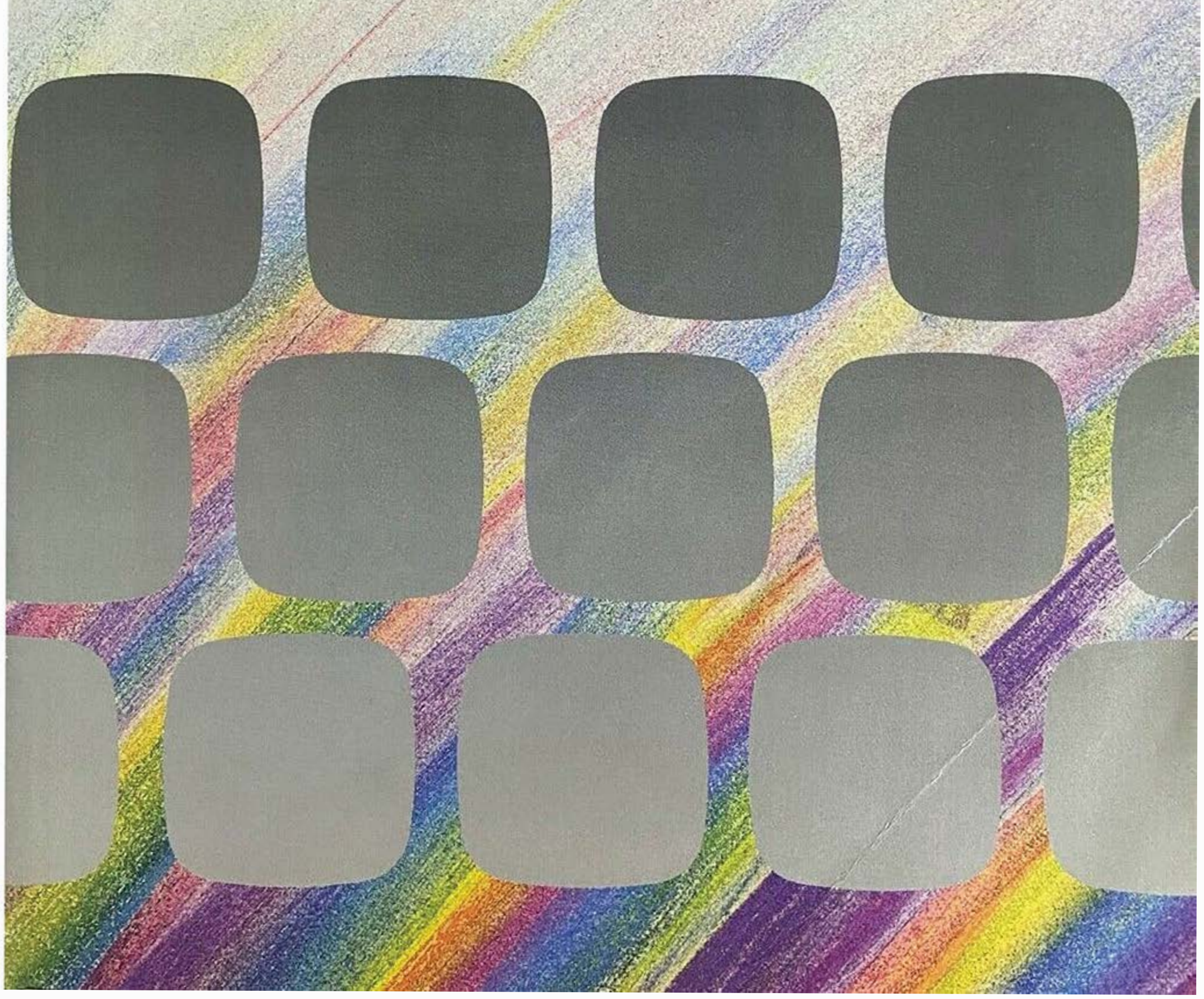
Addison-Wesley Publishing Company

GARY B. LITTLE

Exploring Apple® GS/OS™ and ProDOS® 8



Exploring Apple GS/OS and ProDOS 8

GARY B. LITTLE

Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts • Menlo Park, California • New York •
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn •
Sydney • Singapore • Tokyo • Madrid • San Juan

