBE: 00 00     276           DW   $0000       ;Modification date
BEC0: 00 00     277           DW   $0000       ;Modification time
BEC2: 00 00     278           DW   $0000       ;Create date (GFI only)
BEC4: 00 00     279           DW   $0000       ;Create time (GFI only)
                280
                281 * Parm table for ON_LINE, SET_MARK, GET_MARK,
                282 * SET_EOF,GET_EOF,SET_BUF,GET_BUF:
BEC6: 02        283           DFB  $02         ;Number of parameters
BEC7: 00        284           DFB  $00         ;Unit or reference number
BEC8: 00        285           DFB  $00         ;2-byte pointer to data buffer
BEC9: 00        286           DFB  $00         ; (BUF, ON_LINE), or 3-byte
BECA: 00        287           DFB  $00         ; position (MARK, EOF).
                288
                289 * Parm table for OPEN:
BECB: 03        290           DFB  $03         ;Number of parameters
BECC: BC BC     291           DA   TXBUF-1     ;Pathname pointer
BECE: 00 00     292           DA   $0000       ;Buffer pointer (1K)
BED0: 00        293           DFB  0           ;Reference number
                294
                295 * Parm table for NEWLINE:
BED1: 03        296           DFB  $03         ;Number of parameters
BED2: 00        297           DFB  0           ;Reference number
BED3: 7F        298           DFB  $7F         ;Ignore state of high bit
BED4: 0D        299           DFB  $0D         ;Newline is $0D or $8D
                300
                301 * Parm table for READ and WRITE:
BED5: 04        302           DFB  $04         ;Number of parameters
BED6: 00        303           DFB  $00         ;Reference number
BED7: 00 00     304           DA   $0000       ;Buffer pointer
```

(continued)

Table 5-5. Source listing for the BASIC.SYSTEM global page (continued).

```
BED9: 00 00     305           DW   $0000       ;Number of bytes to read/write
BEDB: 00 00     306           DW   $0000       ;Actual number read/written
                307
                308 * Parm table for CLOSE and FLUSH:
BEDD: 01        309           DFB  $01         ;Number of parameters
BEDE: 00        310           DFB  0           ;Reference number
                311
BEDF: 00        312           DFB  0           ;Unused byte
                313
BEE0: C3 CF D0  314           ASC  "COPYRIGHT APPLE, 1983"
BEE3: D9 D2 C9 C7 C8 D4 A0 C1
BEEB: D0 D0 CC C5 AC A0 B1 B9
BEF3: B8 B3
                315
                316 ************************************************
                317 * Call GETBUFR to free up "A" pages above HIMEM. If *
                318 * the carry flag is set upon exit, there was not    *
                319 * enough memory to do so; otherwise, "A" will       *
                320 * contain the number of the first page of the       *
                321 * buffer. Call FREEBUFR to remove the buffer and    *
                322 * restore HIMEM to its bootup value (that value is  *
                323 * stored at PAGETOP).                               *
                324 ************************************************
BEF5: 4C B5 A2  325 GETBUFR  JMP  PAGEGET     ;Reserve "A" pages above HIMEM
BEF8: 4C 01 A3  326 FREEBUFR JMP  PAGEFREE    ;Restore original HIMEM
BEFB: 96        327 PAGETOP  DFB  $96         ;HIMEM page on boot
                328
BEFC: 00 00 00  329           DS   4           ;Unused bytes
BEFF: 00
```

its own parameter table in the global page—the values in the table are set up before the call to GOSYSTEM.) Since SYSCALL and SYSPARM are located right after the "JSR MLI" instruction, as required by the MLI command interpreter, the command will be properly invoked when the "JSR MLI" is actually executed.

You can call GOSYSTEM directly from your own assembly language programs to execute MLI commands if you wish. To do this, first set up the parameters in the appropriate internal parameter table and then call GOSYSTEM with the MLI command number in the accumulator. The code to do this is very simple and looks like this:

```
[set up parameter
table]
,
LDA #CMDNUM        ;Put MLI command number in A
JSR GOSYSTEM       ;Let GOSYSTEM execute command
BCS ERROR
```

This method is a bit more convenient than simply calling MLI ($BF00) in the usual way (see Chapter 4) because space for the command parameter tables has already been reserved in the global page. Furthermore, GOSYSTEM automatically sets up the JSR MLI/DFB CMDNUM/DA PARMTBL calling block and converts MLI error codes to more familiar BASIC.SYSTEM error codes. (We'll talk more about error handling in the next section.)

Note, however, that GOSYSTEM can only be called to execute MLI commands from $C0 to $D3. Any other commands must be executed using the standard "JSR MLI" technique.

One important caveat. When using GOSYSTEM you must be careful not to disturb the values of certain parameter table entries that are set up as constants by BASIC.SYSTEM. These parameters are:

- The pathname pointers in all parameter lists,
- The time and date entries at $BEAA/$BEAB and $BEA8/$BEA9 in the CREATE parameter list (they should both be zero),
- The "newline character" entry at $BED4 in the NEWLINE parameter list (it should always be $0D).

If you want to change any of these parameters temporarily, save their values first, then restore them after the GOSYSTEM call.

We will be discussing some of the other important areas of the BASIC.SYSTEM global page in the sections that follow.

BASIC.SYSTEM Error Handling

If a call to GOSYSTEM results in a system error, GOSYSTEM branches to BADCALL ($BE8B), a subroutine that converts the returned MLI error code into a BASIC.SYSTEM (Applesoft) error code. The correspondence between a given

MLI code and a BASIC.SYSTEM code is shown in Table 5-6.

(Note that only nineteen MLI error codes are specifically dealt with by BADCALL. All others are automatically converted to error code 8 ("I/O ERROR"). Also, four of the BASIC.SYSTEM error codes do not correspond to any MLI error code at all; these error codes are generated by illegal conditions within BASIC.SYSTEM itself—such as an attempt to load a program that is too large.)

Once the MLI error code has been converted, BASIC.SYSTEM calls ERROUT ($BE09) to handle the error. This subroutine first stores the error code in ERRCODE ($BE0F) and at $DE (this is where the Applesoft interpreter expects to find an error code) and then checks to see if the Applesoft ONERR GOTO error-trapping feature is active; if it is, then control passes to the internal Applesoft error-handling subroutine. If it isn't, then PRINTERR ($BE0C) is called to print the error message corresponding to the error code (see Table 5-6).

Table 5-6. BASIC.SYSTEM error codes and messages.

BASIC.SYSTEM Error Code	*MLI Error Code*	*BASIC.SYSTEM Error Message*
$00	$00	[no error occurred]
$02	$4D	RANGE ERROR
$03	$28	NO DEVICE CONNECTED
$04	$2B	WRITE PROTECTED
$05	$4C	END OF DATA
$06	$45,$44	PATH NOT FOUND
$07	$46	PATH NOT FOUND
$08	[all others]	I/O ERROR
$09	$48	DISK FULL
$0A	$4E	FILE LOCKED
$0B	$53	INVALID PARAMETER
$0C	$56,$42,$41	NO BUFFERS AVAILABLE
$0D	$4B	FILE TYPE MISMATCH
$0E	—	PROGRAM TOO LARGE
$0F	—	NOT DIRECT COMMAND
$10	$40	SYNTAX ERROR
$11	$49	DIRECTORY FULL
$12	$43	FILE NOT OPEN
$13	$47	DUPLICATE FILE NAME
$14	$50	FILE BUSY
$15	—	FILE(S) STILL OPEN
$16	—	DIRECT COMMAND

If you are writing an assembly language program that operates in an Applesoft-BASIC.SYSTEM environment, you can call ERROUT or PRINTERR to handle errors if you wish. You must ensure, however, that you call these subroutines with a BASIC.SYSTEM (Applesoft) error code, rather than an MLI error code, in the accumulator. You can execute a "JSR BADCALL" instruction (with the error code in the accumulator) to perform the necessary conversion.

Executing Disk Command Strings from Assembly Language

The DOSCMD ($BE03) subroutine in the BASIC.SYSTEM global page can be used by an assembly language program to interpret and execute a standard BASIC.SYSTEM disk command that is stored as an ASCII string in the Apple input buffer at $200. The string must be followed by a carriage return code ($8D). DOSCMD is only effective when an Applesoft program is actually running, so the assembly language program must be CALLed from an Applesoft program.

(Under DOS 3.3, disk commands can be executed by sending code $04 (CTRL-D) to the standard character output subroutine, COUT ($FDED), followed by the ASCII codes for the command. This technique is not supported by BASIC.SYSTEM.)

DOSCMD can be used to execute most, but not all, BASIC.SYSTEM disk commands. The commands that are *not* handled properly are "-" ("dash"), RUN, LOAD, CHAIN, READ, WRITE, APPEND, and EXEC. When DOSCMD is called to execute a command that it is able to handle, it will return a BASIC.SYSTEM error code in the accumulator; if no error occurred, the code will be 0 and the carry flag will be clear. If an error occurred, however, the carry flag will be set. An error condition can be handled by calling ERROUT ($BE09) or PRINTERR ($BE0C) as described in the previous section or by passing control to your own error handling code.

Note one important rule that must be followed by programs using DOSCMD: just before the program ends, the carry flag must be cleared by executing a CLC instruction. If the program ends with the carry flag set, the Applesoft program that called it may not work properly.

Adding Commands to BASIC.SYSTEM

One of the most useful features of BASIC.SYSTEM is the ability to extend its command set easily by installing user-defined external commands. To see how this is done, let's take a look at exactly what happens when BASIC.SYSTEM encounters a string of characters that may represent a valid command. (A flowchart of this procedure is presented in Figure 5-2.)

The first thing that BASIC.SYSTEM does is to check if one of its 32 standard commands has been specified (CATALOG, OPEN, WRITE, and so on). If one has been, then it is handled internally.

If an unidentified command is encountered, however, BASIC.SYSTEM does not immediately generate an error condition; rather, it calls a subroutine in its global page, EXTRNCMD ($BE06), to see if it can be claimed by a user-installed external command handler (the handler's address is always stored at $BE07 and $BE08). If no external command handler has actually been installed, EXTRNCMD

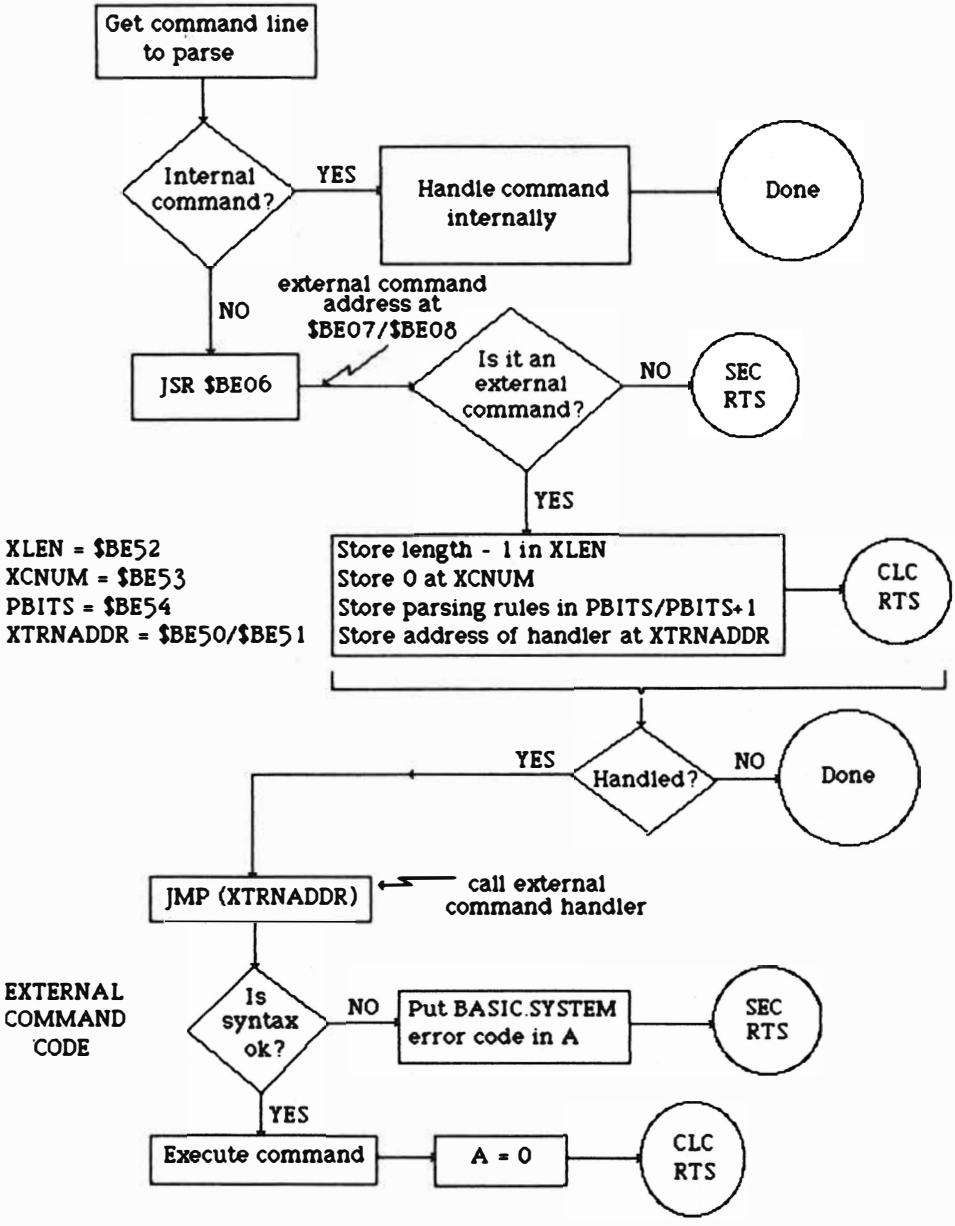


Figure 5-2. A flowchart showing the execution of an external command

will simply jump to a "do-nothing" RTS instruction at XRETURN ($BE9E). If the external command handler does not claim the command, and if the command was entered from within a program, a BASIC.SYSTEM syntax error condition will occur. If, on the other hand, the command was entered in Applesoft

direct mode, it will be passed on for consideration by the Applesoft interpreter; only if the interpreter does not recognize it will an Applesoft syntax error occur.

Let's assume that an external command handler has been installed so that a call to EXTRNCMD will pass control to it. First, such a handler executes a CLD instruction which will be used as an identification byte in future versions of BASIC.SYSTEM. The handler then must determine whether its command has, in fact, been entered; it can do this by checking if the first few characters in the command line match the expected command string. (The command line is stored in the Apple's standard input buffer beginning at $200 in ASCII-encoded form with the high bit of each code set to 1.) If they don't, the subroutine must end with the carry flag set to indicate that the command was not claimed.

If the correct command is detected, the handler can do one of two things. First, it can proceed to parse any expected parameters (such as a pathname, one of the 11 BASIC.SYSTEM letter parameters, or special parameters defined by the command itself) from the command line and then actually to execute the command. Alternatively, if all the parameters used are recognized by BASIC.SYSTEM, the handler can request that BASIC.SYSTEM do the parsing and syntax checking; the handler does this by setting certain bits in PBITS ($BE54) and PBITS+1 ($BE55) to indicate the required command syntax. If BASIC.SYSTEM does the parsing and an error is detected, BASIC.SYSTEM will perform the error handling itself. The command line parameters supported by BASIC.SYSTEM, and the range of values that they can take on, are shown in Table 5-1.

With three exceptions, each bit in PBITS and PBITS+1 is used as a flag to indicate whether the particular parameter associated with that bit is required or allowed. The exceptions are bits that indicate particular characteristics of the command: whether a prefix must be fetched for it, whether it is valid in Applesoft direct mode, and whether a file that is specified should be created if it doesn't already exist. The meaning of each bit is as follows:

PBITS ($BE54)

bit 7 fetch the current prefix if a pathname is not specified. The command line cannot contain a pathname *and* a set of parameters unless bit 0 of PBITS is also set to 1.

bit 6 a slot number is required (example: IN#, PR#). The slot number must be the first parameter after the command name and no pathnames can appear on the command line (so bit 0 and bit 1 of PBITS must both be 0).

bit 5 the command is *not* valid in direct mode.

bit 4 a pathname is optional. Pathnames *and* parameters cannot be specified on the same command line.

bit 3 create a file if the one specified does not exist.

bit 2 the file type parameter is allowed ("T" parameter). The "T" parameter can be a number or a three-character file type mnemonic corresponding to a file type code (see Table 2-4 in Chapter 2). For example, ",TDIR" can be used to select the file type code for a DIR file ($0F).

bit 1 a second pathname is required (example: RENAME). Two pathnames must be specified or else the first letter parameter will be incorrectly interpreted as a pathname.

bit 0 a pathname is allowed. Pathnames and parameters can be specified on the same command line. If the "S and D" bit (bit 2) of PBITS+1 is also set to 1, a pathname is mandatory and parameters alone cannot be specified without generating a syntax error.

PBITS+1 ($BE55):

bit 7 "A" parameter is allowed.
bit 6 "B" parameter is allowed.
bit 5 "E" parameter is allowed.
bit 4 "L" parameter is allowed.
bit 3 "@" parameter is allowed.
bit 2 "S" and "D" parameters are allowed. "S" (slot) and "D" (drive) must correspond to an existing disk drive if preceded by a file name; if preceded by a slot number specification (see bit 6 of PBITS), they do not. If "S" and "D" parameters are allowed, but not specified, their values will default to those stored at DEFSLT ($BE3C) and DEFDRV ($BE3D). If this bit is set, and no prefix is active, the name of the volume directory on the slot "S", drive "D" disk will be fetched and used to create a full pathname whenever a filename or a partial pathname is specified. If a prefix is active, it will be fetched like this only if an "S" or "D" parameter is actually specified.
bit 1 "F" parameter is allowed.
bit 0 "R" parameter is allowed.

(One other parameter is always allowed and always parsed: the "V" (volume) parameter. This parameter is tolerated by BASIC.SYSTEM commands but not used; it has been included to maintain compatibility with DOS 3.3 commands that do support it.)

The descriptions given for PBITS and PBITS+1 are applicable when the corresponding bit is set to 1. For example, if the command allows a pathname and "A" and "E" parameters, the handler would set PBITS to $01 and PBITS+1 to $A0. If a pathname is actually mandatory, bit 2 of PBITS+1 (the "S" and "D" bit) must be set to 1 as well. As indicated above, this actually serves two purposes: first, it tells BASIC.SYSTEM to automatically create a full pathname if one is not specified and, second, it tells BASIC.SYSTEM that a pathname must be specified.

If BASIC.SYSTEM is not to do any parsing, PBITS must be set to 0. Whether or not the command handler does its own parsing, if the command is found, the subroutine must store the length of the command string minus one in XLEN ($BE52), store 0 (the code number for an external command) in XCNUM ($BE53), and then store at XTRNADDR ($BE50/$BE51) the address to which control is to pass after BASIC.SYSTEM ultimately parses the command line. The latter step must be performed even if the handler has indicated that no parsing need be

performed. Lastly, the carry flag must be cleared before executing the RTS to return control to BASIC.SYSTEM.

When control returns to BASIC.SYSTEM, the parameters in the command line are parsed in accordance with the instructions stored in PBITS and PBITS+1 (if applicable). The values of the parameters that are actually parsed from the line are stored in a global page parameter table located from $BE58 to $BE6F (see Table 5-7); if a particular parameter is not detected in the parsing operation, its entry in the table stays as it was before the external command was executed. The actual parameters that were successfully parsed are indicated by setting the appropriate bits in FBITS ($BE56) and FBITS+1 ($BE57). (See Table 5-7 for a description of a BASIC.SYSTEM 1.1.1 bug that hinders the proper parsing of a command line that uses both the "V" and "@" parameters.)

Note that the first pathname that is parsed from a command line is stored in a buffer pointed to by VPATH1 ($BE6C) and the second in a buffer pointed to by VPATH2 ($BE6E). These are the same buffers pointed to by the pathname pointers in the MLI parameter tables used by BASIC.SYSTEM's GOSYSTEM ($BE70) subroutine. This means that an external command handler can use GOSYSTEM to perform MLI commands without having to modify these pointers first.

Table 5-7. BASIC.SYSTEM parameter table.

Location	*Symbolic Name*	*Meaning*
$BE58-$BE59	APARM	"A" (address) parameter
$BE5A-$BE5C	BPARM	"B" (byte #) parameter
$BE5D-$BE5E	EPARM	"E" (end addr) parameter
$BE5F-$BE60	LPARM	"L" (length) parameter
$BE61	SPARM	"S" (slot) parameter
$BE62	DPARM	"D" (drive) parameter
$BE63-$BE64	FPARM	"F" (field #) parameter
$BE65-$BE66	RPARM	"R" (record #) parameter
$BE67	VPARM	"V" (volume #) parameter (*)
$BE68-$BE69	@PARM	"@" (line #) parameter (*)
$BE6A	TPARM	"T" (file type code) parameter
$BE6B	SLPARM	"slot" (for IN#,PR#) parameter
$BE6C-$BE6D	PATH1	Pointer to first pathname
$BE6E-$BE6F	PATH2	Pointer to second pathname

(*) A bug in ProDOS 1.1.1 causes the "V" parameter to be stored in @PARM rather than VPARM (as shown). This means that "V" and "@" cannot be used together on the same command line because the value of the first parameter specified will be overwritten by the value of the other.

Note: The value associated with a parameter is stored in this table as it is parsed by BASIC.SYSTEM. If "S" and "D" parameters are allowed, but not specified, the default values stored at DEFSLT ($BE3C) and DEFDRV ($BE3D) are transferred to this table.

After a successful parse, BASIC.SYSTEM jumps to the subroutine whose address is stored at XTRNADDR ($BE50/$BE51); this is the second half of the external command handler. This subroutine can actually execute the command (if this wasn't done in the first half) and then return with the accumulator zeroed and the carry flag clear if there was no error.

If an error is detected, it can be passed to BASIC.SYSTEM for handling by setting the carry flag and placing the appropriate error code in the accumulator (the BASIC.SYSTEM error code, not the MLI error code). Alternatively, the command handler can deal with the error itself; if it does, the carry flag must be cleared and the accumulator set to 0 before returning to BASIC.SYSTEM.

Note that if BASIC.SYSTEM does the parsing, the second part of the command handler can examine FBITS to determine exactly what parameters were found and then read their values from the table beginning at $BE58. If some parameters (which were marked as optional in PBITS and PBITS+1) must be specified, the second part of the command handler can check the appropriate bits of FBITS and FBITS+1 to ensure that they are 1; if they're not, an error condition can be flagged by loading the accumulator with the BASIC.SYSTEM error code (16 for "syntax error") and setting the carry flag before returning.

The ONLINE Command

The key to understanding how to create an external command is to examine an actual example. In this section we're going to see how to design and install the handler for a new BASIC.SYSTEM command called ONLINE, which can be used to display the names of any, or all, of the disk volumes currently available to the system. This command is very useful if you habitually forget the name of a diskette microseconds after putting it into a disk drive.

The syntax of the ONLINE command will be

```
ONLINE [,S#] [,D#]
```

where the brackets mean that the specified parameter (slot number or drive number) is optional. If a specific slot or drive number is specified, then only the name of the volume for the corresponding disk device will be displayed. If both parameters are omitted, however, then the volume names for all disk devices will be displayed. The ONLINE command can be typed in while in Applesoft direct mode or it can be executed within a program using a PRINT CHR$(4);"ONLINE" statement.

The ONLINE program is shown in Table 5-8; it can be executed by using the BRUN command. The first part of the program is responsible for installing the image of the ONLINE command handler code that begins at $2100. It first finds a safe spot above HIMEM to store the image, patches it so that it will execute at this new position, and then moves the code to its new home. It also links in the command handler by storing its starting address at EXTRNCMD+1 ($BE07) and EXTRNCMD+2 ($BE08). And, just in case another user command handler has already been installed, it grabs the address previously stored in

EXTRNCMD + 1 and EXTRNCMD + 2 and stores it in the target address of a JMP instruction in the body of the ONLINE command handler. This JMP is executed only if the ONLINE handler doesn't recognize the command that is passed to it. This means that control will always daisy-chain down to a previously installed external command handler so that it will have a chance to claim the command.

The GETBUFR ($BEF5) subroutine has been used to locate a "safe" buffer large enough to store the command handler. It is called with the number of pages required in the accumulator (1). If we run out of room, the carry flag will be set and a "PROGRAM TOO LARGE" error message will be printed by calling ERROUT ($BE09). Otherwise, the first memory page in the block freed up will be returned in the accumulator. As we saw earlier in the chapter, we can now use this block to store a program without fear of it later being overwritten by file buffers or string variables.

Since the ONLINE command handler is not inherently relocatable, all references to internal absolute addresses must be altered to reflect the change in the position of the code. The relocation procedure is relatively simple in our example because the code for the command handler was assembled on a page boundary and it is being moved to another page boundary. This means that only the high-order part of each absolute address in the handler need be modified. Although it is possible to write a complex subroutine to patch the code automatically, we have chosen to patch it "manually" by inspecting the handler to identify addresses to be changed and then storing the new page number at these positions. If you change the handler in any way, you will have to re-determine which positions must be patched and make the necessary changes to the installation code.

The code is moved into place by making use of the system monitor block move subroutine, MOVE ($FEC2). This subroutine moves the block of memory beginning at the address stored in $3C/$3D and ending at the address stored in $3E/$3F to the block beginning at the address stored in $42/$43. MOVE must be called with the Y register set to zero.

The main part of the ONLINE command handler begins at CMDCODE. The first thing it does is check if the ASCII codes for the word "ONLINE" or "online" are at the beginning of the input buffer at $200 (intervening spaces are ignored). If not, then the carry flag is set (indicating "not handled") and the jump at NEXTCMD is executed; as explained above, this gives a previously installed command handler a crack at identifying the command.

If the "ONLINE" command is detected, the length of the command (minus 1) is stored at XLEN ($BE52), the external command number (0) is stored at XCNUM ($BE53), and the address of the post-parsing subroutine, EXECUTE, is stored at XTRNADDR ($BE50) and XTRNADDR + 1 ($BE51). Finally, the parsing rules are stored in PBITS ($BE54) and PBITS + 1 ($BE55): "pathname optional" and "slot/drive is allowed." The "pathname optional" bit must be set because the ONLINE command does not use a pathname. After the parsing rules have been set up, the carry flag is cleared ("everything OK") and an RTS returns control to BASIC.SYSTEM.

BASIC.SYSTEM then parses the command line in accordance with the instructions in PBITS, updates FBITS ($BE56) and FBITS + 1 ($BE57) to indicate

Table 5-8. Adding the ONLINE command to BASIC.SYSTEM.