Library of Congress Cataloging-in-Publication Data
Little, Gary B., 1954–
Exploring Apple GS/OS and ProDOS 8 / Gary B. Little.
p. cm.
Bibliography: p.
Includes index.
ISBN 0-201-15008-5 : $21.95
1. ProDOS (Computer operating system) I. Title.
QA76.76.063L563 1989
005.4'469--dc19 88-17470
CIP

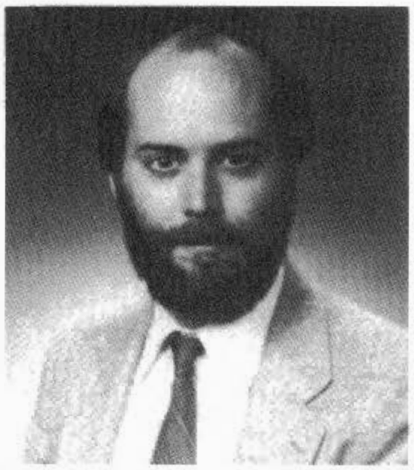
Cover design by Doliber Skeffington
Text design by Kenneth S. Wilson
Set in 10-point Caledonia by Publication Services

ABCDEFGHIJ-AL-89
First printing, January, 1989

This book is dedicated to my father

James Douglas Little

About the Author



Gary Little resides in Belmont, California. Originally from Vancouver, British Columbia, he is a founding member of Apple's British Columbia Computer Society and the famous SAGE organization. Gary is the author of several books for programmers of Apple computers: *Inside the Apple IIe*, *Inside the Apple IIc*, *Exploring the Apple IIGS*, and *Mac Assembly Language: A Guide for Programmers*. He also wrote Point-to-Point, the award-winning Apple II telecommunications program, and developed the international Binary II file format standard.

PREFACE

I've been a fan of ProDOS since Apple first released the 8-bit version in early 1984. (This version is now called ProDOS 8, and it works on all Apple II models; the 16-bit version, GS/OS, works on the Apple IIGS only.) Now, almost five years later, having written two major ProDOS 8 applications and several GS/OS and ProDOS 8 disk utilities, I'm satisfied that I fully understand how these operating systems work, so it's time to share my knowledge with you.

Some of the more interesting topics I cover in this book are

- How the ProDOS file system organizes files on disks
- How to use GS/OS and ProDOS 8 commands to perform disk operations
- How the BASIC.SYSTEM (Applesoft) interpreter works in a ProDOS 8 environment
- How to write and install your own BASIC.SYSTEM disk commands
- How to write GS/OS and ProDOS 8 system programs
- How to communicate with a SmartPort disk controller
- How GS/OS and ProDOS 8 manage interrupts from I/O devices
- How to write and install ProDOS 8 disk and clock drivers
- How to communicate with character devices like the keyboard and the video screen using the GS/OS Console Driver

This book is intended as a reference for intermediate to advanced programmers since I presume you are reasonably familiar with Applesoft BASIC and 6502/65816 assembly language. Even if you're not, you should find the descriptions of how GS/OS and ProDOS 8 handle files and manage peripheral devices useful and revealing.

I've included several programming examples throughout the book to highlight important concepts and to help make the concepts easier to understand. One of these programs is for reading or writing any data block on a disk so that you can easily explore the internal structures of directories and files; another creates a high-speed ProDOS 8 RAMdisk using an area of the Apple II's main memory for block storage; and there are many more. The ProDOS 8 6502 assembly language programs were developed using the Merlin 8/16 assembler (from Roger Wagner Publishing); for the GS/OS 65816 assembly-language programs, I used the assembler in the Apple IIGS

Programmer's Workshop (from the Apple Programmer's and Developer's Association). I review some of the unique features of these assemblers in Appendix I; read this appendix if you are using a different assembler and want to convert the source listings.