```
1    ************************************
2    * BASIC.SYSTEM "ONLINE" COMMAND *
3    *                                  *
4    *       ONLINE [,Sn] [,Dn]         *
5    *                                  *
6    *   Copr. 1985 Gary B. Little      *
7    ************************************
8
9    SBLOCK    EQU  $3C        ;Parameters for block move
10   EBLOCK    EQU  $3E
11   FBLOCK    EQU  $42
12   HIMEM     EQU  $73        ;Use this as ONLINE buffer
13
14   IN        EQU  $200       ;Command input buffer
15
16   EXTRNCMD  EQU  $BE06      ;External command JMP
                                instruction
17   ERROUT    EQU  $BE09      ;Error handler
18   XTRNADDR  EQU  $BE50      ;Start of external cmd handler
19   XLEN      EQU  $BE52      ;External cmd name length (-1)
20   XCNUM     EQU  $BE53      ;Command # (0 for external)
21   PBITS     EQU  $BE54      ;Command parameter bits
22   FBITS     EQU  $BE56      ;Parameters found in parse
23   VSLOT     EQU  $BE61      ;Slot parameter specified
24   VDRIV     EQU  $BE62      ;Drive parameter specified
25   GETBUFR   EQU  $BEF5      ;Get a free space
26
27   MLI       EQU  $BF00      ;Entry point to MLI
28
29   CROUT     EQU  $FD8E      ;Print a CR
```

```
                        30  COUT     EQU  $FDED       ;Std. character output
                                                       subroutine
                        31  MOVE     EQU  $FE2C       ;Block move subroutine
                        32
                        33           ORG  $2000
                        34
                        35  * Calculate # of pages that we need to reserve:
                        36
2000: 38                37           SEC
2001: A9 22             38           LDA  #>END
2003: E9 21             39           SBC  #>CMDCODE
2005: 8D 74 20          40           STA  PAGES
2008: EE 74 20          41           INC  PAGES
                        42
200B: AD 74 20          43           LDA  PAGES       ;Reserve the pages for the
200E: 20 F5 BE          44           JSR  GETBUFR     ; command handler
2011: 90 05             45           BCC  INSTALL     ;Carry clear if OK
                        46
2013: A9 0E             47           LDA  #14         ;"PROGRAM TOO LARGE" error
2015: 4C 09 BE          48           JMP  ERROUT
                        49
2018: 8D 75 20          50  INSTALL  STA  PGSTART     ;Save starting page #
                        51
                        52  * Install the new command handler:
                        53
201B: AD 07 BE          54           LDA  EXTRNCMD+1 ;Set up link to
201E: 8D 26 21          55           STA  NEXTCMD+1  ; existing external command.
2021: AD 08 BE          56           LDA  EXTRNCMD+2
2024: 8D 27 21          57           STA  NEXTCMD+2
                        58
                        59  ****************************************
                        60  * Install the external command handler *
```

(continued)

Table 5-8. Adding the ONLINE command to BASIC.SYSTEM (continued).

```
                        61  * by storing its address after the     *
                        62  * JMP at EXTRNCMD.                     *
                        63  ****************************************
                        64
2027: A9 00             65          LDA  #0
2029: 8D 07 BE          66          STA  EXTRNCMD+1
202C: AD 75 20          67          LDA  PGSTART
202F: 8D 08 BE          68          STA  EXTRNCMD+2
                        69
                        70  * Relocate the code:
                        71
2032: AD 75 20          72          LDA  PGSTART    ;Get new page #
2035: 8D 0F 21          73          STA  CMDCODE+$0F
2038: 8D 1A 21          74          STA  CMDCODE+$1A
203B: 8D 32 21          75          STA  CMDCODE+$32
203E: 8D 49 21          76          STA  CMDCODE+$49
2041: 8D 4E 21          77          STA  CMDCODE+$4E
2044: 8D 55 21          78          STA  CMDCODE+$55
2047: 8D 6F 21          79          STA  CMDCODE+$6F
204A: 8D 75 21          80          STA  CMDCODE+$75
204D: 8D 8A 21          81          STA  CMDCODE+$8A
2050: 8D A4 21          82          STA  CMDCODE+$A4
2053: 8D D5 21          83          STA  CMDCODE+$D5
                        84
                        85  * Set up parameters for block move to final location:
                        86
2056: A9 00             87          LDA  #<CMDCODE
2058: 85 3C             88          STA  SBLOCK
205A: A9 21             89          LDA  #>CMDCODE
205C: 85 3D             90          STA  SBLOCK+1
                        91
```

```
205E: A9 03     92              LDA   #<END
2060: 85 3E     93              STA   EBLOCK
2062: A9 22     94              LDA   #>END
2064: 85 3F     95              STA   EBLOCK+1
                96
2066: A9 00     97              LDA   #0
2068: 85 42     98              STA   FBLOCK
206A: AD 75 20  99              LDA   PGSTART
206D: 85 43     100             STA   FBLOCK+1
                101
206F: A0 00     102             LDY   #0
2071: 4C 2C FE  103             JMP   MOVE        ;Move it!
                104
2074: 00        105 PAGES       DS    1           ;Length of command handler
2075: 00        106 PGSTART     DS    1           ;Starting page of command
                                                   handler
                107
2076: 00 00 00  108             DS    $2100-*     ;(Must start on page boundary)
2079: 00 00 00 00 00 00 00 00
2081: 00 00 00 00 00 00 00 00
2089: 00 00 00 00 00 00 00 00
2091: 00 00 00 00 00 00 00 00
2099: 00 00 00 00 00 00 00 00
20A1: 00 00 00 00 00 00 00 00
20A9: 00 00 00 00 00 00 00 00
20B1: 00 00 00 00 00 00 00 00
20B9: 00 00 00 00 00 00 00 00
20C1: 00 00 00 00 00 00 00 00
20C9: 00 00 00 00 00 00 00 00
20D1: 00 00 00 00 00 00 00 00
20D9: 00 00 00 00 00 00 00 00
20E1: 00 00 00 00 00 00 00 00
20E9: 00 00 00 00 00 00 00 00
```

(continued)

Table 5-8. Adding the ONLINE command to BASIC.SYSTEM (continued).

```
20F1: 00 00 00 00 00 00 00 00
                      109
                      110 CMDCODE  EQU  *
                      111
                      112 ************************************
                      113 * This is the command checker. It  *
                      114 * scans the input buffer to see    *
                      115 * if the command has been entered. *
                      116 ************************************
2100: D8              117          CLD
2101: A0 00           118          LDY  #0
2103: A2 00           119          LDX  #0
2105: BD 00 02        120 CHKCMD   LDA  IN,X        ;Get command character
2108: E8              121          INX
2109: C9 A0           122          CMP  #$A0        ;Is it a blank?
210B: F0 F8           123          BEQ  CHKCMD      ;If it is, ignore it
210D: D9 EE 21        124          CMP  CMDNAME,Y   ;Same as our command?
2110: F0 0B           125          BEQ  CHKCMD1     ;Yes, so branch
2112: C9 E0           126          CMP  #$E0        ;Lower case?
2114: 90 0E           127          BCC  NOTFOUND    ;No, so branch
2116: 29 DF           128          AND  #$DF        ;Convert to uppercase
2118: D9 EE 21        129          CMP  CMDNAME,Y   ;OK now?
211B: D0 07           130          BNE  NOTFOUND    ;No, so branch
211D: C8              131 CHKCMD1  INY
211E: C0 06           132          CPY  #CMDLEN-CMDNAME ;At end?
2120: D0 E3           133          BNE  CHKCMD      ;No, so branch
2122: F0 04           134          BEQ  SETRULES    ;Yes, so branch
                      135
2124: 38              136 NOTFOUND SEC              ;Set carry to indicate failure
2125: 4C 00 00        137 NEXTCMD  JMP  $0000       ;(Fill in when installed)
                      138
```

```
2128: 88        139 SETRULES DEY
2129: 8C 52 BE  140          STY  XLEN        ;Store command length-1
                141
212C: A9 51     142          LDA  #<EXECUTE   ;Put address of command
                                                handler
212E: 8D 50 BE  143          STA  XTRNADDR    ; into XTRNADDR.
2131: A9 21     144          LDA  #>EXECUTE
2133: 8D 51 BE  145      •   STA  XTRNADDR+1
                146
2136: A9 00     147          LDA  #0
2138: 8D 53 BE  148          STA  XCNUM       ;External cmd number = 0
                149
                150 * Set up string parsing rules:
                151
213B: A9 10     152          LDA  #$10        ;Pathname is optional
213D: 8D 54 BE  153          STA  PBITS
2140: A9 04     154          LDA  #$04        ;Slot/Drive allowed
2142: 8D 55 BE  155          STA  PBITS+1
                156
2145: A5 73     157          LDA  HIMEM       ;Set ON_LINE buffer (at least
2147: 8D EC 21  158          STA  BUFFER      ; 256 bytes) to free area
214A: A5 74     159          LDA  HIMEM+1     ; beginning at HIMEM.
214C: 8D ED 21  160          STA  BUFFER+1
                161
214F: 18        162          CLC              ;Clear carry to indicate
                                                 success
2150: 60        163          RTS
                164
                165 * BASIC.SYSTEM comes here after it has
                166 * successfully parsed the command line:
                167
2151: A9 00     168 EXECUTE  LDA  #0
```

(continued)

Table 5-8. Adding the ONLINE command to BASIC.SYSTEM (continued).

```
2153: 8D EB 21  169           STA  UNITNUM     ;(Assume all volumes)
2156: AD 57 BE  170           LDA  FBITS+1     ;Examine result of parse
2159: 29 04     171           AND  #$04        ;Slot/drive specified?
215B: F0 13     172           BEQ  DOCALL      ;No, so check everything
215D: AD 61 BE  173           LDA  VSLOT       ;Get slot # specified
2160: 0A        174           ASL
2161: 0A        175           ASL
2162: 0A        176           ASL
2163: 0A        177           ASL              ;Slot * 16
2164: AE 62 BE  178           LDX  VDRIV       ;Get drive # specified
2167: E0 02     179           CPX  #2          ;Drive 2?
2169: D0 02     180           BNE  SAVEUN      ;No, so branch
216B: 09 80     181           ORA  #$80        ;Set "drive 2" bit
216D: 8D EB 21  182 SAVEUN    STA  UNITNUM     ;Store slot/drive as unit num
                183
2170: 20 00 BF  184 DOCALL    JSR  MLI
2173: C5        185           DFB  $C5         ;ONLINE call
2174: EA 21     186           DA   OLPARM      ;Address of parm table
                187
2176: 20 8E FD  188           JSR  CROUT
2179: A0 00     189           LDY  #0
217B: 98        190 SCAN      TYA
217C: 48        191           PHA
217D: B1 73     192       •   LDA  (HIMEM),Y   ;Get slot/drive + length
217F: F0 61     193           BEQ  SCAN2       ;If $00 then all done
2181: 29 0F     194           AND  #$0F        ;Isolate length bits
2183: F0 4E     195           BEQ  NEXTNAME    ;If 0, then must be error
2185: 48        196           PHA

                197
2186: A2 00     198           LDX  #0
```

```
2188: BD F4 21  199 PRTMSG1  LDA  SLOTMSG,X  ;Print slot #
218B: F0 06     200          BEQ  PRTNUM1
218D: 20 ED FD  201          JSR  COUT
2190: E8        202          INX
2191: D0 F5     203          BNE  PRTMSG1
                204
2193: B1 73     205 PRTNUM1  LDA  (HIMEM),Y  ;Get slot/drive + length
2195: 29 70     206          AND  #$70       ;Isolate slot bits
2197: 4A        207          LSR
2198: 4A        208          LSR
2199: 4A        209          LSR
219A: 4A        210          LSR             ;We now have slot #
219B: 09 B0     211          ORA  #$B0       ;Convert to ASCII digit
219D: 20 ED FD  212          JSR  COUT
                213
21A0: A2 00     214          LDX  #0
21A2: BD FA 21  215 PRTMSG2  LDA  DRIVEMSG,X ;Print drive #
21A5: F0 06     216          BEQ  PRTNUM2
21A7: 20 ED FD  217          JSR  COUT
21AA: E8        218          INX
21AB: D0 F5     219          BNE  PRTMSG2
                220
21AD: A2 B1     221 PRTNUM2  LDX  #$B1       ;Assume drive 1
21AF: B1 73     222          LDA  (HIMEM),Y
21B1: 10 02     223          BPL  PSKIP      ;Branch if drive 1
21B3: A2 B2     224          LDX  #$B2       ;Must be drive 2
21B5: 8A        225 PSKIP    TXA
21B6: 20 ED FD  226          JSR  COUT
21B9: A9 BA     227          LDA  #":
21BB: 20 ED FD  228          JSR  COUT
21BE: A9 A0     229          LDA  #$A0
21C0: 20 ED FD  230          JSR  COUT
                231
```

(continued)

Table 5-8. Adding the ONLINE command to BASIC.SYSTEM (continued).

```
21C3: 68        232           PLA
21C4: AA        233           TAX
21C5: C8        234 PRTNAME   INY
21C6: B1 73     235           LDA  (HIMEM),Y   ;Get next character in name
21C8: 09 80     236           ORA  #$80        ;Set high bit
21CA: 20 ED FD  237           JSR  COUT        ; and display it
21CD: CA        238           DEX
21CE: D0 F5     239           BNE  PRTNAME     ;Branch until done
21D0: 20 8E FD  240           JSR  CROUT
                241
21D3: AD EB 21  242 NEXTNAME  LDA  UNITNUM     ;Was only one volume
                                                 specified?
21D6: D0 0A     243           BNE  SCAN2       ;Yes, so branch
                244
21D8: 68        245           PLA
21D9: 18        246           CLC
21DA: 69 10     247           ADC  #16         ;Move to next name
21DC: A8        248           TAY
21DD: C0 E0     249           CPY  #224        ;At end of table?
21DF: D0 9A     250           BNE  SCAN        ;No, so branch
21E1: 48        251           PHA
                252
21E2: 68        253 SCAN2     PLA
21E3: 20 8E FD  254           JSR  CROUT
21E6: 18        255           CLC              ;CLC ==> no error
21E7: A9 00     256           LDA  #0          ;Error code = 0
21E9: 60        257           RTS
                258
21EA: 02        259 OLPARM    DFB  2           ;Two parameters
21EB: 00        260 UNITNUM   DFB  0           ;Unit number (DSSS0000)
21EC: 00 00     261 BUFFER    DA   $0000       ;Device buffer
```

```
                      262
21EE: CF CE CC        263 CMDNAME   ASC  "ONLINE"    ;External command name
21F1: C9 CE C5
                      264 CMDLEN    EQU *
                      265
21F4: D3 CC CF        266 SLOTMSG   ASC  "SLOT ",00
21F7: D4 A0 00
21FA: AC A0 C4        267 DRIVEMSG  ASC  ", DRIVE ",00
                      268
                      269 END       EQU *
```

the results of the parse, and then jumps to EXECUTE (its address was previously stored in XTRNADDR).

EXECUTE examines FBITS to see if a specific slot/drive was specified. If so, then the slot and drive specified are retrieved from VSLOT ($BE61) and VDRIV ($BE62) and used to form the unit number required by the MLI ON_LINE command. If not, the unit number is set to 0; this indicates to the MLI that all volumes are to be examined.

Once the MLI ON_LINE command has been executed, the names of the active volumes will be stored in the buffer beginning at HIMEM. (See the Chapter 4 discussion of ON_LINE for a description of the structure of this buffer.) The volume names are then extracted from the buffer and displayed in the following format:

```
SLOT 6, DRIVE 1: TEST.VOLUME
```

Chapter 6

INTERRUPTS

In this chapter we will be describing how 6502 IRQ (interrupt request) interrupts are handled in a ProDOS environment. ProDOS allows up to four independent assembly language subroutines to be installed to service these interrupts and defines a standard protocol that these subroutines must use so that they will function smoothly together. This protocol relates to the method to be used to indicate whether a particular subroutine serviced the interrupt or not.

Before we begin, however, we had better review the concept of an interrupt. An interrupt is really just a special electrical signal that an I/O device can send to the 6502 microprocessor in an attempt to get its immediate and undivided attention. These IRQ signals are sent down a special line that is connected between a specific pin on each expansion slot and the IRQ pin on the 6502 integrated circuit package. (On the slotless Apple //c equivalent connections are made between the IRQ sources of each built in I/O device and the 65C02 microprocessor.)

Active interrupt signals are typically generated when an I/O device has new data to be read or is ready to receive more data. When the 6502 detects an active IRQ signal, it stops executing the main program after completing the current instruction and starts executing a special interrupt-handling subroutine that has been installed. This subroutine is responsible for servicing the interrupt by clearing the condition that caused the interrupt and performing the necessary I/O operation. When it finishes, control returns to the main program at the point where it was interrupted, and execution of that program continues as if nothing had happened.

The advantage of using an interrupt scheme such as this to control I/O devices is that the scheme is the most efficient one available to handle asyn-

chronous I/O operations (that is, operations that can occur at any time). If such a scheme were not available, a program would be forced to waste a lot of time while it monitored (or "polled") each I/O device in the system very frequently in order to ensure that incoming data was not lost or that outgoing data was being pushed out as quickly as possible. This is roughly comparable to picking up a telephone without a bell every few seconds to see if anyone is calling in. By adding the bell (the interrupt signal) you can go about your normal duties until the bell rings (an active interrupt signal occurs) and then you can pick up the telephone (service the interrupt).

Common Interrupt Sources

Many I/O devices available for the Apple II are capable of generating interrupts. Let's look at the sources of interrupts that are usually available on three of the most common I/O devices: the clock, the asynchronous serial interface, and the mouse.

CLOCK. A clock card is a device capable of keeping track of the time and date without the assistance of the 6502 microprocessor (the logic is handled by a discrete integrated circuit). It typically contains a small battery that allows the clock to keep track of the time even when the Apple is turned off. Most clock cards are capable of generating interrupts at regular intervals: every second, minute, or hour.

ASYNCHRONOUS SERIAL INTERFACE. An asynchronous serial interface is most commonly used to link the Apple to printers and modems. It can usually be instructed to generate interrupts whenever it is ready to send out a character or when a character has been received.

MOUSE. A mouse is an input device that is normally capable of generating interrupts when it is moved or when its button is pressed.

What Happens When an Active IRQ Signal Occurs

It is very important to realize that the IRQ interrupt signal is "maskable." This means that it is possible to instruct the 6502 microprocessor to ignore an active IRQ interrupt signal (that is, to mask it) if you so wish; this is done by executing an SEI (set interrupt disable flag) instruction. (The interrupt disable flag is a bit in the 6502 processor status register.) If interrupts are disabled in this way, the main program running in the system will never be disturbed. Time-critical operations, such as disk reads and writes, cannot be interrupted without loss of data, so interrupts are always disabled first.

The corresponding instruction that causes the 6502 to respond to interrupts is CLI (clear interrupt disable flag). In the discussion that follows, we will assume that the interrupt disable flag has been cleared.

When a device generates an active IRQ signal, the 6502 responds by first completing the current instruction being executed. The following sequence of events then takes place:

- The current program counter is stored on the stack. This is the address of the next instruction in the program to be executed after the interrupt has been dealt with.
- The break flag in the processor status register is cleared to 0 and then the status register is stored on the stack.
- The interrupt disable flag in the processor status register is set to 1. (This prevents the 6502 from responding to other active IRQ signals that may occur while the current interrupt is being serviced. It is possible to re-enable interrupts explicitly within the interrupt-handling subroutine, however.)

Once these operations have been performed, the 6502 program counter is loaded with the address stored in the "6502 IRQ vector" at $FFFE/$FFFF (low-order byte first) and then control passes to that address. The address stored in the IRQ vector will depend on what Apple II system you are using and on whether the Apple's bank-switched RAM memory is read-enabled when the interrupt occurs. (And it will be if ProDOS is executing an MLI command when the interrupt occurs.)

The IRQ vector normally points to a subroutine that is responsible for determining whether the source of the interrupt was an IRQ signal or a BRK instruction. (BRK is a 6502 instruction that simulates an IRQ interrupt condition.) The subroutine does this by inspecting the state of the break flag in the 6502 status register. If the source was an IRQ signal (that is, the break flag is 0), then control will pass to the address stored at a user-definable interrupt vector located at $3FE and $3FF (low-order byte first). Control eventually returns to the main program when a RTI (return from interrupt) instruction is executed. This instruction pops the three bytes on top of the stack directly into the status register and program counter.

It should be clear, then, that in order to properly handle an IRQ interrupt, an appropriate interrupt-handling subroutine must be placed in memory and its starting address must be stored at $3FE/$3FF. It is very important, of course, that this subroutine be installed BEFORE instructions are executed that permit the I/O device to start generating interrupts.

A properly designed interrupt-handling subroutine must perform the following chores in the following order:

- Save the current values of the A, X, and Y registers.
- Clear the source of the interrupt. (This is usually done by reading the status register of the I/O device.)
- Service the interrupt by performing the I/O operation required.
- Restore the A, X, and Y registers to their initial values.
- End the subroutine with an RTI (return from interrupt) instruction.

Here is the shell of a properly constructed interrupt-handling subroutine that will work on the Apple II:

```
TXA
PHA
TYA
PHA
;
[service the
 interrupt]
;
PLA
TAY
PLA
TAX
LDA $45
RTI
```

The Apple firmware automatically places the value of the accumulator at location $45 when an interrupt occurs and the values of the X and Y registers are conveniently saved on the stack with push instructions as indicated. Just before the subroutine ends with an RTI, the X and Y registers are restored by popping their original values back from the stack and the accumulator is restored from $45. (By the way, the use of location $45 does not interfere with the operation of ProDOS; in a DOS 3.3 environment, however, the system can crash when an interrupt occurs because DOS 3.3 uses $45 for data storage.)

The ProDOS Interrupt-Handling Subroutine

ProDOS automatically installs its own general-purpose interrupt-handling subroutine when it is first loaded into memory. You can determine its starting address by examining the address stored in the IRQ user vector at $3FE/$3FF. Unfortunately, the subroutines used in the first versions of ProDOS (1.1 and earlier) did not work properly; the latest version (1.1.1) is much better, but at least one bug still exists (as we will see later on). The moral is always to use the most current version of ProDOS if you are using interrupts.

The ProDOS subroutine is different from most interrupt handlers, however, in that it contains no specific code to identify and service an interrupt. (This isn't too surprising since it could hardly be expected to support every possible source of interrupts.) To service an interrupt it relies on a table containing the addresses of up to four user-installed subroutines; these subroutines are integrated into ProDOS using the MLI ALLOC_INTERRUPT ($40) command. This command stores the address of a subroutine at the next available location in an eight-byte interrupt vector table in the ProDOS global page beginning at $BF80. (A dummy $0000 address is stored in the table if a vector is unused.) A list of all the global page locations used by the ProDOS interrupt-handling subroutine are shown in Table 6-1.

The chain of events when an interrupt occurs is diagrammed in Figure 6-1. The ProDOS interrupt handler takes over and calls the subroutine whose

Table 6-1. Global Page Data Areas Used by the ProDOS Interrupt-Handling Subroutine.

Address	*Symbolic Label*	*Description*
$BF80	INTRUPT1	The address of the first user-installed interrupt subroutine
$BF82	INTRUPT2	The address of the second user-installed interrupt subroutine
$BF84	INTRUPT3	The address of the third user-installed interrupt subroutine
$BF86	INTRUPT4	The address of the fourth user-installed interrupt subroutine
$BF88	INTAREG	The "A" register is stored here when an interrupt occurs
$BF89	INTXREG	The "X" register is stored here when an interrupt occurs
$BF8A	INTYREG	The "Y" register is stored here when an interrupt occurs
$BF8B	INTSREG	The stack pointer is stored here when an interrupt occurs
$BF8C	INTPREG	The processor status register is stored here when an interrupt occurs
$BF8D	INTBANKID	The identification code for the active $Dx bank is stored here when an interrupt occurs
$BF8E	INTADDR	The address of the instruction being executed when an interrupt occurred is stored here when an interrupt occurs

address is stored in the first entry in the interrupt vector table. This subroutine will either recognize and claim the interrupt or not; if it does, then ProDOS restores all registers and returns to the interrupted program. If it doesn't, ProDOS will try again by calling the next subroutine stored in the interrupt vector table. This process is repeated until the interrupt is claimed. (If none of the installed subroutines claims the interrupt, a critical error will occur and you will have to reboot the system.)

The main advantage of using such a scheme to handle interrupts is that it allows for the development of interrupt-handling subroutines that are specific to one device only. That is, a subroutine need not concern itself with handling mouse, clock, serial, and "you-name-it" interrupts all at once. If the ProDOS protocol is used, you can easily install a mouse interrupt subroutine from one manufacturer and a clock interrupt subroutine from another and they should work properly together. Furthermore, you can also easily establish the priority of interrupts by installing the most time-critical subroutine first.

In order for this system to work properly, a user-installed ProDOS interrupt subroutine must adhere to the following protocol:

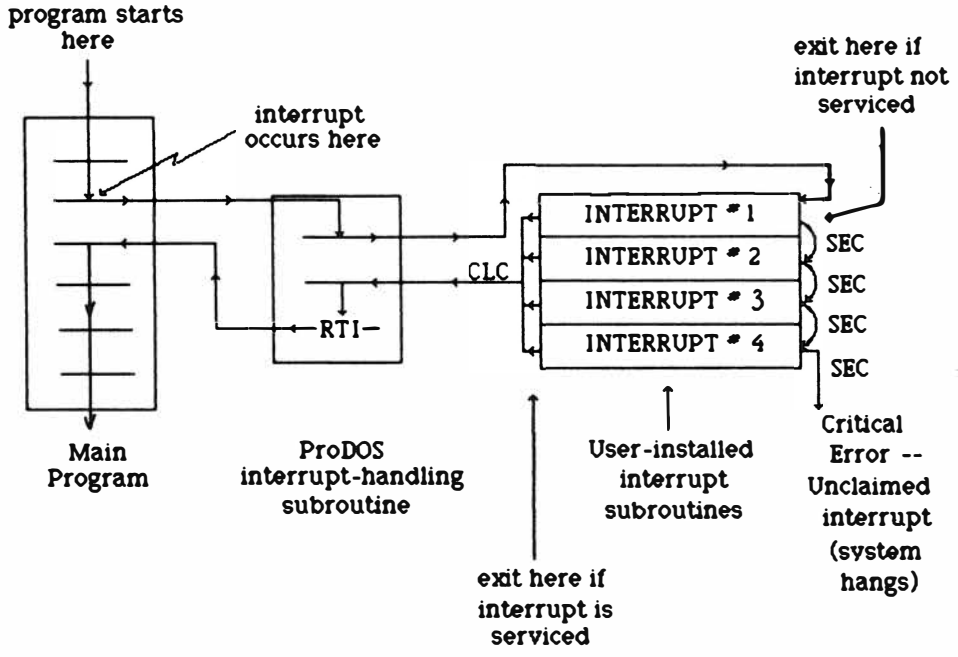


Figure 6-1. How ProDOS handles interrupts

- The first instruction must be CLD.
- If the interrupt is claimed, the carry flag must be cleared (with a CLC instruction) before exit.
- If the interrupt is not claimed, the carry flag must be set (with an SEC instruction) before exit.
- The subroutine must exit with all soft switches in the states they were in upon entry. Most of these switches are used for memory bank switching or for controlling video display modes. (See Appendix III of *Inside the Apple //e.*)
- The subroutine must end with an RTS instruction (*NOT* an RTI instruction). The ProDOS subroutine itself will execute the necessary RTI instruction.

Note that there is no need for such a subroutine to save and restore the 6502 registers. This chore is automatically performed by the main ProDOS interrupt-handling subroutine. Two other nice features of the ProDOS subroutine that significantly simplify the writing of an interrupt subroutine are:

- The contents of locations $FA-$FF are saved before control passes to your interrupt subroutine and are restored when you're through. This frees up seven valuable zero page locations for unrestricted use by your subroutine.

- At least 16 bytes of stack space are freed up before your interrupt subroutine gets control. This should be sufficient for even the most complicated subroutines.

The program in Table 6-2 (MOUSE.MOVE) shows how to properly install an interrupt-handling subroutine in a ProDOS environment. To be able to run this specific example, you must be using an Apple //c with the Apple Mouse option or an Apple //e (or II Plus) with an Apple Mouse card installed in slot 4.

MOUSE.MOVE commands the mouse to generate interrupts whenever it is rolled across a tabletop. When the mouse is moved, the interrupt handler identifies the mouse as the source of the interrupt and then prints the letter "M" on the screen. All this happens more or less invisibly to the main program that is running; it is just slowed down by the time it takes to service the interrupt.

The first thing that MOUSE.MOVE does is install the address of the interrupt handler (IRQHNDL) in the ProDOS interrupt vector table by executing the MLI ALLOC_INTERRUPT ($40) command. If an error occurs, the program will branch to ERROR and enter the system monitor. (An error will only occur if the interrupt vector table is full.) Otherwise, the next step is to initialize the mouse and enable mouse movement interrupts by sending a mouse mode code of 3 to a subroutine called SETMOUSE. The address of this subroutine, and all other standard mouse subroutines, begin somewhere in the mouse interface's firmware in page $C4; the exact offset for each subroutine is stored in a table beginning at location $C412. The offset for SETMOUSE is the zeroth entry in this table; the offsets for the other mouse subroutines that are used are indicated at the beginning of the program.

MOUSER is the standard subroutine that is called to execute all mouse subroutines. It is responsible for setting up the correct subroutine address and placing the correct numbers in the 6502 registers before passing control to the mouse firmware.

When the mouse is moved, an interrupt will occur and ProDOS will quickly call IRQHNDL. This subroutine first does what all good interrupt handlers should: it determines whether the interrupt was caused by the expected source (that is, mouse movement). In the case of the Apple Mouse, this determination is made by calling the SERVEMOUSE subroutine. If the carry flag is set, then something else must have caused the interrupt and the subroutine ends with the carry flag set.

If the interrupt was caused by movement of the mouse, then the interrupt is immediately serviced by displaying the letter "M" on the screen by calling COUT ($FDED), the standard character output subroutine. Before exiting the subroutine, the carry flag is cleared so that ProDOS will know that the interrupt was serviced.

You must remember to perform one important step before calling COUT (or any other system monitor or Applesoft subroutine) however: read-enable the ROM area from $D000 to $FFFF. This is done by reading $C082, the soft-switch that disables bank- switched RAM. This step is necessary because bank-switched RAM (which is where ProDOS resides) is always read-enabled when the interrupt subroutine takes over and so the ROM that shares the same address

space is not available. If you do throw the $C082 switch, you must later re-enable the ProDOS bank-switched RAM (which includes bank1 of the $Dx bank) for reading and writing by reading from $C08B twice in succession.

Interrupt subroutines can be removed from ProDOS by using the MLI DEALLOC_INTERRUPT ($41) command. Before doing this, however, you must ensure that interrupts are disabled on the I/O device. Notice how this is done in MOUSE.MOVE. When the program is entered at $303, control passes to the DISABLE subroutine. This subroutine first turns off mouse interrupts by sending the appropriate mode code (0) to SETMOUSE, and then removes the address of the mouse interrupt handler from the ProDOS interrupt vector table by calling the DEALLOC_INTERRUPT ($41) command. (The interrupt code number is already in the parameter table from the previous ALLOC_INTERRUPT ($40) call.)

Interrupts During MLI Calls

The ProDOS interrupt scheme just described works perfectly well in most situations. Special changes must be made, however, if an MLI command is to be executed as part of the interrupt-handling process. But note that the modified scheme we are about to describe will not work with ProDOS version 1.1.1 (or earlier versions) because of a bug in the ProDOS interrupt-handling subroutine. This bug will (we hope) be eradicated in the next release of ProDOS.

It's easy to see why changes are necessary. Consider a situation in which an interrupt occurs when the main program is in the middle of executing an MLI command. In a typical situation, the MLI command will have stored important information in an MLI data area that is used by all MLI commands. If another MLI command were executed at this time, this data area might be overwritten, causing unpredictable behavior when the first MLI command regained control. We must ensure, then, that an interrupt subroutine does not make MLI calls while another MLI call is pending.

To avoid this potentially disastrous situation, every interrupt subroutine that makes MLI calls must first examine MLIACTV ($BF9B) to see if an MLI command is currently active. Recall from Chapter 4 that bit 7 of MLIACTV is normally 0, but is set to 1 whenever an MLI command is called (using a "JSR $BF00" instruction).

This means that if bit 7 of MLIACTV is 0, the interrupt can be processed normally.

If bit 7 is 1, however, then an MLI call is in progress and the MLI call to be made by the interrupt handler must be deferred until the current call has finished. Here's what an interrupt subroutine must do in order to achieve this result:

- Clear the hardware interrupt condition.
- Take the address stored at CMDADR ($BF9C/$BF9D) and put it in a safe two-byte area. (As we saw in Chapter 4, CMDADR holds the address of the instruction that receives control after a "JSR MLI" instruction is executed.)

Table 6-2. MOUSE.MOVE, a program that handles mouse movement interrupts.

```
                        1   ************************************
                        2   *            MOUSE.MOVE            *
                        3   * Mouse Movement Interrupt Handler *
                        4   ************************************
                        5
                        6   MLI     EQU  $BF00   ;Entry point to ProDOS MLI
                        7
                        8   MTABLE  EQU  $C412   ;Start of mouse ROM table
                        9
                        10  * Mouse subroutine numbers:
                        11  SETM    EQU  0       ;Set mouse mode
                        12  SERVEM  EQU  1       ;Service mouse interrupt
                        13  CLEARM  EQU  3       ;Clear mouse position
                        14  INITM   EQU  7       ;Initialize the mouse
                        15
                        16  COUT    EQU  $FDED   ;Standard output
                        17
                        18          ORG  $300
                        19
0300: 4C 06 03          20          JMP  ENABLE
0303: 4C 27 03          21          JMP  DISABLE
                        22
                        23  * Install the interrupt handler:
                        24
0306: 78                25  ENABLE  SEI
0307: A9 02             26          LDA  #2
0309: 8D 3E 03          27          STA  AIPARMS ;Stuff correct parm count
030C: 20 00 BF          28          JSR  MLI
030F: 40                29          DFB  $40     ;ALLOC_INTERRUPT
0310: 3E 03             30          DA   AIPARMS
0312: B0 2E             31          BCS  ERROR
```

(continued)

Table 6-2. MOUSE.MOVE, a program that handles mouse movement interrupts (continued).

```
                        32
                        33  * Prepare the mouse:
                        34
0314: A2 07             35          LDX   #INITM
0316: 20 5B 03          36          JSR   MOUSER   ;Initialize the mouse
                        37
0319: A2 00             38          LDX   #SETM
031B: A9 03             39          LDA   #$03     ;(Movement interrupt mode)
031D: 20 5B 03          40          JSR   MOUSER   ;Set the mouse mode
                        41
0320: A2 03             42          LDX   #CLEARM
0322: 20 5B 03          43          JSR   MOUSER   ;Clear mouse position
                        44
0325: 58                45          CLI            ;Enable 6502 interrupts
0326: 60                46          RTS
                        47
                        48  * Here's the code to "remove" the interrupt:
0327: 78                49  DISABLE SEI
0328: A2 00             50          LDX   #SETM
032A: A9 00             51          LDA   #0       ;(Turn mouse off)
032C: 20 5B 03          52          JSR   MOUSER
                        53
032F: A9 01             54          LDA   #1
0331: 8D 3E 03          55          STA   AIPARMS  ;Stuff correct parm count
0334: 20 00 BF          56          JSR   MLI      ;(Remove interrupt from ProDOS)
0337: 41                57          DFB   $41      ;DEALLOC_INTERRUPT
0338: 3E 03             58          DA    AIPARMS
033A: B0 06             59          BCS   ERROR
033C: 58                60          CLI
033D: 60                61          RTS
                        62
```

```
033E: 00        63  AIPARMS DS   1       ;# of parms
033F: 00        64        DS   1       ;Interrupt code # stored here
0340: 43 03     65        DA   IRQHNDL ;Address of handler
                66
0342: 00        67  ERROR   BRK
                68
                69  *********************************
                70  * Here's the interrupt handler *
                71  *********************************
0343: D8        72  IRQHNDL CLD
0344: A2 01     73        LDX  #SERVEM
0346: 20 5B 03  74        JSR  MOUSER  ;Check for mouse interrupt
0349: B0 0F     75        BCS  IRQEXIT ;Branch if it isn't
                76
034B: AD 82 C0  77        LDA  $C082   ;Enable monitor ROMs
034E: A9 CD     78        LDA  #$CD
0350: 20 ED FD  79        JSR  COUT    ;Display "M"
0353: AD 8B C0  80        LDA  $C08B
0356: AD 8B C0  81        LDA  $C08B   ;R/W enable bank1 of BSR
                82
0359: 18        83        CLC
035A: 60        84  IRQEXIT RTS
                85
                86  ****************************************
                87  * MOUSER executes the mouse subroutine *
                88  * specified by the code in the X       *
                89  * register.                            *
                90  ****************************************
035B: 48        91  MOUSER  PHA
035C: BD 12 C4  92        LDA  MTABLE,X ;Get low byte of subroutine addr
035F: 8D 7F 03  93        STA  MOUSE    ; and set up for indirect JMP.
0362: 68        94        PLA
```

(continued)

Table 6-2. MOUSE.MOVE, a program that handles mouse movement interrupts (continued).

```
0363: EA        95            NOP
0364: 78        96            SEI           ;Interrupts off for this!!
0365: 8E 76 03  97            STX   XSAVE
0368: 8C 77 03  98            STY   YSAVE
036B: 20 78 03  99            JSR   DOMOUSE ;Execute subroutine
036E: AC 77 03  100           LDY   YSAVE
0371: AE 76 03  101           LDX   XSAVE
0374: 58        102           CLI
0375: 60        103           RTS
                104
0376: 00        105 XSAVE     DS    1
0377: 00        106 YSAVE     DS    1
                107
0378: A2 C4     108 DOMOUSE   LDX   #$C4     ;(Mouse in slot 4)
037A: A0 40     109           LDY   #$40
037C: 6C 7F 03  110           JMP   (MOUSE)
                111
037F: 00        112 MOUSE     DS    1         ;Subroutine address (low)
0380: C4        113           DFB   $C4       ;(High part is always $C4)
```

- Replace CMDADR with the address of the portion of the interrupt handler that makes the MLI call.
- Clear the carry flag (CLC) and finish with RTS.

After these steps have been performed, control will not return to the main program when an interrupt occurs, but rather to the portion of the interrupt handler that makes the MLI call (that is, the new address that was stored in CMDADR). Once the MLI call has been made, the interrupt handler passes control to the address that was originally stored in CMDADR, thus completing the interrupt cycle.

For this procedure to work properly, the re-entrant portion of the interrupt subroutine that makes the MLI call must preserve the value of the status register and the A, X, and Y registers, and must end with a JMP to the old CMDADR. Here is what such a subroutine looks like:

```
PHP
PHA
TYA
PHA
TXA
PHA
;
[make MLI call]
;
PLA
TAX
PLA
TAY
PLA
PLP
JMP (OLDADR)
```

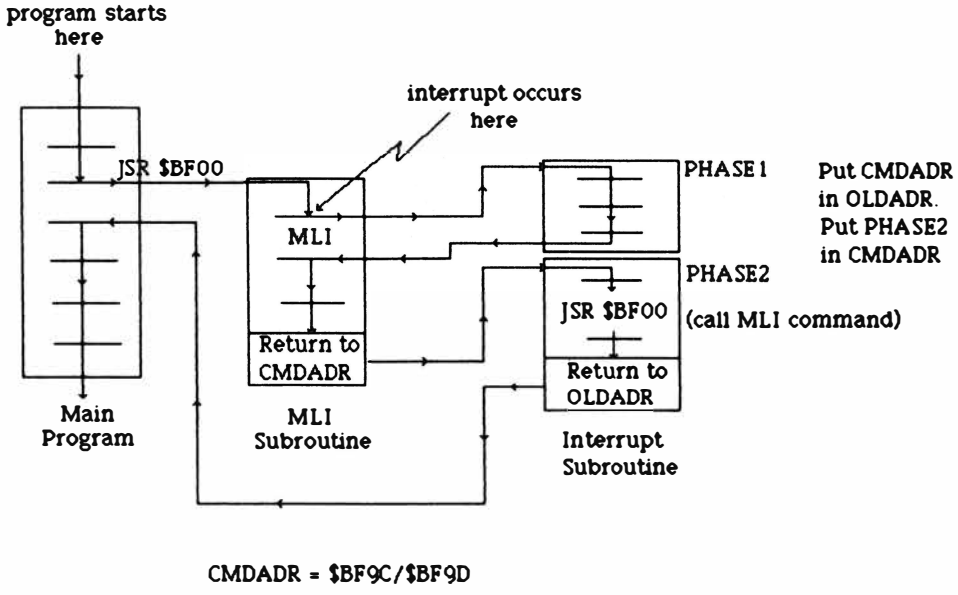
OLDADR is simply the address at which the original address in CMDADR is stored.

This procedure may seem a little confusing at first. The diagram in Figure 6-2 should help to clarify the program flow, however.

Let's also clarify how to deal with the MLI problem by examining the BUTTON.TIME program in Table 6-3. This program enables button interrupts on a mouse and handles such interrupts by reading the current time (making the GET_TIME MLI call) and displaying it on the screen. Once BUTTON.TIME has been installed, the current time will always be at your fingertips.

As usual, the first thing the interrupt handler does is to verify that the source of the interrupt is as expected. If it is, then the state of bit 7 of MLIACTV is tested using a BIT instruction. If no MLI command is active, then bit 7 will be 0 and the interrupt can be serviced right away by calling the MLI GET_TIME ($82) command and then displaying the date.

If an MLI command is active, then bit 7 will be 1 and the BMI branch will transfer control to SWAPADR. SWAPADR takes the current address stored in CMDADR and stores it in OLDADR and then places the address of PHASE2 in



Note: CMDADR initially contains the address in the main program to which control is to pass after the "JSR $BF00" instruction is executed.

Figure 6-2. Handling interrupts during MLI calls

CMDADR before clearing the carry flag and exiting. This means that when the current MLI command ends, PHASE2 will take over and the MLI GET _ TIME ($82) command will be executed. The time is then retrieved from TIME ($BF92 and $BF93), converted to ASCII digits, and displayed on the screen. Finally, a "JMP (OLDADR)" is executed to return control to the main program.

Table 6-3. BUTTON.TIME, a program to illustrate how to handle interrupts during MLI calls.

```
                        1   ***************
                        2   * BUTTON.TIME *
                        3   ***************
                        4
                        5   MLI       EQU   $BF00     ;Entry point to ProDOS MLI
                        6
                        7   MINUTES   EQU   $BF92     ;ProDOS minutes
                        8   HOURS     EQU   $BF93     ;ProDOS hours
                        9
                        10  MLIACTV   EQU   $BF9B     ;>=$80 if MLI busy
                        11  CMDADR    EQU   $BF9C     ;Return address of caller to MLI
                        12
                        13  HEXDEC    EQU   $ED24     ;Print X/A as decimal number
                        14
                        15  MTABLE    EQU   $C412     ;Start of mouse ROM table
                        16
                        17  * Mouse subroutine numbers:
                        18  SETM      EQU   0         ;Set mouse mode
                        19  SERVEM    EQU   1         ;Service mouse interrupt
                        20  CLEARM    EQU   3         ;Clear mouse position
                        21  INITM     EQU   7         ;Initialize the mouse
                        22
                        23  COUT      EQU   $FDED     ;Standard output
                        24
                        25            ORG   $300
                        26
                        27  * Install the interrupt handler:
                        28
0300: 78                29            SEI             ;Disable interrupts for this
0301: 20 00 BF          30            JSR   MLI
0304: 40                31            DFB   $40       ;ALLOC_INTERRUPT
```

(continued)

Table 6-3. BUTTON.TIME, a program to illustrate how to handle interrupts during MLI calls (continued).

```
0305: 1C 03     32            DA    AIPARMS
0307: B0 17     33            BCS   ERROR
                34
                35  * Prepare the mouse:
                36
0309: A2 07     37            LDX   #INITM
030B: 20 9B 03  38            JSR   MOUSER    ;Initialize the mouse
                39
030E: A2 00     40            LDX   #SETM
0310: A9 05     41            LDA   #$05      ;(Button interrupt mode)
0312: 20 9B 03  42            JSR   MOUSER    ;Set the mouse mode
                43
0315: A2 03     44            LDX   #CLEARM
0317: 20 9B 03  45            JSR   MOUSER    ;Clear mouse position
                46
031A: 58        47            CLI             ;Enable 6502 interrupts
031B: 60        48            RTS
                49
031C: 02        50  AIPARMS   DFB   2         ;# of parms
031D: 00        51            DS    1         ;Interrupt code # stored here
031E: 21 03     52            DA    IRQHNDL   ;Address of handler
                53
0320: 00        54  ERROR     BRK
                55
                56  ********************************
                57  * Here's the interrupt handler *
                58  ********************************
0321: D8        59  IRQHNDL   CLD
0322: A2 01     60            LDX   #SERVEM
0324: 20 9B 03  61            JSR   MOUSER    ;Check for mouse interrupt
0327: 90 01     62            BCC   IRQH1     ;Branch if it is
```

```
0329: 60        63            RTS
                64
032A: 2C 9B BF  65  IRQH1     BIT  MLIACTV  ;In middle of MLI call?
032D: 30 52     66            BMI  MLIWAIT  ;Yes, so branch
                67
032F: AD 82 C0  68            LDA  $C082    ;Enable monitor ROMs
0332: 20 4F 03  69            JSR  SHOWTIME
0335: AD 8B C0  70            LDA  $C08B
0338: AD 8B C0  71            LDA  $C08B    ;R/W enable bank1 of BSR
                72
033B: 18        73            CLC           ;(IRQ was serviced)
033C: 60        74            RTS
                75
                76  ********************************************
                77  * This is the 'deferred' interrupt handler *
                78  ********************************************
033D: 08        79  PHASE2     PHP           ;Save all registers first
033E: 48        80            PHA
033F: 8A        81            TXA
0340: 48        82            PHA
0341: 98        83            TYA
0342: 48        84            PHA
                85
0343: 20 4F 03  86            JSR  SHOWTIME
                87
0346: 68        88            PLA           ;Restore all registers
0347: A8        89            TAY
0348: 68        90            PLA
0349: AA        91            TAX
034A: 68        92            PLA
034B: 28        93            PLP
                94
034C: 6C 99 03  95            JMP  (OLDADR)
```

(continued)

Table 6-3. BUTTON.TIME, a program to illustrate how to handle interrupts during MLI calls (continued).

```
                        96
                        97  ********************************
                        98  * Read the time and print it *
                        99  * as HH:MM.                    *
                       100  ********************************
034F: 20 00 BF         101 SHOWTIME JSR  MLI
0352: 82               102          DFB  $82       ;GET_TIME
0353: 00 00            103          DA   $0000
                       104
0355: AE 93 BF         105          LDX  HOURS
0358: E0 0A            106          CPX  #10       ;10 or greater?
035A: B0 05            107          BCS  ST1       ;Yes, so branch
035C: A9 B0            108          LDA  #$B0
035E: 20 ED FD         109          JSR  COUT      ;Print leading zero
0361: A9 00            110 ST1      LDA  #0
0363: 20 24 ED         111          JSR  HEXDEC    ;Print HOURS
0366: A9 BA            112          LDA  #$BA
0368: 20 ED FD         113          JSR  COUT      ;Print a colon
                       114
036B: AE 92 BF         115          LDX  MINUTES
036E: E0 0A            116          CPX  #10       ;10 or greater?
0370: B0 05            117          BCS  ST2       ;Yes, so branch
0372: A9 B0            118          LDA  #$B0
0374: 20 ED FD         119          JSR  COUT      ;Print leading zero
0377: A9 00            120 ST2      LDA  #0
0379: 20 24 ED         121          JSR  HEXDEC    ;Print MINUTES
037C: A9 8D            122          LDA  #$8D
037E: 4C ED FD         123          JMP  COUT
                       124
                       125 ************************************************
                       126 * We now handle the case where an interrupt    *
```

```
                    127 * occurs during an MLI call. The address      *
                    128 * stored at CMDADR is saved and replaced by   *
                    129 * the address of PHASE2.                       *
                    130 ***********************************************
0381: AD 9C BF      131 MLIWAIT  LDA  CMDADR
0384: 8D 99 03      132          STA  OLDADR
0387: AD 9D BF      133          LDA  CMDADR+1
038A: 8D 9A 03      134          STA  OLDADR+1
038D: A9 3D         135          LDA  #<PHASE2
038F: 8D 9C BF      136          STA  CMDADR
0392: A9 03         137          LDA  #>PHASE2
0394: 8D 9D BF      138          STA  CMDADR+1
0397: 18            139          CLC            ;("Interrupt handled")
0398: 60            140          RTS
                    141
0399: 00 00         142 OLDADR   DS   2
                    143
                    144 ***************************************
                    145 * MOUSER executes the mouse subroutine *
                    146 * specified by the code in the X       *
                    147 * register.                            *
                    148 ***************************************
039B: 48            149 MOUSER   PHA
039C: BD 12 C4      150          LDA  MTABLE,X ;Get low byte of subroutine addr
039F: 8D BF 03      151          STA  MOUSE    ; and set up for indirect JMP.
03A2: 68            152          PLA
03A3: EA            153          NOP
03A4: 78            154          SEI            ;Interrupts off for this!!
03A5: 8E B6 03      155          STX  XSAVE
03A8: 8C B7 03      156          STY  YSAVE
03AB: 20 B8 03      157          JSR  DOMOUSE  ;Execute subroutine
03AE: AC B7 03      158          LDY  YSAVE
03B1: AE B6 03      159          LDX  XSAVE
```

(continued)

Table 6-3. BUTTON.TIME, a program to illustrate how to handle interrupts during MLI calls (continued).

```
03B4: 58          160             CLI
03B5: 60          161             RTS
                  162
03B6: 00          163 XSAVE       DS    1
03B7: 00          164 YSAVE       DS    1
                  165
03B8: A2 C4       166 DOMOUSE     LDX   #$C4      ;(Mouse in slot 4)
03BA: A0 40       167             LDY   #$40
03BC: 6C BF 03    168             JMP   (MOUSE)
                  169
03BF: 00          170 MOUSE       DS    1         ;Subroutine address (low)
03C0: C4          171             DFB   $C4       ;(High part is always $C4)
```

Chapter 7

DISK DRIVERS

Low-level communication with a mass storage device such as the standard Apple Disk][disk drive or the ProFile fixed disk is performed by a special assembly language subroutine that, for convenience, we'll call a "disk driver" even though the device may not actually be a disk drive. We say "low-level" because the disk driver is the common subroutine used by every ProDOS MLI command that must access the disk and because it is only the driver that directly manipulates the specific disk I/O locations used by the controller to communicate with the disk.

The most important tasks performed by most disk drivers are:

- Moving the disk's read/write head over any track on the disk by controlling the motion of a stepper motor,
- Identifying sectors or blocks within each track,
- Reading and writing sectors or blocks,
- Reading the write-protect (or other) status of the disk,
- Formatting the disk.

In the case of the Disk][, the driver performs these tasks by making use of several different I/O locations that can be used to control the disk stepper motor, store a byte on a diskette, read a byte from a diskette, and sense the write-protect status of the diskette.

In this chapter we will examine how ProDOS determines what disk devices are installed in the system and how it keeps track of the disk drivers associated with each such device. We will also review the general characteristics of a disk driver and give an example of how to write one.

How ProDOS Keeps Track of Disk Devices

When ProDOS is booted, one of the first things it does is to determine how many disk devices are installed in the system and where they are located. (We will see how disk devices are identified in the next section.) The number of active disk devices, less 1, is stored in DEVCNT ($BF31) in the ProDOS global page. (See Table 7-1 for a list of global page locations used by ProDOS to manage disk devices.)

Table 7-1. ProDOS global page areas used for disk drive identification.

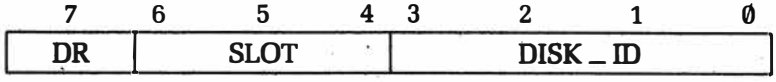
Address	*Symbolic Name*	*Description*
$BF10	DEVADR01	"No device connected" address
$BF12	DEVADR11	Slot 1/Drive 1 driver address
$BF14	DEVADR21	Slot 2/Drive 1 driver address
$BF16	DEVADR31	Slot 3/Drive 1 driver address
$BF18	DEVADR41	Slot 4/Drive 1 driver address
$BF1A	DEVADR51	Slot 5/Drive 1 driver address
$BF1C	DEVADR61	Slot 6/Drive 1 driver address
$BF1E	DEVADR71	Slot 7/Drive 1 driver address
$BF20	DEVADR02	"No device connected" address
$BF22	DEVADR12	Slot 1/Drive 2 driver address
$BF24	DEVADR22	Slot 2/Drive 2 driver address
$BF26	DEVADR32	Slot 3/Drive 2 driver address
$BF28	DEVADR42	Slot 4/Drive 2 driver address
$BF2A	DEVADR52	Slot 5/Drive 2 driver address
$BF2C	DEVADR62	Slot 6/Drive 2 driver address
$BF2E	DEVADR72	Slot 7/Drive 2 driver address
$BF30	DEVNUM	Device code for the last device accessed
$BF31	DEVCNT	Number of active devices - 1
$BF32	DEVLST	Table of active disk device codes (14 entries in table)

Note: The format of the entries in DEVLST and DEVNUM are as indicated in Figure 7-1 (except that the low-order four bits of DEVNUM are always 0).

The physical locations of the disk devices (that is, their slot and drive numbers) are stored in encoded form in a 14-byte table beginning at DEVLST ($BF32). As shown in Figure 7-1, the high-order four bits of an entry in this table hold the drive and slot number in packed form and the low-order four bits hold an identification code that is unique to the type of disk device installed (Disk][, ProFile, and so on).

Suppose you are using a two-drive Apple //e with the optional extended 80-column text card installed in the auxiliary slot and a disk controller card installed in slot 6. ProDOS will set up DEVCNT and DEVLST as follows:

Each byte in the 14-byte DEVLST table holds the slot, drive, and disk identification number in a special packed format:



where DR = 0 for a drive 1 device
= 1 for a drive 2 device
SLOT = 1-7 (slot number for the device)
DISK_ID = $0 for a Disk][or //c drive
= $4 for a ProFile
= $F for the /RAM device
= the high-order four bits stored at $CnFE if a disk controller adhering to the ProDOS extended protocol is being used.

Note: The /RAM device is logically equivalent to a slot 3/drive 2 disk drive. Its DEVLST entry is $BF.

Figure 7-1. The format of DEVLST ($BF32) table entries

```
DEVCNT ($BF31)   $02      <--- three devices
DEVLST ($BF32)   $E0      <--- slot 6/drive 2
                 $60      <--- slot 6/drive 1
                 $BF      <--- slot 3/drive 2 (/RAM)
                 $00   }
                 ,     }
                 ,     }  <--- 11 zero entries
                 ,     }
                 $00   }
```

ProDOS reserves a 32-byte area beginning at $BF10 for use as a disk driver vector table. This table holds the addresses of the disk driver to be used for each of the 14 possible slot/drive combinations, and 2 impossible ones (slot 0/drive 1 and slot 0/drive 2). The first part of the table, from $BF10 to $BF1F, holds the addresses for the eight drive 1 devices in ascending slot order (0–7); the second part holds similar information for the eight drive 2 devices.

Since a disk controller card may not be placed in slot 0 (a slot that doesn't even exist on the Apple //e or //c), ProDOS uses the two slot 0 entries in the disk driver vector table for a special purpose: to hold the address of the subroutine that will cause MLI error $28 to be generated if it is called. This is the code for the "no device connected" error. If the vector table entry for a given slot/drive combination is this address, then no disk device has been assigned to that slot/drive.

The four most common entries in the disk driver vector table are as follows:

$D000 disk driver for the Disk][(in bank-switched RAM)
$FF00 disk driver for the /RAM RAMdisk volume (in bank-switched RAM)

$DEA2 address of "no device connected" error subroutine (in bank-switched RAM)

$CnEA disk driver for the ProFile fixed disk (n = slot number of the ProFile controller card).

Note that the first three addresses given here are those used by ProDOS version 1.1.1 only (the fourth is fixed in ROM on the ProFile controller card). They will undoubtedly change as later versions of ProDOS are released.

How ProDOS Identifies Disk Devices

Disk devices are interfaced to the Apple II by means of a controller card that is inserted into one of the Apple's peripheral expansion slots. This card is designed to control the finer details of transferring data between the Apple and the disk (such as controlling a stepper motor to move the disk read/write head into position). A controller card also contains a ROM chip holding a program that occupies the address space from $Cn00 to $CnFF (where "n" is the slot number) and, sometimes, from $C800 to $CFFF as well. In most cases, such a program is only capable of transferring a short loader program from the disk into RAM and then executing it; this loader then reads in the rest of the disk operating system so that all disk operations can be performed. (This is where the term "booting" comes from: the operating system is literally picking itself up by its own bootstraps.) The program in Apple's standard Disk][controller card ROM is the best example of such a program. Other controllers may contain code that can perform much more sophisticated tasks, such as performing read or write operations on any block on the disk and doing status checks; such a controller is the one used with the ProFile fixed disk.

When ProDOS first starts up, it examines each slot to determine whether a controller card for a disk-like device is present. A controller card is identified by the following unique pattern of bytes in its ROM ("n" is the slot number):

```
$Cn01 : $20
$Cn03 : $00
$Cn05 : $03
```

The value of the byte stored at $Cn07 is also important. If the three identification bytes are present and $3C is stored at $Cn07, and if the controller is in a higher-numbered slot than any other controller with the same four values, then the standard Apple II system monitor will automatically boot the drive when the power is turned on. Unfortunately, $3C cannot be stored at $Cn07 in the ROM of a controller for a disk device other than a Disk][, because Apple's Pascal operating system erroneously believes that any such device is a Disk][drive. This means, for example, that it is not possible to automatically boot from a ProFile fixed disk. Note, however, that you can boot from a non-Disk][device if you have an Apple //e that uses the special "icon" ROM that was released by

Apple in early 1985; this is because the system monitor in the icon ROM identifies a bootable disk drive by the presence of the first three identification bytes only.

When such a pattern of bytes is detected, ProDOS looks at the byte stored at $CnFF to determine the exact type of controller that has been found. If $CnFF contains $00, then ProDOS identifies the card as a Disk][controller with standard 16-sector-per-track ROMs and places the appropriate device code in the DEVLST table and the address of the internal ProDOS Disk][device driver in the disk driver vector table. Note that two entries in each table are actually made since each Disk][controller can have two drives (or "volumes") attached to it (they are called drive 1 and drive 2). The disk driver itself will ultimately determine if there is actually a drive 2 device attached and will return a "device not connected" error code if an attempt is made to access it and it is not there.

If $CnFF contains $FF, then ProDOS identifies the card as a Disk][controller with 13-sector-per-track ROMs (this was the original diskette formatting scheme used with the Disk][). ProDOS does not support this type of controller card and so will ignore it.

If $CnFF contains any other value, then ProDOS assumes that the controller is being used with an intelligent disk device that has a device driver located in ROM at $CnXX, where XX is the value stored at $CnFF. If bits 0 and 1 of the byte stored at $CnFE are both 1 (we'll describe the meaning of these bits in the next section), this address will be stored in the device driver vector table, and an appropriate device code will be added to DEVLST (the low-order four bits of the DEVLST entry will be set equal to the high-order four bits of the byte at $CnFE). If one, or both, of bits 0 and 1 of $CnFE are 0, then the device will be ignored.

ProDOS identifies one special "disk" device in quite a different way. If it detects the presence of an extended 80-column text card on an Apple //e (that is, the one with 64K of auxiliary RAM on it), or if an Apple //c is being used, then ProDOS will install a special device, commonly called a RAMdisk, which will be treated as a slot 3/drive 2 disk device. The medium for this "disk" is the 64K auxiliary memory space on the //e or //c, and disk I/O operations simply involve the movement of data blocks between auxiliary and main memory. The volume name for this RAMdisk is always /RAM; we'll be describing the volume in greater detail later in this chapter.

Extended ProDOS Protocol for Disk Controller ROMS

Apple has also defined a special extended controller card ROM protocol that manufacturers of non-standard disk devices and disk controller cards must use in order to allow ProDOS to work properly with them. (The Apple Disk][controller does not follow this protocol and is handled as a special case by ProDOS.) This protocol defines the use of four bytes in the controller card ROM space as follows ("n" is the slot number of the card):

$CnFC and $CnFD. The total number of blocks on the volume is stored here (low-order byte first). This information is for use by a formatting program that must also initialize the volume directory and volume bit map on the disk. The controller for the ProFile has the number $2600 (9728) stored here. If the number stored is $0000, then you must send a status request to the disk device driver to determine the volume size; the number of blocks will be returned in the X register (low) and Y register (high). We'll see how to make status requests in the next section.

$CnFE. This is the device characteristics byte; each bit holds miscellaneous information about the device:

bit 7	1 = the disk medium is removable
bit 6	1 = the device is interruptible
bits 5,4	The number of drives (or volumes) on the device (0–3). An even value (0 or 2) indicates one drive; an odd value (1 or 3) indicates two drives.
bit 3	1 = the device driver supports format
bit 2	1 = the device driver supports write
bit 1	1 = the device driver supports read
bit 0	1 = the device driver supports status

The controller for the ProFile has the value $47 stored at $CnFE. This means that the disk medium is not removable (bit 7 = 0), that the ProFile is interruptible (bit 6 = 1), that there is only one volume supported (bits 5,4 = 00), and that the device driver for the ProFile, located in ROM on the controller card, cannot format the medium (bit 3 = 0) but that it can perform write (bit 2 = 1), read (bit 1 = 1), and status (bit 0 = 1) operations.

$CnFF. This byte contains the offset (from $Cn00) of the address of the ProDOS driver for this device. If the byte at $CnFE indicates that the device can be read from and its status can be read (that is, bits 0 and 1 of the byte stored at $CnFE are both 1), the driver address is stored in the "drive 1" portion of the device driver vector table in the ProDOS global page when ProDOS is first booted. If the byte at $CnFE indicates that two drives are attached to the controller, the address of the device driver is also stored in the "drive 2" portion of the table. The device driver for a controller that handles two drives is responsible for determining what drive has been requested before performing any I/O operations; it can easily do this because ProDOS passes the drive number to the disk driver whenever it calls it (see page 215). After the vector table is updated, bits 4–7 of the byte stored at $CnFE are stored in the low-order four bits of the DEVLST entry for the device.

The controller for the ProFile has the value $EA stored at $CnFF and its DEVLST entry is of the form "n4" where "n" is the controller slot number. This means that the address of the disk driver is $CnEA.

Special Cases: $CnFF will contain $00 for a 16-sector Disk][controller and $FF for a 13-sector disk controller. In these situations, ProDOS will attribute no special meaning to the values stored at $CnFC, $CnFD, and $CnFE.

You should note that the disk controller card for the Disk][does NOT follow the extended ProDOS protocol and is handled as a special case by ProDOS. When ProDOS identifies a standard 16-sector Disk][controller, it assumes that the disk holds a single volume of 280 blocks and will automatically use its own special internal disk driver to communicate with it. ProDOS will ignore the older 13-sector Disk][controllers.

Communicating with a Disk Driver

Just before ProDOS calls a disk driver subroutine, it sets up four parameters in the 6502 zero page that serve to inform the disk driver of the precise operation to be performed. These parameters define the type of disk operation (read, write, format, or check device status), the slot and drive number of the disk device, the address of the 512-byte block buffer to be used, and the block number on the disk device to be accessed.

The four parameters are stored in locations $42–$47 and have the following meanings:

COMMAND ($42). This location holds the command code for the disk operation to be performed. The following four codes are defined:

0 Check device status. If the device is ready for read and write operations, the carry flag will be cleared and the accumulator will be zeroed. In addition, if the device's controller ROM adheres to the extended ProDOS protocol (remember that the Disk][controller does not), the number of blocks on the disk will be returned in the X register (low) and Y register (high). If the device is not ready for read and write operations, the carry flag will be set and an MLI error code will be stored in the accumulator. The standard Disk][driver will return an error code on a status request only if the diskette is write-protected (error $2B).

1 Read one block from the disk.

2 Write one block to the disk.

3 Format the disk. When the disk is formatted, special address marks are set up to allow each sector to be identified by the disk driver. Generally speaking, the formatting process does *not* also set up the boot record, volume directory, and bit map blocks; this must be done by making write requests. (The /RAM driver is an exception.) The format request is actually not supported by the standard Disk][or ProFile device drivers because of space limitations; instead, a separate utility program (such as FILER on the ProDOS master diskette) must be used to format a diskette or a fixed disk and to lay out the boot record, volume directory, and bit map. The source code for the standard diskette formatting subroutines (called FORMATTER and BUILDDISK) can also be licensed from Apple for use in other formatting programs. The format request is supported by the /RAM driver.

SLOT _ DRIVE ($43). These locations hold the drive and slot number of the disk device to be accessed in the following format:

bit 7	0 (drive 1) or 1 (drive 2)
bits 4,5,6	slot number (1–7)
bits 0,1,2,3	always 0

For example, a slot 6/drive 2 device would be represented as 11100000 ($E0).

BUFFER _ PTR ($44/$45). These locations hold the address (low-order byte first) of the start of a 512-byte area of memory that holds the image of the block to be written to the disk (COMMAND = 2) or that will hold the block read from the disk (COMMAND = 1). BUFFER _ PTR should also be properly set up before making a format request (COMMAND = 3) because the formatting subroutines for some disk devices (like /RAM) may use the buffer area for temporary data storage.

BLOCK _ NUM ($46/$47). These locations hold the number (low-order byte first) of the block on the disk that is to be written (COMMAND = 2) or read (COMMAND = 1).

The disk driver will perform the I/O operation dictated by these parameters and then return control to the caller. If no error occurred, then the carry flag will be clear and the accumulator will be zero.

Errors can occur, of course, when ProDOS communicates with a disk device. Error conditions are flagged by the disk drivers in the standard MLI way: by setting the carry flag and placing an appropriate MLI error code in the accumulator. The error codes and conditions supported by the ProDOS disk driver for Apple's standard Disk][drives are shown in Table 7-2. Any other properly implemented disk driver will identify and report these error conditions in the same way.

Table 7-2. Disk driver error codes.

MLI Error Code	*Meaning*
$27	I/O error
$28	No disk device is connected
$2B	The medium is write protected

The ProDOS RAMdisk: The /RAM Volume

As we saw earlier, ProDOS will automatically install a special RAMdisk driver, if you are using an Apple //c or an Apple //e with an extended 80-column text card and will create a special volume called /RAM. (Apple II and Apple II

Plus users are out of luck.) Both these systems have 64K of *auxiliary* memory that is mapped to addresses in exactly the same way as the standard 64K of main RAM memory usually used for program and data storage. It is in this auxiliary memory that the RAMdisk driver stores the volume directory, volume bit map, and file blocks. A map of the usage of auxiliary memory by /RAM is shown in Figure 7-2.

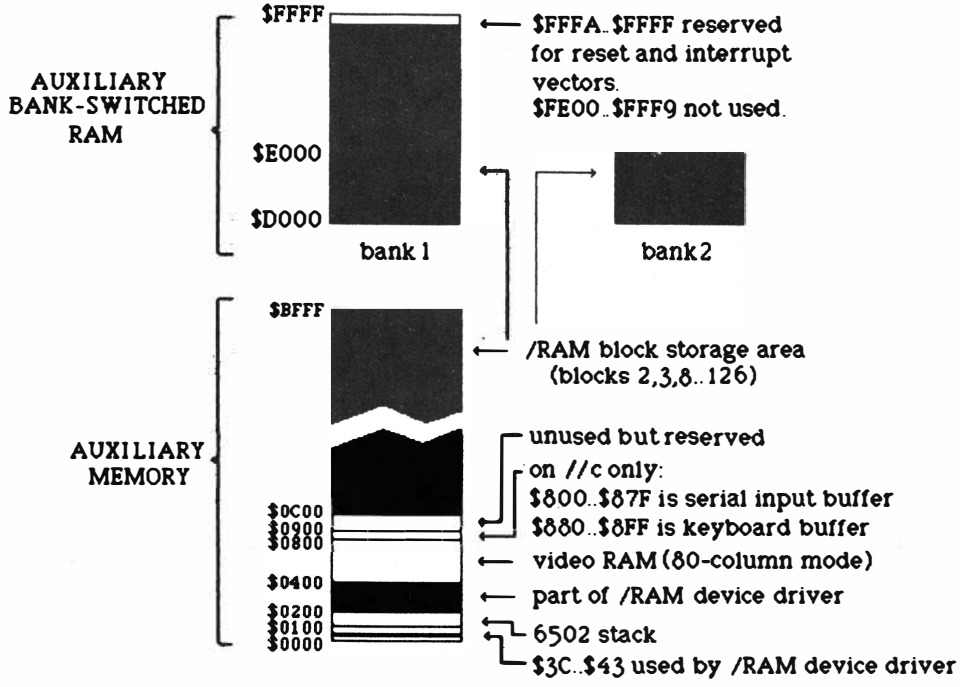


Figure 7-2. A map of auxiliary memory usage on the //e and //c

Since no slow-moving mechanical parts are used to perform "disk" operations (all I/O operations simply involve block moves from one part of memory to another), the RAMdisk responds much more quickly than a conventional disk drive. Its contents are temporary, however, so you must be careful to transfer any files from it to a permanent disk before turning off the Apple or rebooting the disk operating system or else you will lose all your data.

Characteristics of the /RAM Volume

When the /RAM volume is initialized by ProDOS, only one volume directory block (block 2) is allocated (recall that four blocks are used on standard disks). This means that there is room for only twelve entries, not the usual 51, in the volume directory. If files are stored in subdirectories, however, as many files as will fit on the volume can be stored.

A total of 119 blocks are available for file storage when the /RAM volume is first initialized (they are numbered from 8 to 126). Since a 64K space would normally be capable of holding 128 512-byte blocks, you might be wondering about the "missing" nine blocks. Two of these are relatively easy to track down: one is used for the volume directory (block 2) and another for the volume bit map (block 3). There is no room in auxiliary memory for the other seven blocks (0, 1, 4–7, and 127) because space must be reserved to support the /RAM disk driver itself ($0000–$03FF), the 80-column text screen ($0400–$07FF), the keyboard and serial input buffers on the Apple //c ($0800–$08FF), and the auxiliary memory 6502 interrupt vectors ($FFFA–$FFFF). Thus, these seven blocks are marked as in use in the /RAM volume bit map.

The areas of auxiliary memory that are not used by the /RAM volume or its driver are as follows:

- $00–$3B, $44–$FF
- $0900–$0BFF
- $FE00–$FFF9

Despite the apparent availability of these areas, they should be considered reserved for future use by later versions of ProDOS and must not be used by non-system software.

The first 8K of memory allocated for use by files stored in /RAM is mapped to locations $2000–$3FFF in auxiliary memory. As all serious users of the //c or //e will know, this same space is used whenever page1 of the special double-width high-resolution graphics display mode available on those models is active. If you are going to use this graphics mode while /RAM is active, then you must first prevent any meaningful program from being stored at these locations. The easiest way to do this is to ensure that the first file saved to /RAM is a dummy file that is exactly 8K bytes long. You can do this by entering the following command from Applesoft direct mode:

```
BSAVE /RAM/DUMMY,A$2000,E$3FFF
```

The second 8K area used to store files in /RAM is mapped to locations $4000–$5FFF, the same area that is used as the second page of double-width high-resolution graphics. You can protect this page by saving another dummy file that is 8K in size.

"Removing" and Re-Installing /RAM

You may find that you want, or need, to use the auxiliary memory area used by the /RAM volume for purposes other than as a file storage device. Other common uses for auxiliary memory are as a data buffer for a printer spooler, or as an input buffer for a communications program. Before you start overwriting the RAM volume with such data, however, you must first remove the /RAM volume from the system in an orderly manner. If you don't, the system could

crash when ProDOS tries to interpret what you've written to auxiliary memory as directory, bit map, or file information.

It's actually quite simple to remove the /RAM device from the system.

- Examine MACHID ($BF98) to see if you're running in a 128K system (bits 4 and 5 of MACHID will both be 1 if you are). /RAM can only exist in a 128K system.
- Check that /RAM has not already been removed by locating the $BF device code (slot 3/drive 2) among the active entries in the DEVLST table. (You should also check for any entry of the form $BX where X = $3, $7, or $B; by convention, these slot 3/drive 2 devices, although not equivalent to /RAM, will also use auxiliary memory). The actual $BX byte stored in DEVLST must be saved if you later want to re-install the /RAM device.
- Remove the $BX entry from the DEVLST table by moving higher-addressed active entries down one position (starting with the lowest-addressed one).
- Replace the slot 3/drive 2 entry in the device vector table (at $BF26/$BF27) with the address stored at the slot 0/drive 1 entry (at $BF10/$BF11). (This will be the address of the subroutine that will generate a "no device connected" error condition.) The original slot 3/drive 2 entry must be saved if you later want to re-install the /RAM device.
- Decrement DEVCNT ($BF31).

When all these steps have been performed, the /RAM device will disappear from ProDOS and auxiliary memory can be safely used for other purposes.

After /RAM has been removed you may want to install it again. This can be done by performing the following steps:

- As a precaution, verify that you have not already re-installed /RAM by checking for a slot 3/drive 2 device code in DEVLST.
- Restore the original slot 3/drive 2 device vector that you saved before /RAM was disconnected.
- Move each active entry in DEVLST to the next higher memory location (starting with the highest-addressed entry) and then store the /RAM device code (that you saved before /RAM was disconnected) at the first entry in the list (at $BF32).
- Increment DEVCNT ($BF31).
- Initialize the volume directory and volume bit map of the /RAM device by setting up the disk driver parameters for a format request ($42 = 3, $43 = $B0, $44/$45 = 512-byte buffer address) and then calling the disk driver. Since the /RAM device driver resides in bank 1 of bank-switched RAM, you will have to enable that bank by reading $C08B twice in succession before making the call. When the call ends, re-enable the Applesoft and motherboard ROMs by reading $C082. Here is a subroutine that will perform all these chores:

```
LDA #3          ;Format code
STA $42
```

```
        LDA #$B0     ;Unit number code
        STA $43
        LDA $73      ;Set buffer address
        STA $44      ; to HIMEM
        LDA $74
        STA $45
        LDA $C08B
        LDA $C08B    ;Read/write enable bank1
        JSR TORAM
        LDA $C082    ;Re-enable Applesoft ROMs
        RTS
TORAM   JMP ($BF26) ;Call the /RAM driver
```

After /RAM has been reinstalled like this it is once again available for use as a file storage device.

Writing a Disk Driver Program

The best way to learn about disk drivers and how ProDOS installs them is to actually write one. In this section we're going to do just that by creating a driver for an 8K version of /RAM that we'll call /RAM8. It will be suitable for use in an Applesoft programming environment and can be used by all Apple II users (recall that /RAM is only available on the Apple //e or //c). The RAMdisk driver itself will reside in page 3 and the "disk" storage space it uses will be located from $800 to $27FF. We will ensure that Applesoft programs will not conflict with the RAMdisk storage space by setting the Applesoft start-of-program pointer at $67/$68 to $2801 and then initializing the other Applesoft pointers and data areas by executing a NEW command.

Before we begin to write the disk driver, let's examine the steps to follow to remedy the Applesoft conflict, bind the driver into ProDOS, and then initialize the RAMdisk. This is really a five-step process.

The first step in the procedure is to adjust the Applesoft pointers so that when programs are entered or loaded, they will not overwrite the /RAM8 volume:

```
LDA #$01   ;Starting address (low)
STA $67    ;Program pointer (low)
LDA #$28   ;Starting address (high)
STA $68    ;Program pointer (high)
LDA #0
STA $2800  ;Program must start with $00
JSR $D64B  ;Applesoft NEW command
```

(Applesoft insists that the byte preceding the start of the program, $2800, be set to $00.)

Second, a slot and drive number for our new device must be selected. This is most easily done by examining the DEVLST table to see what combinations

are already in use and picking one that isn't. Let's assume that slot 3/drive 1 is available and use it.

We then must store $30 in the DEVLST table (this is the code for a slot 3/drive 1 device—see Figure 7-1) and increment DEVCNT. Here's the code to do it:

```
LDA #$30       ;DEVLST code for slot 3/drive 1
INC DEVCNT     ;Adding one device
LDY DEVCNT     ;DEVCNT now points to next
               ;available position in DEVLST
STA DEVLST,Y   ;Stuff device code in DEVLST
```

The next step is to install the address of the disk driver in the disk driver vector table (low-order byte first). The address of the slot 3/drive 1 entry in this table is $BF16. Here's how to store the address:

```
LDA #<RAMDISK ;Get low-order address byte
STA $BF16
LDA #>RAMDISK ;Get high-order address byte
STA $BF17
```

RAMDISK is the address of the disk driver that performs the I/O operations (we'll see what it looks like in a moment).

Finally, we must initialize the volume directory block and the volume bit map. Before we can do this, however, we have to know:

- the number of directory blocks
- the block number of the volume bit map block
- the number of blocks on the volume

Since it's unlikely that we'll be saving very many files in the 8K /RAM8 volume, we can save some space by using just one directory block (instead of the four used on standard disks). This block must be located at block 2 since ProDOS always expects the first block in the volume directory to be stored there.

The volume bit map block will be stored at block 3, leaving a total of 14 blocks (7K) that can be used to store files. To keep the file storage area contiguous, we'll assign these blocks to numbers 4 through 17 and mark blocks 0 and 1 as in use in the volume bit map. (We can't use block 0 for file storage anyway, since ProDOS uses a zero entry in a file index block as an end-of-file indicator.) This means that ProDOS will think that the volume size is 18 blocks (instead of 16) but that will not matter since the two extra blocks will not be available for file storage.

Since a "1" bit in the volume bit map indicates that a block is free, the volume bit map block must begin with a $0F byte (blocks 0-3 in use, blocks 4-7 free) followed by an $FF byte (blocks 8-15 free) and a $C0 byte (blocks 16 and 17 free). The remaining bytes in the block will never be used, but should be set to zero.

With this background information it is relatively simple to initialize /RAM8. The first step is to prepare an image of the volume directory block and then use the MLI WRITE_BLOCK command to write it to block 2. (You may want to review Chapter 2 for a description of the structure of such a block.) Every byte in the block will be zero except the following:

$04	storage type code and name length ($F4)
$05–$08	ASCII string for "RAM8" ($52 $41 $4D $38)
$22	access code ($C3)
$23	entry length ($27)
$24	entries per block ($0D)
$27/$28	block number for volume bit map ($0003)
$29/$2A	number of blocks on volume ($0012)

Since the directory links (at $00/$01 and $02/$03 in the block) are both zero, this will be the only block that ProDOS will examine for files in the volume directory.

The final step in the initialization procedure is to write an image of the volume bit map to block 3.

Now all we have to do is write the special /RAM8 disk driver. Before we begin, we must decide what memory locations will be used to hold each block in the volume. A convenient mapping scheme to use follows:

```
block  2 → $800–$9FF
block  3 → $A00–$BFF
block  4 → $C00–$DFF
,
,
,
block 17 → $2600–$27FF
```

(The driver will return an error code if a block number greater than 17 is specified.) If this scheme is used, the page number for a given block is equal to twice the block number plus 4. This number can be easily calculated by the driver subroutine. (To simplify the driver, we will also assign block 0 to $400–$5FF and block 1 to $600–$7FF even though these blocks are never used.)

As we saw earlier in this chapter, when the disk driver takes control, certain parameters are set up in the 6502 zero page by the calling program. One of these parameters is a command code that indicates what type of operation is to be performed: read, write, check status, or format. To save space, our driver won't include the formatting code, so we'll ignore all format requests. Status requests will also be ignored because such requests are meaningless in the context of a RAMdisk. Here's what the driver will look like:

```
CLD             ;(required by ProDOS)
LDA $6          ;Save zero page locations
STA ZPSAVE
```

```
        LDA $7
        STA ZPSAVE+1
        LDA $47         ;Check block number (high)
        BNE IOERROR     ;Error if not zero
        LDA $46         ;Check block number (low)
        CMP #18         ;Is it out of bounds?
        BCS IOERROR     ;It's >=18, so error
        ASL             ;Multiply block by 2
        CLC
        ADC #4          ; . . .  and add 4 to get
        STA $7          ;starting page of block
        LDA #0
        STA $6
        LDA $42         ;Get command code
        CMP #3          ;Format?
        BEQ EXIT        ;Yes, so exit normally
        CMP #0          ;Check status?
        BEQ EXIT        ;Yes, so exit normally
        CMP #1          ;Read?
        BEQ READ        ;Yes, so branch
        CMP #2          ;Write?
        BEQ WRITE       ;Yes, so branch
EXIT    CLC             ;CLC ==> no error
        LDA #0
EXIT1   PHP
        PHA
        LDA ZPTEMP      ;Restore zero page
                        locations
        STA $6
        LDA ZPTEMP+1
        STA $7
        PLA             ;Restore error code
        PLP             ;Restore carry status
        RTS
IOERROR SEC             ;SEC ==> error occurred
        LDA #$27        ;I/O ERROR code
        BNE EXIT1       ;(always taken)
READ    ,
        ["read" subroutine]
        ,
        JMP EXIT
WRITE   ,
        ["write" subroutine]
        ,
        JMP EXIT
ZPTEMP  DS 2            ;Temporary storage space
```

Note that the driver must begin with the CLD instruction that ProDOS checks to see if a valid driver is installed. The first part of the driver saves the contents of two zero page locations that we're going to overwrite and then checks to see whether the requested block number (which is stored at $46/$47)

is within the allowable range. If it isn't, then the driver ends with the carry flag set and the error code for "I/O error" ($27) in the accumulator.

The next part simply calculates the address of the requested block and stores it in two consecutive zero page locations ($6/$7) so that the block data can be accessed using the 6502 indirect indexed addressing mode.

The bodies of the READ and WRITE subroutines are both very simple to write. The READ code is responsible for moving the block of data from the address just calculated to the address specified by the caller (this address is stored at $44/$45). The WRITE code does just the reverse. Here are the two subroutines that will do the trick:

```
READ  LDY #0
R1    LDA ($6),Y   ;Get block data
      STA ($44),Y  ; and move it to caller's
                   buffer
      INY
      BNE R1       ;Branch until 256 bytes done
      INC $6       ;Move to second half
      INC $44
R2    LDA ($6),Y   ;Get block data
      STA ($44),Y  ; and move it to caller's
                   buffer
      INY
      BNE R2       ;Branch until 256 bytes done
      DEC $44
      JMP EXIT
WRITE LDY #0
W1    LDA ($44),Y  ;Get data from caller's
                   buffer
      STA ($6),Y   ; and move it to "disk" block
      INY
      BNE R1       ;Branch until 256 bytes done
      INC $44      ;Move to second half
      INC $6
W2    LDA ($44),Y  ;Get data from caller's
                   buffer
      STA ($6),Y   ; and move it to "disk" block
      INY
      BNE R2       ;Branch until 256 bytes done
      DEC $44
      JMP EXIT
```

As you can see, an I/O operation is performed simply by moving a 512-byte block of data from one area of memory to another.

The complete source listing for a slightly embellished form of this driver is shown in Table 7-3. One additional feature it includes is the marking of pages 3 and 8–27 as "in use" in the system bit map in the ProDOS global page to prevent the /RAM8 volume from being overwritten. Any attempt to load a file

into these areas (using BLOAD or BRUN) will result in a "NO BUFFERS AVAILABLE" error.

Use the BRUN command to install the driver program and then prove to yourself that it exists by entering the command

```
CATALOG/RAM8 (or CATALOG,S3,D1)
```

You should see a standard CATALOG listing followed by an indication that there are 14 blocks free and 4 blocks used, as expected. You can now save files to /RAM8 as you would to any other volume.

If you use the /RAM8 disk driver, be careful not to run any graphics programs that use the primary high-resolution graphics screen. The video RAM buffer used by this screen ($2000–$3FFF) overlaps the /RAM8 block storage area. Furthermore, the Applesoft program must not overwrite the device driver in page 3, or the storage space itself, by using POKE statements. If you want to avoid these memory conflicts, you can relocate the disk driver (and its corresponding disk space) to an area above HIMEM and the BASIC.SYSTEM general-purpose file buffer using the techniques described in Chapter 5.

You can remove the /RAM8 device from the system by using the same general technique described above for the removal of the /RAM volume. In addition, you will have to clear the appropriate bits in the system bit map, reset the Applesoft program pointer to $801, and execute an Applesoft NEW command to initialize other important Applesoft data pointers.

Table 7-3. The /RAM8 disk driver program.

```
                1   ********************************
                2   *                              *
                3   *  ProDOS RAMdisk disk driver  *
                4   *                              *
                5   * This driver controls a 8K    *
                6   * RAMdisk volume called /RAM8. *
                7   *                              *
                8   ********************************
                9
                10  RAMPTR   EQU  $6       ;Pointer to RAMdisk block
                11
                12  COMMAND  EQU  $42      ;Command code
                13  BUFFER   EQU  $44      ;Buffer address
                14  BLOCK    EQU  $46      ;Block number
                15
                16  TXTTAB   EQU  $67      ;Applesoft program pointer
                17
                18  INITBLK  EQU  $3000    ;Block buffer
                19
                20  MLI      EQU  $BF00    ;MLI interface
                21  DEVADR01 EQU  $BF10    ;Start of disk driver table
                22  DEVCNT   EQU  $BF31    ;Number of disk devices (minus
                                            1)
                23  DEVLST   EQU  $BF32    ;Table of slot/drive for disks
                24  BITMAP   EQU  $BF58    ;Start of system bit map
                25
                26           ORG  $2000
                27
                28  * Move device driver code into place:
                29
2000: A0 00     30           LDY  #0
```

```
2002: B9 F4 20 31  MOVECODE LDA  BEGIN,Y
2005: 99 00 03 32           STA  RAMDISK,Y
2008: C8       33           INY
2009: C0 7C    34           CPY  #END-RAMDISK
200B: D0 F5    35           BNE  MOVECODE
               36
               37  ************************************
               38  * Mark pages 3, 8..27 as "in use" *
               39  * in the system bit map. This     *
               40  * prevents /RAM8 or its driver    *
               41  * from being overwritten by BLOAD.*
               42  ************************************
200D: AD 58 BF 43           LDA  BITMAP
2010: 09 10    44           ORA  #$10      ;Block 3 bit = 1
2012: 8D 58 BF 45           STA  BITMAP
2015: A9 FF    46           LDA  #$FF
2017: 8D 59 BF 47           STA  BITMAP+1  ;Blocks 8..15
201A: 8D 5A BF 48           STA  BITMAP+2  ;Blocks 16..23
201D: AD 5B BF 49           LDA  BITMAP+3
2020: 09 F0    50           ORA  #$F0      ;Block 24..27 bits = 0
2022: 8D 5B BF 51           STA  BITMAP+3
               52
2025: AD C7 20 53           LDA  SLFAKE
2028: 0A       54           ASL
2029: 0A       55           ASL
202A: 0A       56           ASL
202B: 0A       57           ASL            ;Multiply slot by 16
202C: AC C8 20 58           LDY  DFAKE
202F: C0 01    59           CPY  #1        ;Drive 1?
2031: F0 02    60           BEQ  SETDS     ;Yes, so branch
2033: 09 80    61           ORA  #$80      ;Set bit 7 ("drive 2" bit)
2035: 8D DF 20 62  SETDS    STA  NEWDRSL
               63
```

(continued)

Table 7-3. The /RAM8 disk driver program (continued).

```
                        64  * Check for existing device:
2038: AC 31 BF          65          LDY  DEVCNT
203B: B9 32 BF          66  DUPCHECK LDA  DEVLST,Y ;Get existing slot/drive
203E: CD DF 20          67          CMP  NEWDRSL  ;Same as RAMdisk slot/drive?
2041: D0 01             68          BNE  DC1
                        69
2043: 00                70          BRK          ;Crash if duplicate found
                        71
2044: 88                72  DC1     DEY
2045: 10 F4             73          BPL  DUPCHECK  ;No, so on to next device
                        74
2047: EE 31 BF          75          INC  DEVCNT   ;Add "disk" drive
204A: AC 31 BF          76          LDY  DEVCNT
204D: AD DF 20          77          LDA  NEWDRSL
2050: 99 32 BF          78          STA  DEVLST,Y ;Save slot/drive code
                        79
2053: AD C7 20          80          LDA  SLFAKE   ;Get slot #
2056: 0A                81          ASL           ;x2 to step into table
2057: AC C8 20          82          LDY  DFAKE
205A: C0 01             83          CPY  #1       ;Drive 1?
205C: F0 03             84          BEQ  FIXTABLE ;Yes, so branch
                        85
205E: 18                86          CLC
205F: 69 10             87          ADC  #16      ;Offset to drive 2 table
                        88
2061: A8                89  FIXTABLE TAY
2062: A9 00             90          LDA  #<RAMDISK ;Save address of driver
2064: 99 10 BF          91          STA  DEVADR01,Y ; in vector table.
2067: A9 03             92          LDA  #>RAMDISK
2069: 99 11 BF          93          STA  DEVADR01+1,Y
                        94
```

```
                        95  ************************************
                        96  * Change Applesoft program pointer *
                        97  * and initialize program space.    *
                        98  ************************************
206C: A9 01             99          LDA  #1
206E: 85 67             100         STA  TXTTAB
2070: A9 28             101         LDA  #$28
2072: 85 68             102         STA  TXTTAB+1
2074: A9 00             103         LDA  #0
2076: 8D 00 28          104         STA  $2800     ;Must begin with $00 byte
2079: 20 4B D6          105         JSR  $D64B     ;Applesoft "NEW" command
                        106
                        107 ***************************
                        108 * Initialize the RAMdisk *
                        109 ***************************
207C: 20 E4 20          110         JSR  ZEROBLK
                        111
207F: A0 00             112         LDY  #0
2081: B9 C9 20          113 DONAME  LDA  VOLNAME,Y
2084: F0 06             114         BEQ  SETLEN
2086: 99 05 30          115         STA  INITBLK+5,Y ;Put volume name in block
                                                      buffer
2089: C8                116         INY
208A: D0 F5             117         BNE  DONAME
                        118
208C: 98                119 SETLEN  TYA
208D: 09 F0             120         ORA  #$F0      ;Set "directory" bits
208F: 8D 04 30          121         STA  INITBLK+4 ;Save file type + name length
                        122
                        123 * Store misc. volume parameters:
2092: A0 22             124         LDY  #$22
2094: B9 AC 20          125 DOPARMS LDA  INITPARM-$22,Y
2097: 99 00 30          126         STA  INITBLK,Y
```

(continued)

Table 7-3. The /RAM8 disk driver program (continued).