Several specialized topics I refer to in this book are not explained in great detail because they really have little to do with ProDOS 8 or GS/OS itself. For more information on these topics, refer to my earlier books, *Exploring the Apple IIGS* (which explains how to use IIGS tool sets), *Inside the Apple IIe*, and *Inside the Apple IIc*. The first book is published by Addison-Wesley and the last two by Brady/Prentice Hall Press. See Appendix III for a bibliography of other useful reference material.

Be aware that this book is *not* a tutorial on how to use the standard Applesoft disk commands that the ProDOS 8 BASIC.SYSTEM interpreter provides. Nor does it describe ProDOS 16 (an early version of GS/OS) in any detail. If you require books on these topics, I suggest you read Apple's own *BASIC Programming With ProDOS* (Addison-Wesley, 1985) and *Apple IIGS ProDOS 16 Reference* (Addison-Wesley, 1987). Instead, I concentrate on the GS/OS and ProDOS 8 commands that are accessible from assembly-language programs only.

Finally, there is no need to manually enter the programs listed in the book. Instead, you can order a disk directly from me that contains these programs (in both source and object code formats) as well as some additional bonus utility programs (described in Appendix IV). For ordering information, see the last page of the book.

* * *

My thanks to Matt Deatherage of the Apple II Developer Technical Support group at Apple Computer, Inc. for his invaluable comments on the technical content of this book prior to publication. Matt helped keep me honest and accurate, two traits one needs to write a useful reference book for software developers.

Thanks in advance to Carole Alden, Steve Stansel, Linda O'Brien, and Abby Genuth of Addison-Wesley who did a superb job in developing, marketing, and selling my last book, *Exploring the Apple IIGS*. I know you'll do just as good a job again.

Gary B. Little
Belmont, California, U.S.A.
September 1988

CONTENTS

Exploring Apple GS/OS and ProDOS 8

CHAPTER 1

An Introduction to GS/OS and ProDOS 8

In this book, we take a close look at the two standard disk operating systems for the Apple II family of computers: GS/OS (Apple IIGS/Operating System) and ProDOS 8 (*Pro*fessional *D*isk *O*perating *S*ystem, 8-bit version).

GS/OS is the primary disk operating system for the Apple IIGS with ROM version 01 or higher. It does not run on any other model in the Apple II family. GS/OS takes advantage of the advanced features of the 16-bit 65816 microprocessor in the IIGS, such as the powerful instruction set and the ability to directly address 16Mb of memory. It is the successor to ProDOS 16, an interim IIGS operating system which Apple provided from the introduction of the IIGS in September 1986 to September 1988. For the sake of compatibility, GS/OS supports all ProDOS 16 commands, so older applications written to run under ProDOS 16 will also run properly under GS/OS.

ProDOS 8 works with the Apple II Plus, IIe, and IIc. It also works on a IIGS running in IIe/IIc emulation mode, and you can switch between it and GS/OS if GS/OS was the operating system you booted from. ProDOS 8 is a fairly simple 8-bit operating system that works in the 6502 (or 65C02) microprocessor's 64K memory space only. Nearly every ProDOS 8 command has a GS/OS equivalent, but the method used to invoke the command is different, at least for assembly-language programs.

GS/OS and ProDOS 8, like all operating systems, manage the flow of data to and from a storage medium, such as a 5.25- or 3.5-inch floppy disk or a hard disk. (GS/OS also manages character devices like the keyboard and the video hardware.) They do this by translating the high-level disk commands an application program uses into the low-level instructions needed to communicate directly with the disk drive controller.

The operating system also defines the data structures used to store groups of related data, called *files*, on the disk; the directories where it stores the names of files (and other file attributes); the method it uses to keep track of what parts of the medium are in use; the method it uses to load itself from disk; and related matters.