```
209A: C8          127           INY
209B: C0 2B       128           CPY   #$2B
209D: D0 F5       129           BNE   DOPARMS
                  130
209F: A9 02       131           LDA   #2
20A1: 8D E2 20    132           STA   BLKNUM    ;Writing to block 2
20A4: A9 00       133           LDA   #0
20A6: 8D E3 20    134           STA   BLKNUM+1
20A9: 20 D7 20    135           JSR   DOWRITE
                  136
                  137 ******************************
                  138 * Fix up the volume bit map *
                  139 ******************************
20AC: 20 E4 20    140           JSR   ZEROBLK
20AF: A9 0F       141           LDA   #$0F      ;0..3 in use / 4..7 free
20B1: 8D 00 30    142           STA   INITBLK
20B4: A9 FF       143           LDA   #$FF      ;8..15 free
20B6: 8D 01 30    144           STA   INITBLK+1
20B9: A9 C0       145           LDA   #$C0      ;16, 17 free
20BB: 8D 02 30    146           STA   INITBLK+2
                  147
20BE: EE E2 20    148           INC   BLKNUM    ;Change to block 3
20C1: 20 D7 20    149           JSR   DOWRITE
                  150
20C4: 4C D0 03    151           JMP   $3D0      ;Reconnect ProDOS hooks
                  152
20C7: 03          153 SLFAKE    DFB   3         ;RAMdisk slot #
20C8: 01          154 DFAKE     DFB   1         ;RAMdisk drive #
                  155
20C9: 52 41 4D    156 VOLNAME   ASC   'RAM8',00 ;Volume name
20CC: 38 00
                  157
```

```
20CE: C3        158 INITPARM DFB  $C3       ;Access code
20CF: 27        159          DFB  $27       ;Entry length
20D0: 0D        160          DFB  13        ;Entries/block
20D1: 00 00     161          DW   0         ;File count
20D3: 03 00     162          DW   3         ;Block for bit map
20D5: 12 00     163          DW   18        ;Total blocks
                164
                165 ********************************
                166 * Write a block to the device *
                167 ********************************
20D7: 20 00 BF  168 DOWRITE  JSR  MLI
20DA: 81        169          DFB  $81       ;WRITE_BLOCK command
20DB: DE 20     170          DA   CMDLIST
20DD: 60        171          RTS
                172
20DE: 03        173 CMDLIST  DFB  3
20DF: 00        174 NEWDRSL  DS   1         ;Drive and slot
20E0: 00 30     175          DA   INITBLK   ;I/O buffer
20E2: 00 00     176 BLKNUM   DW   0         ;Block # gets filled in here
                177
                178 ******************
                179 * Zero the block *
                180 ******************
20E4: A9 00     181 ZEROBLK  LDA  #0
20E6: A8        182          TAY
20E7: 99 00 30  183 ZB1      STA  INITBLK,Y
20EA: C8        184          INY
20EB: D0 FA     185          BNE  ZB1
20ED: 99 00 31  186 ZB2      STA  INITBLK+256,Y
20F0: C8        187          INY
20F1: D0 FA     188          BNE  ZB2
20F3: 60        189          RTS
                190
```

(continued)

Table 7-3. The /RAM8 disk driver program (continued).

```
                       191 BEGIN    EQU  *
                       192
                       193 *****************************
                       194 * This is the device driver *
                       195 * for the /RAM8 volume.     *
                       196 *****************************
                       197
                       198          ORG  $300
                       199
                       200 RAMDISK  EQU  *
                       201
0300: D8               202          CLD            ;(Required by ProDOS)
                       203
                       204 * Save zero page locations:
0301: A5 06            205          LDA  RAMPTR
0303: 8D 7A 03         206          STA  ZPTEMP
0306: A5 07            207          LDA  RAMPTR+1
0308: 8D 7B 03         208          STA  ZPTEMP+1
                       209
                       210 *********************************
                       211 * Check for block range error *
                       212 *********************************
030B: A5 47            213          LDA  BLOCK+1   ;Check block number (high)
030D: D0 34            214          BNE  IOERROR   ;Error if not zero
030F: A5 46            215          LDA  BLOCK     ;Check block number (low)
0311: C9 12            216          CMP  #18       ;Is it out of bounds?
0313: B0 2E            217          BCS  IOERROR   ;It's >=18, so error
                       218
                       219 ***********************************
                       220 * Convert block # to RAM address *
                       221 ***********************************
```

```
0315: 0A        222           ASL             ;Multiply block by 2
0316: 18        223           CLC
0317: 69 04     224           ADC   #4        ;... and add 4 to get
0319: 85 07     225           STA   RAMPTR+1  ;starting page of block
031B: A9 00     226           LDA   #0
031D: 85 06     227           STA   RAMPTR
                228
                229 ***********************
                230 * Check command code *
                231 ***********************
031F: A5 42     232           LDA   COMMAND   ;Get command code
0321: C9 03     233           CMP   #3        ;Format?
0323: F0 0C     234           BEQ   EXIT      ;Yes, so exit normally
0325: C9 00     235           CMP   #0        ;Check status?
0327: F0 08     236           BEQ   EXIT      ;Yes, so exit normally
0329: C9 01     237           CMP   #1        ;Read?
032B: F0 1B     238           BEQ   READ      ;Yes, so branch
032D: C9 02     239           CMP   #2        ;Write?
032F: F0 30     240           BEQ   WRITE     ;Yes, so write
                241
0331: 18        242 EXIT      CLC             ;CLC ==> no error
0332: A9 00     243           LDA   #0
0334: 08        244 EXIT1     PHP
0335: 48        245           PHA
0336: AD 7A 03  246           LDA   ZPTEMP
0339: 85 06     247           STA   RAMPTR
033B: AD 7B 03  248           LDA   ZPTEMP+1
033E: 85 07     249           STA   RAMPTR+1
0340: 68        250           PLA             ;Restore error code
0341: 28        251           PLP             ;Restore carry status
0342: 60        252           RTS
                253
```

(continued)

Table 7-3. The /RAM8 disk driver program (continued).

```
0343: 38        254 IOERROR  SEC             ;SEC ==> error occurred
0344: A9 27     255          LDA  #$27       ;I/O ERROR code
0346: D0 EC     256          BNE  EXIT1      ;(always taken)
                257
                258 ******************************
                259 * Perform READ command by    *
                260 * transferring data from the *
                261 * RAM to the data buffer.    *
                262 ******************************
0348: A0 00     263 READ     LDY  #0
034A: B1 06     264 FROMCARD LDA  (RAMPTR),Y
034C: 91 44     265          STA  (BUFFER),Y
034E: C8        266          INY
034F: D0 F9     267          BNE  FROMCARD
0351: E6 07     268          INC  RAMPTR+1
0353: E6 45     269          INC  BUFFER+1
0355: B1 06     270 FC1      LDA  (RAMPTR),Y
0357: 91 44     271          STA  (BUFFER),Y
0359: C8        272          INY
035A: D0 F9     273          BNE  FC1
035C: C6 45     274          DEC  BUFFER+1
035E: 4C 31 03  275          JMP  EXIT
                276
                277 ******************************
                278 * Perform WRITE command by   *
                279 * transferring data from the *
                280 * data buffer to the RAMcard.*
                281 ******************************
0361: A0 00     282 WRITE    LDY  #0
0363: B1 44     283 TOCARD   LDA  (BUFFER),Y
0365: 91 06     284          STA  (RAMPTR),Y
```

```
0367: C8        285             INY
0368: D0 F9     286             BNE   TOCARD
036A: E6 45     287             INC   BUFFER+1
036C: E6 07     288             INC   RAMPTR+1
036E: B1 44     289 TC1         LDA   (BUFFER),Y
0370: 91 06     290             STA   (RAMPTR),Y
0372: C8        291             INY
0373: D0 F9     292             BNE   TC1
0375: C6 45     293             DEC   BUFFER+1
0377: 4C 31 03  294             JMP   EXIT
                295
037A: 00 00     296 ZPTEMP      DS    2
                297
                298 END         EQU   *
```

Chapter 8

CLOCK DRIVERS

In Chapter 2 we saw that the directory entry for each ProDOS file contains four bytes that hold the time and date on which the file was created and four more bytes that hold the time and date on which it was last modified. This date-stamping feature of ProDOS is very useful, especially to those who routinely save several versions of the same program on different disks. Three months later you won't have to rack your brains trying to remember which one is the latest version; all you have to do is compare the modification dates (they are displayed when you enter the BASIC.SYSTEM CATALOG command).

But how does ProDOS calculate what the current time and date are? By using a plug-in peripheral called a clock card, that's how. A clock card contains special integrated circuits that allow it to keep track of the current time and date independently of the 6502 microprocessor. It is the Apple's digital watch, if you like. For convenience, you can install batteries in a clock card so that it will keep time even when the Apple is turned off.

A special assembly language program, called a *clock driver*, must be used to transfer the time and date from the card to the Apple in an understandable form. ProDOS has an internal clock driver that can be used to read one particular clock card only: the Thunderclock (manufactured by Thunderware, Inc. of Orinda, California). If the Thunderclock card (or a compatible card such as the Prometheus Versacard or the Applied Engineering Timemaster II) has been installed, ProDOS will automatically bind this clock driver into the system on bootup. If not, then a null driver will be used and you may be prompted to manually enter the correct time and date by whatever program is running.

In this chapter we'll discuss how ProDOS deals with time in general. In particular, we'll see how it detects the presence of the Thunderclock, how the Thunderclock clock driver is installed, and how you can install clock drivers

for clock cards that are not compatible with the Thunderclock. We'll also present some useful examples of how to make the most of the time and date feature of ProDOS.

How ProDOS Reads the Time and Date

Whenever ProDOS needs to know the time and date it always makes the same call: "JSR DATETIME". The code starting at DATETIME ($BF06) is either a one-byte RTS instruction (if a clock card is not installed) or a three-byte JMP instruction that passes control to a ProDOS clock driver (if a ProDOS-compatible clock card is installed). In either case, the two bytes at $BF07/$BF08 always hold the address of the ProDOS clock driver.

The clock driver reads the time and date from the clock card and stores the results in a special format at TIME ($BF92/$BF93) and DATE ($BF90/$BF91) in the ProDOS global page; the format used is described in Figure 8-1. If no clock driver is installed, TIME and DATE will not be modified because the RTS instruction stored at DATETIME ($BF06) immediately bounces control back to the caller. The only way of setting the time and date in this situation is to write directly to TIME and DATE.

(a) DATE ($BF90/$BF91)

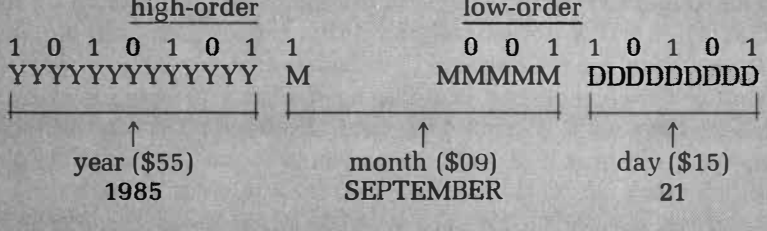
7	6	5	4	3	2	1	0	
Y6	Y5	Y4	Y3	Y2	Y1	Y0	M3	$BF91
M2	M1	M0	D4	D3	D2	D1	D0	$BF90

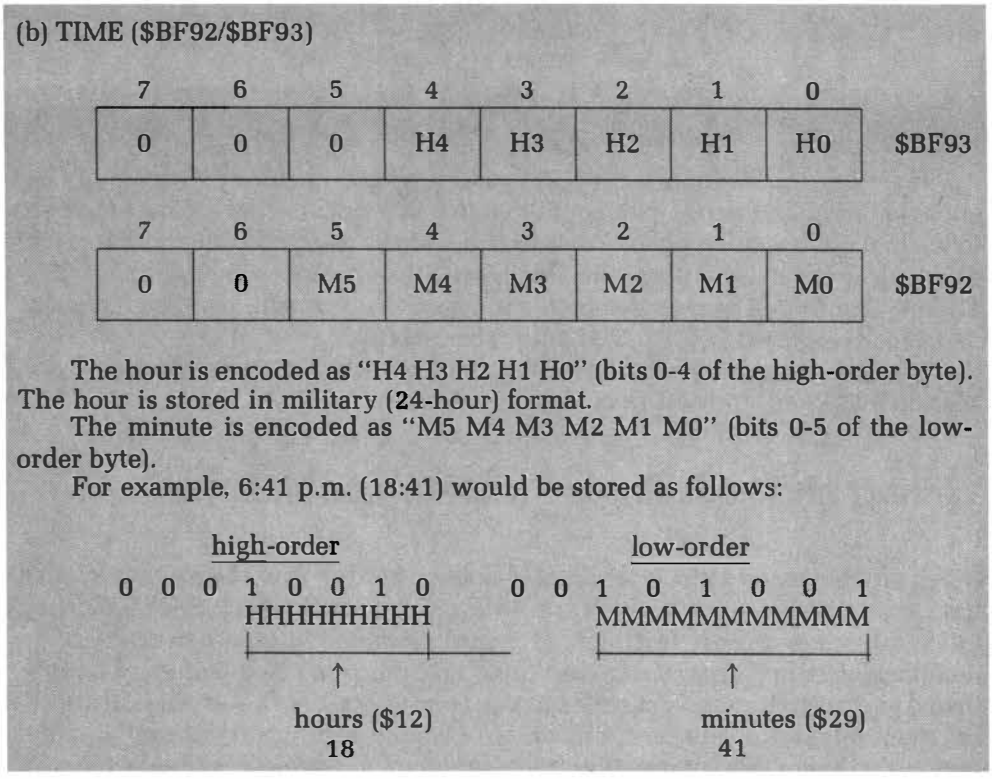
The year is encoded as "Y6 Y5 Y4 Y3 Y2 Y1 Y0" (bits 1-7 of the high-order byte). Only the last two digits of the year are stored (that is, "85" for "1985").

The month is encoded as "M3 M2 M1 M0" (bits 5-7 of the low-order byte and bit 0 of the high-order byte). January is month 1 and December is month 12.

The day of the month is encoded as "D4 D3 D2 D1 D0" (bits 0-4 of the low-order byte).

For example, September 21, 1985, would be stored as follows:





The hour is encoded as "H4 H3 H2 H1 H0" (bits 0-4 of the high-order byte). The hour is stored in military (24-hour) format.

The minute is encoded as "M5 M4 M3 M2 M1 M0" (bits 0-5 of the low-order byte).

For example, 6:41 p.m. (18:41) would be stored as follows:

Figure 8-1. The formats of the DATE and TIME bytes.

The approved method of determining the date and time in your own program is to use the MLI GET_TIME ($82) command. Recall from Chapter 4 that this is done by executing a subroutine like this:

```
JSR $BF00 ;Make a call to the MLI
DFB $82   ;GET_TIME
DA $0000  ;Dummy parameter table
RTS
```

After this subroutine finishes, the TIME and DATE locations will contain the current time and date in the format described above.

How ProDOS Identifies a Clock Card

When ProDOS is first booted it examines each peripheral expansion slot in the system for the presence of a Thunderclock clock card, or equivalent. ProDOS is able to identify such a card by the presence of the following unique pattern of bytes in the card's $Cn00–$CnFF ROM space ("n" is the slot number):

$Cn00	$08
$Cn02	$28
$Cn04	$58
$Cn06	$70

If a clock card is found, ProDOS installs its built-in clock driver simply by changing the RTS opcode ($60) at $BF06 to a JMP opcode ($4C). Since the two bytes that follow this opcode contain the address of the Thunderclock clock driver (low-order byte first), the driver will take control whenever the "JSR $BF06" instruction is executed by the program that wants to read the time and date (such as the MLI GET _ TIME ($82) command).

ProDOS also sets the clock bit (bit 0) of the machine identification byte, MACHID ($BF98), to 1 if a clock card is found.

Writing and Installing a ProDOS CLock Driver

If you are using a clock card that is not compatible with the Thunderclock, then you will have to write and install your own ProDOS clock driver. Writing a clock driver is no easy feat, since it requires detailed information concerning the clock circuitry interface to the Apple and the procedure that must be followed to extract time and date information from the card. In a best case situation, the manufacturer of the card will have a detailed technical reference manual that will contain this information. More commonly, however, you will have to beg, borrow, or steal this information before you can get started! Happily, most reputable manufacturers of clock cards have already written their own ProDOS clock drivers and include them on diskette with their hardware.

The general characteristics of a clock driver are:

- It must start with a CLD instruction.
- It must read the time and date from the clock card and store the results in the proper format in the global page TIME ($BF92/$BF93) and DATE ($BF90/$BF91) locations.

Once the driver has been written, you have to move it to an area of memory that will not be used by other programs. The best area available is that used by the very Thunderclock clock driver that you are replacing; the starting address of this area is always stored at $BF07/$BF08 (low-order byte first).

If you do choose to use the Thunderclock area (and we do recommend this selection), you must keep several important considerations in mind:

- Never assume that the Thunderclock driver will reside at the same position in every version of ProDOS. To ensure that your driver will run properly at any address that might be stored at $BF07/$BF08, you should avoid using JMP and JSR instructions or storing data within the main body of the driver. If you don't the code will not be relocatable and you will have to patch it to resolve all internal absolute address references after it has been moved to its new position.

- Make sure that your clock driver is no longer than 125 bytes. ProDOS reserves this amount of space for its Thunderclock driver and Apple has guaranteed this amount of driver space.
- Before moving your clock driver into position, write-enable bank 1 of bank-switched RAM by reading from location $C08B twice in succession (the Thunderclock driver resides in bank-switched RAM). After the move, re-enable the Applesoft and system monitor ROM area by reading from location $C082.

The next step in the installation procedure is to set up a JMP instruction at $BF06 that points to your clock driver. This can be done by storing $4C (the JMP opcode) at $BF06, and the address of the driver at $BF07/$BF08 (low-order byte first). If you have loaded the driver at the address of the Thunderclock driver, you can skip the latter step since the correct driver address will already be in place.

Finally, you should set bit 0 of MACHID ($BF98) to 1 to indicate that a clock card has now been installed in the system. You can do this be executing the following short piece of code:

```
LDA MACHID ;Get ID byte
ORA #$01   ;Store a 1 in bit 0
STA MACHID ;Update ID byte
```

The easiest way to install a clock driver is to make the installation program part of the STARTUP program automatically run when ProDOS executes the BASIC.SYTEM Applesoft interpreter.

Time/Date Utility Programs

Loading the Time and Date into an Applesoft String Variable

Some dialects of BASIC have a special variable called TIME$ that always contains the current time in the standard HH:MM:SS form. This variable is very useful when it is necessary to display the current time, to automatically time-stamp printed reports, to calculate elapsed times, to perform benchmarking studies, and so on.

Unfortunately, Applesoft does not have such a variable. Until now, that is! The READ.TIME subroutine in Table 8-1 can be used to return the time and date in the form "DD-MM-19YY HH:MM" in any string variable you want. To use the subroutine after it has been loaded, execute the following statement from within an Applesoft program:

```
CALL 768,TM$
```

where TM$ represents the name of the variable that is to hold the time string.

Table 8-1. READ.TIME, a program to load the time and date into an Applesoft string variable.

```
                      1   ********************************
                      2   *          READ.TIME           *
                      3   *                              *
                      4   * This program reads the time  *
                      5   * and date into an Applesoft   *
                      6   * string variable. The syntax  *
                      7   * is:                          *
                      8   *                              *
                      9   *       CALL 768,TM$           *
                      10  *       TM$=MID$(TM$,1)        *
                      11  *                              *
                      12  * The result is stored in the  *
                      13  * form: DD-MM-19YY  HH:MM      *
                      14  ********************************
                      15
                      16  VARPNT   EQU  $83        ;Pointer to string data
                      17
                      18  IN       EQU  $200       ;Input buffer
                      19
                      20  MLI      EQU  $BF00      ;Entry point to MLI
                      21  DATE     EQU  $BF90      ;Year + Month + Day
                      22  TIME     EQU  $BF92      ;Minutes + Hours
                      23
                      24  CHKCOM   EQU  $DEBE      ;Skip comma
                      25  PTRGET   EQU  $DFE3      ;Locate a variable
                      26
                      27           ORG  $300
                      28
0300: 20 00 BF        29           JSR  MLI        ;Call the MLI
0303: 82              30           DFB  $82        ; and select GET_TIME command.
0304: 00 00           31           DA   $0000      ;(no parameter table)
```

```
                        32
                        33  * "Unpack" the time:
                        34
0306: AD 92 BF  35              LDA   TIME        ;Get minutes
0309: 8D AB 03  36              STA   MINUTES     ; and save them
030C: AD 93 BF  37              LDA   TIME+1      ;Get hours
030F: 8D AC 03  38              STA   HOURS       ; and save them
0312: AD 90 BF  39              LDA   DATE        ;Get "day" bits (0...4),
0315: 29 1F     40              AND   #$1F         ; strip "month" bits,
0317: 8D AD 03  41              STA   DAY         ; and store correct number.
031A: AD 91 BF  42              LDA   DATE+1      ;Get "year" bits (1...7)
031D: 8D AF 03  43              STA   YEAR        ; and month bit (0).
0320: AD 90 BF  44              LDA   DATE        ;Get month bits (5...7)
0323: 4E AF 03  45              LSR   YEAR        ;Put "year" bits into 0...6
0326: 6A        46              ROR               ;Get all "month" bits in one
                                                  byte
0327: 4A        47              LSR               ; and move them into
0328: 4A        48              LSR               ; the lower five bits.
0329: 4A        49              LSR
032A: 4A        50              LSR
032B: 8D AE 03  51              STA   MONTH       ;Save month bits (0...4)
                        52
                        53  * Assemble the Applesoft time/date string:
                        54
032E: A2 00     55              LDX   #0
0330: 8E AA 03  56              STX   TIMEPOS     ;Clear ptr to time string
0333: 8A        57  FORMTIME    TXA
0334: 48        58              PHA
0335: BD B0 03  59              LDA   FORMAT,X    ;Get formatting byte
0338: 08        60              PHP
0339: AE AA 03  61              LDX   TIMEPOS
033C: 28        62              PLP
033D: 30 1E     63              BMI   NOTNUM      ;Branch if not number
```

(continued)

Table 8-1. READ,TIME, a program to load the time and date into an Applesoft string variable (continued).

```
033F: A8          64              TAY             ;Get time code in Y
                  65
0340: B9 AB 03    66              LDA  TIMEDATA,Y ;Get binary time/date data
0343: 20 85 03    67              JSR  CONVERT    ;Convert to BCD
0346: 48          68              PHA             ;Save number
0347: 4A          69              LSR             ;Move "tens" digit to
0348: 4A          70              LSR             ; lower four bits by
0349: 4A          71              LSR             ; shifting right four
034A: 4A          72              LSR             ; times.
034B: 09 30       73              ORA  #$30       ;Convert to ASCII digit
034D: 9D 00 02    74              STA  IN,X
0350: E8          75              INX
0351: 68          76              PLA             ;Get original number back
0352: 29 0F       77              AND  #$0F       ;Isolate units digit
0354: 09 30       78              ORA  #$30       ;Convert to ASCII digit
0356: 9D 00 02    79              STA  IN,X
0359: E8          80              INX
035A: 4C 63 03    81              JMP  TONEXT
                  82
035D: 29 7F       83  NOTNUM      AND  #$7F       ;Strip high bit for Applesoft
035F: 9D 00 02    84              STA  IN,X       ;Insert punctuation
0362: E8          85              INX
0363: 8E AA 03    86  TONEXT      STX  TIMEPOS
0366: 68          87              PLA
0367: AA          88              TAX
0368: E8          89              INX             ;Go to next position
0369: E0 0C       90              CPX  #12        ;At end of template?
036B: D0 C6       91              BNE  FORMTIME   ;No, so keep going
                  92
                  93  * Point Applesoft variable to time/date string:
                  94
```

```
036D: 20 BE DE  95            JSR  CHKCOM      ;Skip over comma
0370: 20 E3 DF  96            JSR  PTRGET
0373: AD AA 03  97            LDA  TIMEPOS     ;Get length of string
0376: A0 00     98            LDY  #0
0378: 91 83     99            STA  (VARPNT),Y  ;... and save it
037A: C8        100           INY
037B: A9 00     101           LDA  #<IN
037D: 91 83     102           STA  (VARPNT),Y  ;Save address (low)
037F: C8        103           INY
0380: A9 02     104           LDA  #>IN
0382: 91 83     105           STA  (VARPNT),Y  ;Save address (high)
0384: 60        106           RTS
                107
                108 ****************************
                109 * Binary to BCD Conversion *
                110 * Number must be 0...99    *
                111 ****************************
0385: 8D A9 03  112 CONVERT   STA  TEMP        ;Put # into work area
0388: 8E A8 03  113           STX  XSAVE
038B: A9 00     114           LDA  #0          ;Start with a 0 result
038D: F8        115           SED              ;Use decimal arithmetic
038E: A2 06     116           LDX  #6          ;Examine bits 0...6
0390: 4E A9 03  117 NEXTBIT   LSR  TEMP        ;Move low bit into carry
0393: 90 04     118           BCC  NOWEIGHT    ;Branch if it was zero
0395: 18        119           CLC              ; else add it
0396: 7D A1 03  120           ADC  BINDEC,X    ; to result.
0399: CA        121 NOWEIGHT  DEX              ;Count down to -1
039A: 10 F4     122           BPL  NEXTBIT     ;Branch if more to go
039C: D8        123           CLD              ;Return to binary arithmetic
039D: AE A8 03  124           LDX  XSAVE
03A0: 60        125           RTS
                126
```

(continued)

Table 8-1. READ,TIME, a program to load the time and date into an Applesoft string variable (continued).

```
03A1: 64 32 16  127 BINDEC   DFB  $64,$32,$16 ;These are the weights of the
03A4: 08 04 02  128          DFB  $08,$04,$02 ;the low seven bits in
03A7: 01        129          DFB  $01         ; a byte (in BCD).
                130
03A8: 00        131 XSAVE    DS   1           ;Temporary X location
03A9: 00        132 TEMP     DS   1           ;Temporary work area
03AA: 00        133 TIMEPOS  DS   1
                134
                135 TIMEDATA EQU  *
                136
03AB: 00        137 MINUTES  DS   1            ;Minutes (0...59)
03AC: 00        138 HOURS    DS   1            ;Hours (0...23)
03AD: 00        139 DAY      DS   1            ;Day of month (1...31)
03AE: 00        140 MONTH    DS   1            ;Month of year (1...12)
03AF: 00        141 YEAR     DS   1            ;Year (0...99)
                142
                143 * Formatting template for "DD-MM-19YY  HH:MM"
                144 * (digits refer to entries in TIMEDATA table)
                145
                146 FORMAT   EQU  *
                147
03B0: 02        148          DFB  2
03B1: AD 03 AD  149          DFB  "-",3,"-"
03B4: B1 B9 04  150          DFB  "1","9",4
03B7: A0 A0 01  151          DFB  $A0,$A0,1
03BA: BA 00     152          DFB  ":",0
```

When READ.TIME is called, it first uses the MLI GET_TIME ($82) command to read the current time and date into the ProDOS global page locations. It then unpacks the year, month, and day data from the DATE locations and stores each of them in their own temporary locations. The hours and minutes are already unpacked, but they are also transferred to temporary locations.

After this has been done, READ.TIME begins to assemble the ASCII time string in the Applesoft input buffer starting at $200. It does this by scanning a special template string that contains either ASCII characters or single-digit time codes. The ASCII characters are transferred directly to the time string. When a time code is encountered, however, the corresponding time parameter is loaded, converted to a binary-coded decimal (BCD) number, and then stored as two consecutive ASCII digits in the time string.

The string is then moved from the input buffer to the main Applesoft string space in the high end of memory to ensure that the string will not be overwritten when the next Applesoft INPUT statement is executed. This is done by using two Applesoft ROM subroutines called GETSPACE ($E4B2) and MOVSTR ($E5E2). When GETSPACE is called with the string length in the accumulator, it makes room for the string by lowering FRETOP ($6F/$70), the pointer to the bottom of string space, by the appropriate number of bytes. MOVSTR moves a string of length A that is pointed to by Y (high) and X(low) to this free space.

Once the time string is in position, READ.TIME locates the TM$ variable in the Applesoft variable table by executing the following two instructions:

```
JSR CHKCOM
JSR PTRGET
```

(CHKCOM ($DEBE) and PTRGET ($DFE3) are two more Applesoft ROM subroutines.) The first instruction advances the Applesoft program pointer by one byte, effectively skipping over the comma separating the CALL address from the variable. The second instruction stores the address of the three byte descriptor that defines the string variable in VARPNT ($83) and VARPNT+1 ($84). The first byte in the descriptor is the length of the string; the next two bytes contain the pointer to the contents of the string.

The final step is to store the new string length and pointer in the descriptor. The length (TIMEPOS) is stored in the first descriptor byte, and the pointer to the string, found at FRETOP ($6F) and FRETOP+1 ($70), is stored in the other two bytes.

Setting the Time and Date on a Clockless Apple

Even if you do not have a clock card installed in your Apple (and you won't if you are using an Apple //c), you can still date- and time-stamp a file by manually storing the current date and time in the ProDOS global page locations just before saving the file to a disk. This is somewhat inconvenient, but it's better than nothing. If you can survive with just the correct date, things become much easier, because then you need only set the date once when you first turn on the Apple (assuming, perhaps naively, that you don't work past midnight).

The TIMEDATE program in Table 8-2 will allow you to enter a time and date in English. After you do so, the program will convert the information into the encoded format used by ProDOS, and then store it in the ProDOS global page locations.

Table 8-2. TIMEDATE, a program to manually set the time and date.

```
0     REM "TIMEDATE"
100   NOTRACE : TEXT : PRINT CHR$ (21): SPEED=
      255: NORMAL : HOME
110   DIM MT$(12)
140   FOR I = 1 TO 12: READ MT$(I): NEXT
150   DATA JANUARY,FEBRUARY,MARCH,APRIL,MAY,
      JUNE,JULY,AUGUST,SEPTEMBER,OCTOBER,
      NOVEMBER,DECEMBER
160   PRINT "PRODOS TIME/DATE SETTER"
165   PRINT "COPYRIGHT 1984 GARY B. LITTLE"
170   T1 = 49042: REM $BF92 (MINUTES)
180   T2 = 49043: REM $BF93 (HOURS)
190   T3 = 49040: REM $BF90 (MMMDDDDD)
200   T4 = 49041: REM $BF91 (YYYYYYYM)
400   VTAB 6: CALL - 958: INPUT "ENTER YEAR (0-
      99): ";A$:YR = VAL (A$): IF YR < 0 OR YR >
      99 OR A$ = "" THEN 400
500   VTAB 7: CALL - 958: INPUT "ENTER MONTH
      (JAN...DEC): ";A$
510   FOR I = 1 TO 12: IF A$ = MT$(I) OR A$ =
      LEFT$ (MT$(I),3) THEN MT = I:I = 12: NEXT :
      GOTO 600
520   NEXT : GOTO 500
600   VTAB 8: CALL - 958: INPUT "ENTER DAY OF
      MONTH (1-31): ";A$:DY = VAL (A$): IF DY < 1
      OR DY > 31 THEN 600
720   VTAB 9: CALL - 958: INPUT "ENTER HOUR (0-
      23): ";A$:HR = VAL (A$): IF HR < 0 OR HR >
      23 OR A$ = "" THEN 720
800   VTAB 10: CALL - 958: INPUT "ENTER MINUTES
      (0-59): ";A$:MN = VAL (A$): IF MN < 0 OR MN
      > 59 OR A$ = "" THEN 800
1000  PRINT : PRINT "PRESS ANY KEY TO SET": PRINT
      "THIS TIME AND DATE: ";: GET A$: PRINT A$
1010  POKE T1,MN
1020  POKE T2,HR
1030  POKE T4,2 * YR + INT (MT / 8)
1040  POKE T3,32 * (MT - 8 * INT (MT / 8)) + DY
```

Appendix I

USING 6502 ASSEMBLERS

All the assembly language programs presented in this book were entered and assembled using the Merlin Pro assembler written by Glen E. Bredon and published by Roger Wagner Publishing, Inc. (10761 Woodside Avenue, Suite E, Santee, California 92071). If you want to modify and reassemble the programs presented in this book and you are not using Merlin Pro, then you will likely have to make several changes to the program source codes to account for any differences in syntax and command structure. Differences usually arise in the area of "pseudo-instructions;" these are assembler-specific commands that appear in the 6502 instruction field of a line of source code, but that represent commands to the assembler, rather than 6502 instructions. They can be used to place data bytes at specific locations within the program (DFB, DS, DA, and ASC), to define symbolic labels (EQU), to indicate the starting address of the program (ORG), and for several other purposes.

Here are some examples of how to use Merlin Pro's most important pseudo-opcodes:

DFB $03	Stores the byte $03 in the object code.
DS 16	Reserves a data space of 16 bytes (to no particular value).
DA $FDED	Stores the address $FDED in the object code as $ED $FD (that is, low-order byte first).
ASC 'ABCD'	Stores the ASCII codes for ABCD in the object code (with bit 7 cleared to 0).

	ASC "ABCD"	Stores the ASCII codes for ABCD in the object code (with bit 7 set to 1).
COUT	EQU $FDED	Equates the symbolic label "COUT" with the address $FDED.
	ORG $0300	Instructs the assembler to start assembling the code beginning at $300.

The operand formats for most 6502 assemblers are generally quite similar. (The operand is the part that identifies what data or address an instruction is to act on.) One major difference is the way in which the high-order or low-order byte of a two-byte address is identified as an immediate quantity. With Merlin Pro, you use an operand of the form "#<ADDRESS" to identify the low- order byte and "#>ADDRESS" to identify the high-order byte, where "ADDRESS" is the address being examined.

Most other assemblers use quite a different method, the most common of which is to use #ADDRESS to identify the low-order byte and /ADDRESS to identify the high-order byte. Perversely, one assembler, the Apple 6502 Editor/ Assembler, uses the same general method, but it reverses the meaning: "#>" is used to specify the low-order byte and "#<" is used to specify the high-order byte! Be careful.

Appendix II

EXISTING VERSIONS OF ProDOS AND BASIC.SYSTEM

Version Number	*Version Date*	*ProDOS Global Page Identification Code*
ProDOS 1.0	01-NOV-83	KVERSION ($BFFF) = 0
ProDOS 1.0.1	01-JAN-84	KVERSION ($BFFF) = 0
ProDOS 1.1	17-AUG-84	KVERSION ($BFFF) = 1
ProDOS 1.1.1	18-SEP-84	KVERSION ($BFFF) = 1
BASIC 1.0	15-NOV-83	IVERSION ($BFFD) = 0
BASIC 1.1	18-JUN-84	IVERSION ($BFFD) = 1

Appendix III

CORRESPONDENCE OF ProDOS BLOCKS TO DOS 3.3 SECTORS

The ProDOS MLI READ _ BLOCK ($80) and WRITE _ BLOCK ($81) commands discussed in Chapter 4 can also be used to access directly any sector on any track of a DOS 3.3-formatted diskette. This is convenient because it means that it is easy to write ProDOS utilities capable of reading DOS 3.3 files or creating and writing DOS 3.3 files. To handle DOS 3.3 files properly you will, of course, need detailed information on how DOS 3.3 organizes and manages diskette files. Refer to Chapter 5 of *Inside the Apple //e* for this information.

To use READ _ BLOCK and WRITE _ BLOCK with DOS 3.3 diskettes, you first have to translate the DOS 3.3 sector number into a block number that these commands understand. Sectors on a DOS 3.3 diskette are identified by a track number (0–34) and a sector number within the track (0–15). The corresponding ProDOS block number can be calculated from the track/sector values by first multiplying the track number by 8 to determine the base block number, and then adding to the base the relative block number for the sector. The relative block numbers for each DOS 3.3 sector are as follows:

Relative Block Number	*DOS 3.3 Sector Number*
0	00 and 14
1	13 and 12
2	11 and 10
3	09 and 08
4	07 and 06
5	05 and 04
6	03 and 02
7	01 and 15

For example, track 17/sector 15 on a DOS 3.3 diskette corresponds to block number 143 (8 × 17 + 7).

Note that since a ProDOS block is twice the size of a DOS 3.3 sector, each ProDOS block corresponds to two DOS 3.3 sectors, as shown in the table. The first half of the block corresponds to the first sector in the pair and the last half to the second sector. This doubling causes a complication when writing to a DOS 3.3 diskette: a sector other than the one you want to write to will also be written to. To avoid destroying the data in the other sector, it is first necessary to read the desired block into a buffer, transfer to it the contents of the sector to be written, and then write the block back to diskette. In this way, the contents of the other sector will not be disturbed.

Appendix IV

BIBLIOGRAPHY

Since ProDOS is a relatively new operating system there is not yet a great deal of published material that explains how it works and how to use it. The bibliography that follows is a reasonably complete list of all articles and books on ProDOS that were published up to February 1985.

Magazine Articles:

R.D. Bacon, "ProMENU," *Nibble*, September 1984, pp. 73-84.

C. Bongers, "ProDOS—Pros and Cons," *Call -A.P.P.L.E.*, May 1984, pp. 51-54.

M.P. Caffrey, "Introducing ProDOS," *Apple Orchard*, January 1984, pp. 12-16.

G.W. Charpentier and D. Sparks, "Taking the "Pro" out of ProDOS," *Call -A.P.P.L.E.*, November 1983, pp. 9-14.

J. Eugenides, "Booting ProDOS With a Modified Monitor ROM," *Apple Assembly Line*, June 1984, p. 18.

C. Fretwell, "Adding a TYPE command to ProDOS," *Call -A.P.P.L.E.*, May 1984, pp. 42-49.

C. Fretwell, "ProDOS and Friends," *Call -A.P.P.L.E.*, January 1984, pp. 65-69.

C. Fretwell, "ProDOS PREFIX Command," *Call -A.P.P.L.E.*, February 1984, pp. 57-59.

C. Fretwell, "Setting I/O Hooks in ProDOS," *Call -A.P.P.L.E.*, April 1984, p. 39.

K. Kashmarek, "Converting Random Files to ProDOS," *Call -A.P.P.L.E.*, April 1984, pp. 42-43. A program is presented that can be used to transfer random-access textfiles between DOS 3.3 and ProDOS.

K. Kashmarek, "ProDOS Profiles," *Call -A.P.P.L.E.*, September 1984, pp. 48-49.

K. Kashmarek, "ProDOS Prose From a ProDOS Pro," *Call -A.P.P.L.E.*, January 1985, pp. 26-32.

A. Messing, "/RAM, A Free RAM Disk for ProDOS Users," *Nibble*, February 1985, p. 71.
R. Moore, "ProDOS," *Byte*, February 1984, pp. 252-262. This is a review.
S. Mossberg, "A Technical Overview of ProDOS," *Nibble*, April 1984, pp. 58-63.
S. Mossberg, "An Introduction to ProDOS," *Nibble*, March 1984, pp. 153-159.
S. Mossberg, "Disassembling the ProDOS BASIC Interpreter," *Nibble*, September 1984, pp. 101-105.
S. Mossberg, "Launching the ProDOS BASIC Interpreter," *Nibble*, November 1984, pp. 120-135.
S. Mossberg, "ProDOS BASIC Global Page," *Nibble*, January 1985, pp. 96-112.
S. Mossberg, "RAM Reservation Center," *Nibble*, December 1984, pp. 80-93. This describes the ProDOS system bit map.
R.D. Norling, "Running Applesoft Programs with ProDOS," *Call -A.P.P.L.E.*, July 1984, pp. 43-50.
B. Sander-Cederlof, "Commented Listing of ProDOS: $F800-$F90B, $F996-$FEBD," *Apple Assembly Line*, November 1983, pp. 2-14.
B. Sander-Cederlof, "Listing of ProDOS: $F90C-$F995, $FD00-$FE95, $FEBE-$FFFF," *Apple Assembly Line*, December 1983, pp. 2-11.
B. Sander-Cederlof, "ProDOS and Clock Drivers," *Apple Assembly Line*, November 1983, pp. 25-28.
B. Sander-Cederlof, "Timemaster II from Applied Engineering," *Apple Assembly Line*, December 1983, pp. 19-20. A discussion of ProDOS-compatible clock cards.
B. Sander-Cederlof, "Will ProDOS Really Fly?," *Apple Assembly Line*, June 1984, p. 18.
E.A. Seiden, "ProDOS Reset Trap," *Nibble*, October 1984, p. 107.
B. Stout, "Will ProDOS Work on a Franklin?," *Apple Assembly Line*, March 1984, p. 20. See update in *Apple Assembly Line*, June 1984, p. 1.
R. Sutcliffe, "Managing with Ampermanager," *Call -A.P.P.L.E.*, July 1984, pp. 30-34. This article describes how to create hybrid DOS 3.3/ProDOS diskettes.
T. Weishaar, "A TYPE Command for ProDOS," *Softalk*, June 1984, pp. 157-165.
T. Weishaar, "Breaking the Floppy Barrier: An Introduction to Apple's ProDOS," *Softalk*, January 1984, pp. 112-118.
T. Weishaar, "Drawing a Bead on Text Files," *Softalk*, August 1984, pp. 43-47. How to use ProDOS textfiles.
T. Weishaar, "IIc Blues and Text File Reviews," *Softalk*, July 1984, pp. 41-47. How to use ProDOS textfiles.
T. Zilbert, "Return to Apple, Embrace Change," *Softalk*, August 1984, pp. 61-64. A beginner's introduction to ProDOS.

Books and Assemblers:

BASIC Programming with ProDOS, Apple Computer, Inc., 1983.
G. Bredon, *Merlin Pro*, Roger Wagner Publishing, Inc., 1984. This is a ProDOS-based 6502 assembler.

J.L. Campbell, *Inside Apple's ProDOS*, Reston Publishing Company, Inc., 1984.
ProDOS Assembler Tools, Apple Computer, Inc., 1983. This is a ProDOS-based 6502 assembler.
ProDOS Technical Reference Manual, Apple Computer, Inc., 1983.
ProDOS User's Manual, Apple Computer, Inc., 1983.
D.D. Worth and P.M. Lechner, *Beneath Apple ProDOS*, Quality Software, 1984.

The following books should also be consulted for detailed information on the internal software structure of the Apple II:

All About Applesoft, Apple Pugetsound Program Library Exchange, 1981.
Apple II Reference Manual, Apple Computer, Inc., 1979. This is the reference manual for the Apple II and Apple II Plus and it contains the source code for the system monitor on these two models.
Applesoft BASIC Programmer's Reference Manual, Volumes 1 and 2, Apple Computer, Inc., 1982, 1983.
G.B. Little, *Inside the Apple //c*, Brady Communications Co., Inc., 1985.
G.B. Little, *Inside the Apple //e*, Brady Communications Co., Inc., 1985.
Reference Manual for //e Only, Apple Computer, Inc., 1982.
Reference Manual Addendum: Monitor ROM Listings, Apple Computer Inc., 1982. This book contains the source code for the system monitor on the Apple //e.
The Apple //c Reference Manual, Volumes 1 and 2, Apple Computer, Inc., 1984. Volume 2 contains the source code for the system monitor on the Apple //c.

Appendix V

THE PROGRAM DISKETTE

A diskette containing the source code for each of the programs described in this book, as well as three "bonus" programs, can be ordered directly from the author. See the last page of this book for ordering information.

The files on the diskette are of three types:

1. TXT (text) files having names of the form "xxxxxxxxxx.S." These files contain assembly language source code in the format expected by the Merlin Pro assembler.
2. BAS (BASIC) files. These files contain Applesoft programs that can be executed using the BASIC.SYSTEM RUN or "dash" command.
3. BIN (binary) files. These files contain machine language programs that can be executed using the BASIC.SYSTEM BRUN or "dash" command. A BIN file is created from its corresponding source code file by assembling the source with Merlin Pro.

Note that the program diskette is not bootable because it does not contain a copy of the PRODOS and BASIC.SYSTEM files. These files can be transferred to it from a ProDOS master diskette using the ProDOS FILER program.

The names of the programs on the diskette are the same as those used in this book.

Here are descriptions of the three bonus programs:

DISK.MAP

The DISK.MAP program draws a map on the Apple's low-resolution graphics screen showing the usage of each block on a ProDOS- formatted diskette. To run the program, enter the command

```
BRUN DISK.MAP
```

from Applesoft direct mode. After you do this, you will be asked to insert the diskette that you want to examine and to press any key to begin. DISK.MAP maps each block on the diskette to a unique position in a 8x35 rectangular grid map. The horizontal axis represents the track number from 0 (left) to 34 (right); the vertical axis represents the relative block number within the track from 0 (bottom) to 7 (top).

Differently colored low-resolution graphics blocks are used to indicate the usage of any particular disk block. If blue is used, the disk block is in use and readable; if white is used, the disk block is in use but not readable (that is, it has been damaged). If the graphics block is grey, then the disk block is not being used.

DISK.MAP also displays the amount of free space on the diskette and the name of the volume directory.

PROTIME

When PROTIME is executed (using the BRUN or "dash" command), a command called "TIME" is added to the BASIC.SYSTEM command set. When the TIME command is entered in Applesoft direct mode, the current time and date are displayed in the following format:

```
DD-MMM-19YY  HH:MM
```

where DD represents the day of the month, MMM represents the first three characters in the name of the month, 19YY represents the year, HH represents the hour, and MM represents the minute.

For example, if the current date is March 6, 1986 and the time is 3:22 p.m., the result will be displayed as:

```
06-MAR-1986  15:22
```

Notice that the time is displayed in 24-hour (military) format.

The TIME command behaves differently when it is invoked from within an Applesoft program. In this case, the time is not displayed on the screen; rather, the string variable associated with the very next INPUT statement in the program is set equal to the time string. For example, when the following program line is executed,