GS/OS and ProDOS 8 work well with all disk devices Apple sells for the Apple II family: the Apple 5.25 Drive (and its predecessors), the HD20SC hard disk, the UniDisk 3.5 and the Apple 3.5 Drive, the Apple II Memory Expansion card (a RAMdisk device), and the AppleCD SC CD-ROM drive. ProDOS 8 expects the media used in these devices to be formatted for the ProDOS file system, but GS/OS understands foreign file systems as well (if you provide it with the file system translator files described below).

The two standard drives for Apple II computers are the 5.25-inch drive (140K capacity) and the 3.5-inch drive (800K capacity). They interface to the system through a cable connected to a disk controller card plugged into one of the slots at the back of the Apple IIGS, IIe, or II Plus (slot 6 is recommended for 5.25-inch drives; slot 5 for 3.5-inch drives). On all but the first model of the slotless Apple IIc, the disk controllers for both types of drives are built in. The IIGS also has a built-in disk drive port for both types of drives; you can use it instead of two separate plug-in controller cards.

Apple's 20Mb hard disk, the HD20SC, works with all members of the Apple II family except the Apple IIc. Unlike a floppy drive, its magnetic medium cannot be removed from the drive unit. This device can access information much more quickly and hold much more of it than a 5.25- or 3.5-inch drive. It interfaces to the Apple II through a SCSI (Small Computer System Interface) controller card, one quite different from the one used with floppy drives.

APPLE II OPERATING SYSTEMS—A HISTORY

When the Apple II debuted in 1977, the cassette recorder was the only mass-storage device available to its 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 accessed by the Apple II, so you must keep meticulous written records of what programs are stored where so that you can properly position the tape by hand.

Steve Wozniak, the inventor of the Apple II, was apparently as frustrated with cassette tape as everyone else. In the winter of 1977–1978, he designed a disk controller peripheral card for a standard disk drive unit that was later to be called the Disk II. At the same time, Bob Shepardson, and later Randy Wigginton, Dick Huston, and Rick Auricchio, were busy writing a disk operating system that would make it easy for programmers to create, organize, and access files on the 5.25-inch disk medium that the disk drive uses.

Apple eventually shipped the Disk II, its controller card, and the first released version of the disk operating system (DOS 3.1) in the early summer of 1978. (The Disk II was later renamed the UniDisk, then the Apple 5.25 Drive.) This was probably the most important event in the early history of Apple because it meant, for the first time, useful business software could be written for the Apple II. Such software needs 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 disks.

Several changes were made to DOS 3.1 in the months following its initial release 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 disks with 35 data tracks and with 13 256-byte data sectors per track (for a total of 113.75K of storage, where 1K = 1024 bytes). In fact, the program in ROM on the disk controller card could start up (or *boot*) only disks using this specific 13-sector format.

Apple also released its Pascal operating system in 1979. This system manages files quite differently from either DOS 3.x or ProDOS. To transfer a Pascal textfile to a DOS disk (and vice versa), you can use utility programs available from commercial sources and user groups.

Apple upgraded DOS 3.2.1 substantially in 1980 to support the new 16-sector-per-track formatting scheme used by Apple Pascal. The result was DOS 3.3, a version still current when Apple released ProDOS 8 in early 1984. The formatting change also forced a change in the ROM boot program on the disk controller card. The main advantage of switching to the new formatting scheme was that disks could hold an additional 16.25K of information (for a total of 140K). The main disadvantage was that DOS 3.3 could not read files directly from a DOS 3.2.1-formatted disk (and DOS 3.2.1 disks could not be directly booted). Fortunately, Apple supplied a program called MUFFIN for transferring files from the old disk format to the new one and another program called BOOT13 for booting DOS 3.2.1 disks with a 16-sector drive controller.