```
100 PRINT CHR$(4);"TIME": INPUT TM$
```

the time string will be assigned to the TM$ variable. The Applesoft string parsing commands can then be used to isolate the elements of the string that are needed.

PROTYPE

The PROTYPE program adds the TYPE command to the BASIC.SYSTEM command set. This command displays the contents of a file on the video screen or sends it to a printer. It is most useful for examining the contents of a file that contains readable text.

To install the TYPE command, enter the command

```
BRUN PROTYPE
```

from Applesoft direct mode. If all goes well, you will see the message

```
TYPE COMMAND IS NOW INSTALLED.
```

and the command will be available for use.

The syntax for the TYPE command is:

```
TYPE pn [,L#][,F#][,E#][,R#][,T#][,@#][,S#][,D#]
```

where brackets are used to enclose optional parameters and "#" represents a decimal or hexadecimal number (a hexadecimal number must be preceded by "$"). Here is the meaning of each parameter:

pn = the pathname for the file
,L# = number of lines to be printed per page
,F# = form size (in lines)
,E# = left margin position
,R# = rest code (non-zero = => page pause)
,T# = title code (non-zero = => number the pages)
,@# = slot number for output
,S# = slot number for the file
,D# = drive number for the file

The default parameters are: 54 (,L#), 66 (,F#), 0 (,E#), 0 (,R#), 0 (,T#), current output (,@#).

As you can see, the TYPE command supports several parameters used to format the output and specify its destination. For example, the command

```
TYPE MY.TEXT,@1,F84,L72,R1,T1,E5
```

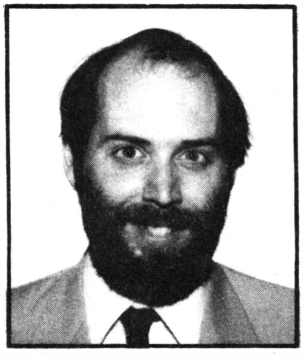
would be used to send a file called MY.TEXT to a printer in slot 1 (,@1). The size of the paper is 84 lines (,F84), 72 lines will be printed before a form feed is generated (,L72), and there will be a pause at the top of each new page to allow you to insert single sheet paper (,R1). In addition, a page number will appear on each page (,T1) and there will be a left margin of five spaces (,E5).

Note that you can temporarily halt all output generated by the TYPE command by entering [control-S] from the keyboard. To resume, press [control-S] (or any other key except [control-C]) once again. You can press [control-C] at any time to cancel the command.

Index

About the Author

Gary B. Little is an expert Apple II (and II Plus, //e, //c, . . .) programmer who resides in Vancouver, British Columbia. He is a founding member of the Apple British Columbia Computer Society and of SAGE (Serious Apple Group, Eh!) and is also an active member of several business organizations that promote and assist software developers. Gary has written numerous articles for several computer publications and is the author of two recent microcomputer books published by Brady Communications, "Inside the Apple //e" and "Inside the Apple //c."



— NOTES —

— NOTES —

— NOTES —

RELATED RESOURCES SHELF

Inside the Apple //e
Gary B. Little

This book offers a comprehensive look at the advanced features and capabilities of the Apple IIe. You will explore the fundamentals behind the 6502 microprocessor, the operating system and more.

☐1985/399pp/paper/D5513-9/$19.95
☐Book-Diskette/1985/D5556-8/$49.95
☐Diskette/1985/D5548-5/$30.00

Inside the Apple //c
Gary B. Little

This book presents an insider's view of the 65C02 microprocessor that controls the //c, ProDos, the //c monitor commands, input/output, and more. A must-have for every //c owner.

☐1985/363pp/paper/D5645-9/$19.95
☐Diskette/1985/D5653-3/$25.00

BASIC for the Apple //e and //c
Bill Searle and Donna Jones

A tutorial of BASIC on the Apple //e and //c that stresses structured BASIC and interaction with the ProDOS operating system. Introduces the use of subroutines early.

☐1985/288pp/paper/D3375-5/$16.95g

TO ORDER, simply clip or photocopy this entire page, check off your selection, and complete the coupon below. Enclose a check or money order for the stated amount. (Please add $2.00 postage and handling per book plus local sales tax.)

Mail to: **Brady Communications Co., Inc., Dept. TS, Bowie, MD 20715**

Name _____

Address _____

City/State/Zip _____

Charge my credit card instead: ☐ MasterCard ☐ Visa

Account# _____ Expiration Date _____

Signature _____

Dept. Y Y0510-BB(5)

Prices subject to change without notice.

TO ORDER, simply clip or photocopy this entire page and complete the coupon below. Enclose a check or money order for $25.00. Or charge it to your MasterCard or VISA.

Mail to: **RMS Inc.,** No. 210-131 Water St., Vancouver, B.C. Canada V6B 4M3
(604) 681-3371

Name _____

Address _____

City/State/Zip _____

Charge my credit card instead: ☐ MasterCard ☐ Visa

Account# _____ Expiration Date _____

Signature _____

Do this and you will receive the diskette for
Apple ProDOS: Advanced Features for Programmers.

Now you can discover the magic of ProDOS. The diskette offers virtually instant access to most of the programs in the book—an invaluable tool for every Apple owner.

MLI COMMANDS AND PARAMETER TABLES

ALLOC _ INTERRUPT ($40)

+0	(byte) parameter count (2)
+1	(byte, result) interrupt number
+2,3	(address) address of interrupt handler

CLOSE ($CC)

+0	(byte) parameter count (1)
+1	(byte) reference number

CREATE ($C0)

+0	(byte) parameter count (7)
+1,2	(address) pathname pointer
+3	(byte) access code
+4	(byte) file type code
+5,6	(word) auxiliary type code
+7	(byte) storage type code
+8,9	(word) date of creation
+10,11	(word) time of creation

DEALLOC _ INTERRUPT ($41)

+0	(byte) parameter count (1)
+1	(byte) interrupt number

DESTROY ($C1)

+0	(byte) parameter count (1)
+1,2	(address) pointer to pathname

FLUSH ($CD)

+0	(byte) parameter count (1)
+1	(byte) reference number

GET _ BUF ($D3)

+0	(byte) parameter count (2)
+1	(byte) reference number
+2,3	(address, result) location of file buffer

GET _ EOF ($D1)

+0	(byte) parameter count (2)
+1	(byte) reference number
+2,3,4	(3 bytes, result) EOF position

GET _ FILE _ INFO ($C4)

+0	(byte) parameter count (10)
+1,2	(address) pathname pointer
+3	(byte, result) access code
+4	(byte, result) file type code
+5,6	(word, result) auxiliary type code (*)
+7	(byte, result) storage type code
+8,9	(word, result) blocks used
+10,11	(word, result) date of modification
+12,13	(word, result) time of modification
+14,15	(word, result) date of creation
+16,17	(word, result) time of creation

GET _ MARK ($CF)

+0	(byte) parameter count (2)
+1	(byte) reference number
+2,3,4	(3 bytes, result) MARK position

GET _ PREFIX ($C7)

+0	(byte) parameter count (1)
+1,2	(address, result) pointer to prefix

GET _ TIME ($82)

--- no parameter table ---

NEWLINE ($C9)

+0	(byte) parameter count (3)
+1	(byte) reference number
+2	(byte) "AND" enable mask
+3	(byte) newline character

ON _ LINE ($C5)

+0	(byte) parameter count (2)
+1	(byte) unit number
+2,3	(address) location of data buffer

OPEN ($C8)

+0	(byte) parameter count (3)
+1,2	(address) pointer to pathname
+3,4	(address) location of file buffer
+5	(byte, result) reference number

QUIT ($65)

+0	(byte) parameter count (4)
+1	(byte) [reserved]
+2,3	(address) [reserved]
+4	(byte) [reserved]
+5,6	(address) [reserved]

READ ($CA)

+0	(byte) parameter count (4)
+1	(byte) reference number
+2,3	(address) location of data buffer
+4,5	(word) requested number
+6,7	(word, result) actual number read

READ _ BLOCK ($80)

+0	(byte) parameter count (3)
+1	(byte) unit number
+2,3	(address) location of data buffer
+4,5	(word) block number

RENAME ($C2)

+0	(byte) parameter count (2)
+1,2	(address) pointer to current pathname
+3,4	(address) pointer to new pathname

SET _ BUF ($D2)

+0	(byte) parameter count (2)
+1	(byte) reference number
+2,3	(word) address of new file buffer

SET _ EOF ($D0)

+0	(byte) parameter count (2)
+1	(byte) reference number
+2,3,4	(3 bytes) new EOF position

SET _ FILE _ INFO ($C3)

+0	(byte) parameter count (7)
+1,2	(address) pathname pointer
+3	(byte) access code
+4	(byte) file type code
+5,6	(word) auxiliary type code
+7	(byte) [not used]
+8,9	(word) [not used]
+10,11	(word) date of modification
+12,13	(word) time of modification

SET _ MARK ($CE)

+0	(byte) parameter count (2)
+1	(byte) reference number
+2,3,4	(3 bytes) new MARK position

SET _ PREFIX ($C6)

+0	(byte) parameter count (1)
+1,2	(address) pointer to pathname

WRITE ($CB)

+0	(byte) parameter count (4)
+1	(byte) reference number
+2,3	(address) location of data buffer
+4,5	(word) requested number
+5,6	(word, result) actual number written

WRITE _ BLOCK ($81)

+0	(byte) parameter count (3)
+1	(byte) unit number
+2,3	(address) location of data buffer
+4,5	(word) block number

(*) if the pathname is that of a volume directory, the total number of blocks on the volume is returned instead of the auxiliary file type code.

MLI ERROR CODES

$00 No error occurred
$01 Invalid MLI command
$04 Invalid parameter count
$25 Interrupt table is full
$27 Disk I/O error
$28 No disk device connected
$2B Disk is write protected
$2E Disk volume was removed
$40 Invalid pathname syntax
$42 No more file buffers allowed
$43 File is not open
$44 Directory does not exist
$45 Volume directory does not exist
$46 File does not exist
$47 Duplicate file name
$48 Disk is full
$49 Directory is full
$4A Incompatible version
$4B Invalid storage type
$4C End of data
$4D Range error
$4E File is locked
$50 File is busy
$51 Damaged directory
$52 Not a ProDOS disk
$53 Parameter out of range
$55 VCB table is full
$56 Buffer area is in use
$57 Volumes have same name
$58 Volume bit map damaged

MLI COMMAND INTERPRETER

Calling sequence:

```
JSR   $BF00      ;call MLI interpreter
DFB   CMDNUM     ;MLI command number
DA    PARMTBL    ;address of parm table
BCS   ERROR      ;Carry set if error occurs
```

If an error occurs, the carry flag will be set, the zero flag will be clear, and an error code will be returned in the accumulator.

UNIT NUMBER CODE

DSSS0000
where D = 1 (drive 2) or 0 (drive 1)
SSS = 1-7 (slot)

STORAGE TYPE CODE

$00	inactive (deleted) file
$01	seedling file
$02	sapling file
$03	tree file
$0D	directory file
$0E	subdirectory header
$0F	volume directory header

FILE TYPE CODES

$04	TXT
$06	BIN
$0F	DIR
$19	ADB
$1A	AWP
$1B	ASP
$EF	PAS
$F0	CMD
$F1-$F8	[user-definable]
$FA	INT
$FB	IVR
$FC	BAS
$FD	VAR
$FE	REL
$FF	SYS

(all others are reserved)

ACCESS CODE

bit 7 : 1 = delete-enabled
bit 6 : 1 = rename-enabled
bit 5 : 1 = backup required
bit 4 : always 0
bit 3 : always 0
bit 2 : always 0
bit 1 : 1 = write-enabled
bit 0 : 1 = read-enabled

DATE FORMAT

MMMDDDDD (low byte)
YYYYYYYM (high byte)
M = month (1..12)
D = day (1..31)
Y = year (0..99)

TIME FORMAT

00MMMMMM (low byte)
000HHHHH (high byte)
M = minutes (0..59)
H = hours (0..23)

AUXILIARY TYPE CODES

TXT	record length
BAS, BIN, SYS, VAR	address from which file was stored

PATHNAMES

The pathname pointed to by an entry in an MLI parameter table begins with a length byte and is followed by the ASCII codes for the name.

USEFUL LOCATIONS IN THE ProDOS GLOBAL PAGE

$BF00	MLI	Main entry point to MLI interpreter
$BF03	QUIT	Call to quit code (used by QUIT)
$BF06	DATETIME	Call to clock driver (used by GETTIME)
$BF09	SYSERR	System error handler
$BF0C	SYSDEATH	Critical error handler
$BF0F	SERR	System error number
$BF10/$BF11	DEVADR01	"No device connected" vector
$BF12/$BF13	DEVADR11	Slot 1, Drive 1 disk driver
$BF14/$BF15	DEVADR21	Slot 2, Drive 1 disk driver
$BF16/$BF17	DEVADR31	Slot 3, Drive 1 disk driver
$BF18/$BF19	DEVADR41	Slot 4, Drive 1 disk driver
$BF1A/$BF1B	DEVADR51	Slot 5, Drive 1 disk driver
$BF1C/$BF1D	DEVADR61	Slot 6, Drive 1 disk driver
$BF1E/$BF1F	DEVADR71	Slot 7, Drive 1 disk driver
$BF20/$BF21	DEVADR02	"No device connected" vector
$BF22/$BF23	DEVADR12	Slot 1, Drive 2 disk driver
$BF24/$BF25	DEVADR22	Slot 2, Drive 2 disk driver
$BF26/$BF27	DEVADR32	/RAM driver (//e, //c only)
$BF28/$BF29	DEVADR42	Slot 4, Drive 2 disk driver
$BF2A/$BF2B	DEVADR52	Slot 5, Drive 2 disk driver
$BF2C/$BF2D	DEVADR62	Slot 6, Drive 2 disk driver
$BF2E/$BF2F	DEVADR72	Slot 7, Drive 2 disk driver
$BF30	DEVNUM	Unit number for last disk device used
$BF31	DEVCNT	Number of disk devices (-1)
$BF32-$BF3F	DEVLST	List of disk device unit numbers
$BF58-$BF6F	BITMAP	System bit map
$BF90/$BF91	DATE	Date bytes
$BF92/$BF93	TIME	Time bytes
$BF94	LEVEL	Current file level
$BF98	MACHID	Machine identification byte:
	00xx0xxx	= Apple II
	01xx0xxx	= Apple II Plus
	10xx0xxx	= Apple //e
	11xx0xxx	= Apple III emulation mode
	10xx1xxx	= Apple //c
	xx01xxxx	= 48K
	xx10xxxx	= 64K
	xx11xxxx	= 128K (//e, //c only)
	xxxxxx0x	= no 80-column card
	xxxxxx1x	= 80-column card
	xxxxxxx0	= no clock card
	xxxxxxx1	= clock card
$BF9A	PFXPTR	Prefix active byte (0 = no prefix)
$BF9B	MLIACTV	MLI active flag (>127 = active)
$BF9C/$BF9D	CMDADR	Return address for last MLI command
$BFFC	IBAKVER	Minimum ProDOS version for interpreter
$BFFD	IVERSION	Current interpreter version
$BFFE	KBAKVER	Earliest compatible ProDOS version
$BFFF	KVERSION	Current ProDOS version

BASIC.SYSTEM EXTERNAL COMMAND HANDLING

$BE06	EXTRNCMD	JMP to external command parser
$BE50/$BE51	EXTRNADDR	Address of external command handler
$BE52	CMDLEN	Length of external command (minus 1)
$BE53	XCNUM	External command number (always 0)
$BE54/$BE55	PBITS	Permitted parameters
$BE56/$BE57	FBITS	Found parameters

Meaning of bits in PBITS/FBITS:
bit 7 : fetch prefix if pathname not specified
bit 6 : slot number required/found
bit 5 : command NOT valid in direct mode
bit 4 : pathname is optional (no names + parms)
bit 3 : create file if it doesn't exist
bit 2 : file type optional ("T" parameter)/found
bit 1 : second pathname required (for RENAME)/found
bit 0 : filename allowed/found

Meaning of bits in PBITS + 1/FBITS + 1:
bit 7 : "A" parameter allowed/found
bit 6 : "B" parameter allowed/found
bit 5 : "E" parameter allowed/found
bit 4 : "L" parameter allowed/found
bit 3 : "@" parameter allowed/found
bit 2 : "S"/"D" parameters allowed/found
bit 1 : "F" parameter allowed/found
bit 0 : "R" parameter allowed/found

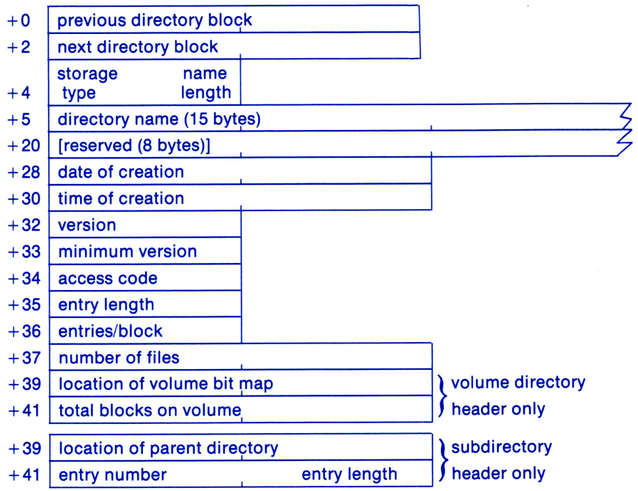
$BE58/$BE59	APARM	Value of "A" parameter
$BE5A-$BE5C	BPARM	Value of "B" parameter
$BE5D/$BE5E	EPARM	Value of "E" parameter
$BE5F/$BE60	LPARM	Value of "L" parameter
$BE61	SPARM	Value of "S" parameter
$BE62	DPARM	Value of "D" parameter
$BE63/$BE64	FPARM	Value of "F" parameter
$BE65/$BE66	RPARM	Value of "R" parameter
$BE67	VPARM	Value of "V" parameter
$BE68/$BE69	@PARM	Value of "@" parameter
$BE6A	TPARM	Value of "T" parameter
$BE6B	SLPARM	Value of "snum" parameter
$BE6C/$BE6D	PNAME1	Pointer to first pathname
$BE6E/$BE6F	PNAME2	Pointer to second pathname

BASIC.SYSTEM MLI CALLER

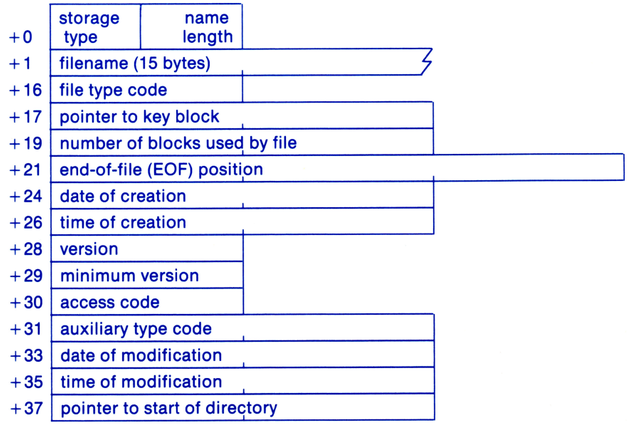
$BE70	GOSYSTEM	Enter with MLI code in A; error code returned in A
$BEA0	CREATE parameter list	
$BEAC	GET _ PREFIX parameter list	
$BEAC	SET _ PREFIX parameter list	
$BEAC	DESTROY parameter list	
$BEAF	RENAME parameter list	
$BEB4	GET _ FILE _ INFO parameter list (*)	
$BEB4	SET _ FILE _ INFO parameter list (*)	
$BEC6	ON _ LINE parameter list	
$BEC6	SET _ MARK parameter list	
$BEC6	GET _ MARK parameter list	
$BEC6	SET _ EOF parameter list	
$BEC6	GET _ EOF parameter list	
$BEC6	SET _ BUF parameter list	
$BEC6	GET _ BUF parameter list	
$BECB	OPEN parameter list	
$BED1	NEWLINE parameter list	
$BED5	READ parameter list	
$BED5	WRITE parameter list	
$BEDD	CLOSE parameter list	
$BEDD	FLUSH parameter list	

(*) the proper parameter count must be set before using this parameter list

VOLUME DIRECTORY AND SUBDIRECTORY HEADER



GENERAL FILE ENTRY



BASIC.SYSTEM ERROR CODES

Error Code	Error Message
$00	[no error occurred]
$02	RANGE ERROR
$03	NO DEVICE CONNECTED
$04	WRITE PROTECTED
$05	END OF DATA
$06	PATH NOT FOUND
$07	PATH NOT FOUND
$08	I/O ERROR
$09	DISK FULL
$0A	FILE LOCKED
$0B	INVALID PARAMETER
$0C	NO BUFFERS AVAILABLE
$0D	FILE TYPE MISMATCH
$0E	PROGRAM TOO LARGE
$0F	NOT DIRECT COMMAND
$10	SYNTAX ERROR
$11	DIRECTORY FULL
$12	FILE NOT OPEN
$13	DUPLICATE FILE NAME
$14	FILE BUSY
$15	FILE(S) STILL OPEN
$16	DIRECT COMMAND

USEFUL LOCATIONS IN THE BASIC.SYSTEM GLOBAL PAGE

$BE00	BIENTRY	Warm-start entry point to BASIC.SYSTEM
$BE03	DOSCMD	Execute command string at $200
$BE06	EXTRNCMD	JMP to external command parser
$BE09	ERROUT	Call Applesoft error handler
$BE0C	PRINTERR	Print error message (error code in A)
$BE0F	ERRCODE	BASIC.SYSTEM error code number
$BE30/$BE31	VECTOUT	BASIC.SYSTEM output link
$BE32/$BE33	VECTIN	BASIC.SYSTEM input link
$BEF5	GETBUFR	Reserve "A" pages above HIMEM
$BEF8	FREEBUFR	Free all reserved pages
$BEFB	PAGETOP	HIMEM page on boot

Prepublication reviewers say:

". . .thorough subject coverage and technically accurate. . .well researched and well written, in a simple and understandable form!"

"The programming examples for the MLI (machine language interface) commands are excellent. . .indicative of the author's mastery of the subject!"

". . .anticipated and answered many questions I had accumulated about ProDOS. Unquestionably an important reference work and a much needed aid to the ProDOS programmer!"

Apple ProDOS: Advanced Features for Programmers

Gary B. Little

Now, the man who brought you *Inside the Apple //e* does it again with an invaluable new technical guide to Apple ProDOS! Designed for intermediate and advanced users of ProDOS, this thorough, detailed reference shows you how to use and get the most out of the lower-level internal ProDOS commands that are defined by its machine language interface and that are accessible only from 6502 assembly language programs. Valuable programming examples are presented throughout the user-friendly guide including a program for the creation of a high-speed RAMdisk and a program that allows you to easily explore the internal structures of directories. With this book you will:

- use ProDOS to organize files on disks
- use the ProDOS MLI commands to access disk files
- learn how the BASIC.SYSTEM (Applesoft) interpreter works
- write and install your own BASIC.SYSTEM disk commands
- write and install input/output drivers for disk devices
- write and install clock drivers, and MUCH MORE!

CONTENTS

Also Available. . .Optional Diskette!

This valuable diskette contains all of the programs listed in the book (in both source and object code formats) as well as useful ProDOS utilities in the form of "bonus" programs!

ISBN 0-89303-441-X