Apple first released ProDOS 8, then called simply ProDOS, in January 1984. It runs on any Apple IIe, Apple IIc, or Apple IIGS or on an Apple II Plus with a 16K memory card installed in peripheral slot zero. It also runs on the original Apple II with a 16K memory card if the Applesoft language, not the Integer BASIC language, is installed in ROM. With the release of ProDOS 8, Apple served notice that it would no longer release new software products that use DOS 3.3 and urged independent software developers to do the same. Nevertheless, DOS 3.3 remains a popular operating system, particularly among developers of educational software, and new programs that use it are still quite common.

A ProDOS 8-compatible controller card for the 5Mb ProFile hard disk that Apple had released a couple of years earlier for use with its Apple III system also came out in January 1984. On bootup, ProDOS 8 automatically recognizes the presence of the ProFile and interacts with it just as if it were another 5.25-inch disk device (except that ProDOS 8 knows the ProFile has a much greater storage capacity). The internal structure of the ProDOS file system is such that it can easily deal with even higher-capacity devices; it supports a volume size of up to 32Mb.

Apple later replaced the ProFile with the 20Mb HD20SC hard disk, a Small Computer Systems Interface (SCSI) device. It connects to the system through the Apple II SCSI interface card.

In September 1985, the UniDisk 3.5 drive made its first appearance. Its medium is a 3.5-inch, removable, hard-shell disk with a storage capacity of 800K. ProDOS 8 automatically recognizes its controller card on bootup, so there is no need to install a special driver. (Apple later began shipping a version of the IIc with a built-in controller for the UniDisk 3.5.) Apple also released an expansion slot Apple II Memory Expansion card, which ProDOS 8 recognizes as a RAMdisk on bootup.

Apple announced the Apple IIGS in September 1986. At this time, Apple renamed the original ProDOS as ProDOS 8 and released ProDOS 16, an operating system specifically for the IIGS. Although ProDOS 16 formats disks and stores files on disk in the same way as ProDOS 8 (meaning the two can co-exist on one disk), they are incompatible at the programming level. Apple released ProDOS 16 to take advantage of the full 16Mb memory space the 65816 uses; ProDOS 8 works in a minimal 64K memory space only.

With the IIGS, came the Apple 3.5 Drive, another drive that uses 800K, 3.5-inch, hard-shell disks. The difference between it and the UniDisk 3.5 is that it doesn't have the intelligent processor built in to the UniDisk 3.5, and it works on the IIGS only.

Another version of the IIc debuted in September 1986. This one has a connector you can attach a memory expansion card to. Like the Apple II Memory Expansion card, ProDOS 8 recognizes this card as a RAMdisk.

In September 1988, Apple began providing GS/OS, a new Apple IIGS operating system intended to replace ProDOS 16. Since GS/OS understands all ProDOS 16 commands, all ProDOS 16-based programs will work just fine under GS/OS. But GS/OS also supports a new set of operating system commands that is much more powerful than the ProDOS 16 set. One important new feature of GS/OS is that it is can access disks formatted for the standard ProDOS file system *and* disks formatted for foreign file systems like High Sierra (for CD-ROMs), HFS (used by the Macintosh), and MS-DOS. Access to foreign operating systems is enabled by putting file system translator (FST) modules on the GS/OS system disk. In the initial release of GS/OS, Apple provided FSTs for the ProDOS and High Sierra file systems.

Another model of the IIc, the Apple IIc Plus, also came out in September 1988. It features a built-in 3.5-inch drive that works with ProDOS 8.

Early versions of ProDOS 8 suffered from several minor but annoying bugs that were removed in later versions. As of this writing, the current version is 1.7. GS/OS, a much more complex operating system, is not now nearly as stable as ProDOS 8. Apple releases new versions about twice a year.

COMPARING PRODOS 8 WITH DOS 3.3

DOS 3.3 is made up of two main modules: the I/O (input/output) driver, which communicates directly with a 5.25-inch disk controller, and the Applesoft command interpreter, which parses and executes the Applesoft disk commands that DOS 3.3 provides (OPEN, READ, CATALOG, and so on). The equivalent modules in ProDOS 8 are split into two program files called PRODOS (the I/O driver) and BASIC.SYSTEM (the Applesoft command interpreter). On many application disks, PRODOS automatically loads

BASIC.SYSTEM when the disk starts up. Thus it is necessary to compare DOS 3.3 with the PRODOS–BASIC.SYSTEM combination and not simply with PRODOS proper.

Table 1-1 gives short descriptions of the Applesoft disk commands that BASIC.-SYSTEM and DOS 3.3 provide. 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. (We review these parameters in Chapter 5.) Moreover, some commands behave slightly differently in one system from how they behave in the other.

Not surprisingly, the more powerful PRODOS–BASIC.SYSTEM environment occupies a lot more memory space than DOS 3.3 does; in fact, it uses almost twice as much space. Fortunately, most of ProDOS 8 resides in a 16K bank-switched RAM space that does not conflict with the space the Applesoft interpreter uses. This space is built in to an Apple IIe, IIc, and IIGS and can be added to an Apple II or Apple II Plus by installing a 16K memory card in slot zero. Two side effects of the use of this space by ProDOS 8 are that ProDOS 8 cannot function with a program that uses the memory card for data storage or with Integer BASIC, the original version of Apple BASIC. In a DOS 3.3 environment, Integer BASIC loads into the same bank-switched RAM area ProDOS 8 uses and then is selected by throwing a special software-controlled switch.

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 an open file uses; 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 first starts up, it automatically sets up three such buffers; a different number (from 1 to 16) can be reserved 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 8, 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, ProDOS 8 lowers HIMEM by 1024 bytes and assigns the buffer 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 eight files can be open simultaneously.) Because ProDOS 8 uses this dynamic space allocation method, it is not possible to use the DOS 3.3 technique of reserving a safe space for an assembly-language program by lowering HIMEM and storing the program between the current and previous HIMEMs. But there is an alternative method for freeing up space above HIMEM, and we examine it in Chapter 5.

Important Features of ProDOS 8 and BASIC.SYSTEM

A PRODOS–BASIC.SYSTEM environment supports several useful features that improve program execution speed and permit easy integration of non-Apple devices into the system. Here are some of the more important features.

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

		Availability	
Command	*Description*	*ProDOS 8*	*DOS 3.3*
APPEND	Opens a file and prepares to add data to it	Yes	Yes
BLOAD	Loads a file (usually binary)	Yes	Yes
BRUN	Loads and executes an assembly-language program that is in a binary file	Yes	Yes
BSAVE	Saves a file (usually binary)	Yes	Yes
CATALOG	Lists all the files on the medium (long form)	Yes	Yes
CLOSE	Closes a file	Yes	Yes
DELETE	Deletes a file	Yes	Yes
EXEC	Executes commands from a textfile	Yes	Yes
IN#	Redirects character input	Yes	Yes
LOAD	Loads an Applesoft program	Yes	Yes
LOCK	Locks a file	Yes	Yes
NOMON	[Permitted but ignored under ProDOS 8]	Yes	Yes
OPEN	Opens a file	Yes	Yes
POSITION	Prepares to read from or write to a specific position in the file	Yes	Yes
PR#	Redirects character output	Yes	Yes
READ	Reads from a file	Yes	Yes
RENAME	Renames a file	Yes	Yes
RUN	Loads and executes an Applesoft program (or, if no filename is specified, executes the program in memory)	Yes	Yes
SAVE	Saves an Applesoft program	Yes	Yes
UNLOCK	Unlocks a file	Yes	Yes
VERIFY	Checks for the existence of a file; if no filename is specified, displays a copyright notice	Yes	Yes
WRITE	Writes to a file	Yes	Yes

Table 1.1 Continued

Command	Description	Availability ProDOS 8	Availability DOS 3.3
- (dash)	Executes an Applesoft, binary, text, or system file	Yes	No
BYE	Transfers control to another system program	Yes	No
CAT	Lists the files on the medium (short form)	Yes	No
CHAIN	Transfers control to another Applesoft program while maintaining the current variables	Yes	No[a]
CREATE	Creates a file (usually a directory file)	Yes	No
FLUSH	Writes the contents of a file buffer to the medium	Yes	No
FRE	Performs Applesoft garbage collection	Yes	No[b]
PREFIX	Sets up the name of the active directory	Yes	No
RESTORE	Restores Applesoft variables from a file	Yes	No
STORE	Saves Applesoft variables to a file	Yes	No
FP	Initializes Applesoft mode	No	Yes
INIT	Formats a disk	No[c]	Yes
INT	Initializes Integer BASIC mode	No	Yes
MAXFILES	Creates space for file buffers	No	Yes
MON	Enables the display of DOS operations	No	Yes

NOTES:
[a]You can chain Applesoft programs under DOS 3.3 by loading and calling a subroutine called CHAIN that is stored on the DOS 3.3 master disk.
[b]You can use the Applesoft FRE command to garbage-collect under DOS 3.3 (and ProDOS 8). It executes much more slowly than the corresponding ProDOS 8 command, however.
[c]Under ProDOS 8, you format a disk using a separate program on the ProDOS 8 master disk (either Filer or System Utilities).

Machine Language Interface. Probably the most important feature of ProDOS 8 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 perform such standard file-handling chores as opening, reading, writing, and closing. The calling parameters for each command have been carefully defined by Apple. We take a close look at the MLI in Chapter 4.

Date-Stamping of Files. Whenever ProDOS 8 creates or writes to files, it reads the current time and date from a clock device (if one is installed in the system) and stores the information in the file's directory entry on disk. When the disk is cataloged, the time and date of creation and of last modification appears next to the filename. ProDOS 8 works with the built-in IIGS clock and clock cards that emulate the command set of the Thunderware Thunderclock. As we see in Chapter 8, it is possible to install clock drivers for other types of clock cards as well.

Disk Controller Card and Device Driver Protocols. One annoying trait of DOS 3.3 is that it is very difficult to integrate foreign disk devices (non-Apple-brand hard disks, higher-density floppy disk drives, and so on) into the system. Not so with ProDOS 8. Apple has published a disk controller protocol recognized by ProDOS 8 that permits such devices to be automatically installed at bootup time. 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 to pass parameters to a ProDOS 8 disk driver subroutine and how the driver returns error codes to the caller. We see how to write a disk driver subroutine in Chapter 7.

Improved Interrupt Handling. In Chapter 6, we see that ProDOS 8 automatically installs its own internal interrupt-handling subroutine that takes control whenever an I/O device generates an active IRQ (interrupt request) signal. This subroutine will, in turn, call subroutines you can install to service such interrupts. This means it is very simple to integrate an interrupt subroutine even though another one may already be active.

Hierarchical Directory Structure. Using ProDOS 8, 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 so that each is contained within another (called the *parent*); the path of directories ultimately leads back to the root directory (also called the *volume* directory). The root directory is the one created and named when the disk is first formatted. We analyze the hierarchical structure of directories in Chapter 2.

/RAM Disk Device. The Apple IIGS, Apple IIc, and Apple IIe (with an extended 80-column text card) have 64K of *auxiliary* memory in addition to the 64K of main memory normally used for program storage. ProDOS 8 uses this memory space for file storage just as if it were storage space on a floppy disk or hard disk. The RAM medium is called a RAMdisk. The main differences between using the RAMdisk and conventional disk media are that I/O operations execute much more quickly (after all, there are no mechanical parts to move about) and that the RAMdisk vanishes when you turn the power off. As we see in Chapter 2, each disk in the system has a name associated with it (the *volume* name). The volume name for the RAMdisk is /RAM. We examine the characteristics of /RAM in Chapter 7. We also examine the /RAM5 RAMdisk you can set up on an Apple IIGS. This RAMdisk uses memory on a card you put in the IIGS's special memory expansion slot.

Extensibility of BASIC.SYSTEM. The BASIC.SYSTEM program defines a reasonably simple method you can use to add more commands to the BASIC.SYSTEM command set. We see how to do this in Chapter 5.

"Separation of Powers." Unlike DOS 3.3, the low-level ProDOS 8 command interpreter that performs all fundamental disk I/O operations is not mixed with the BASIC.SYSTEM interpreter that provides the set of "English" disk commands used in an Applesoft program. This means if you wish to write another language interpreter, or a 100 percent 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 BASIC.SYSTEM command very popular with people who do not like to type. It executes either an Applesoft program file (just as RUN does), a binary file (BRUN), or a textfile (EXEC) by automatically determining what type of file has been specified and then performing the steps needed to execute such a file. Dash can also execute system program files like BASIC.SYSTEM. (See Chapter 5 for a description of system programs. Briefly, a system program is a standalone 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, you can use the ,@# suffix (where # represents a line number) with the BASIC.SYSTEM RUN command to load a program and then run it beginning at any line number. Moreover, you can use the ,E# suffix (where # represents a memory address) 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#; when reading a textfile, use it to skip over a specified number of fields (a field is a group of characters followed by a carriage return). We discuss parameters recognized by BASIC.SYSTEM in Chapter 5.

Speed. ProDOS 8 performs disk I/O operations on a 5.25-inch disk at the rate of about 8K bytes per second. This is significantly faster than the DOS 3.3 rate of about 1K bytes per second. Furthermore, BASIC.SYSTEM includes a version of the FRE command that garbage-collects Applesoft string variables much faster than the Applesoft command of the same name; BASIC.SYSTEM also garbage-collects 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 can take several minutes. (See Chapter 4 of *Inside the Apple IIe* for a description of the garbage collection process.)

File Size and Volume Size. ProDOS 8 can deal with files that hold up to 16Mb and with block-structured (disklike) devices that hold up to 32Mb of information. DOS 3.3 volumes cannot exceed 400K.

COMPARING GS/OS WITH PRODOS 8

The fundamental difference between GS/OS and ProDOS 8 is, of course, that GS/OS works on the Apple IIGS only. This is because GS/OS is written in 65816 assembly language, and it uses IIGS-specific tool sets like the Memory Manager and the System Loader. Although most GS/OS commands have ProDOS 8 equivalents, several unique commands make GS/OS a much richer programming environment.

Listed below are the most important differences between the GS/OS and ProDOS 8 programming environments.

1. A GS/OS application can call GS/OS commands from anywhere within the 16Mb memory space of the 65816. A ProDOS 8 application can call ProDOS 8 commands from the first 64K of memory only.
2. GS/OS applications are stored in relocatable *load files*, meaning they can be loaded and run at any memory location. ProDOS 8 applications are simple binary images of program code, so they generally run at only one memory location. (It is possible to write relocatable ProDOS 8 applications, but it makes programming so difficult that most programmers don't bother trying.)
3. GS/OS applications use the Apple IIGS Memory Manager tool set to ensure they won't use memory areas already in use by other system resources. ProDOS 8 applications are responsible for their own memory management, so programmers must be aware of what areas ProDOS 8 occupies.
4. GS/OS has 33 pathname prefixes that can be referred to by special shorthand names like 1/ or 28/. ProDOS 8 has only one pathname prefix (called the *default prefix*).
5. GS/OS identifies disk devices by name, whereas ProDOS 8 identifies them by slot and drive number.

6. GS/OS has a built-in disk-formatting command (Format) and a built-in cataloging command (GetDirEntry). ProDOS 8 does not.
7. GS/OS has a command that lets you move files from one directory to another (ChangePath). ProDOS 8 does not.
8. Under GS/OS, an application can determine its own name with the GetName command. ProDOS 8 has no similar command although an application can deduce its name by inspecting a pathname buffer.
9. GS/OS has an enhanced Quit command that an application can use to pass control directly to the system program that called it, to pass control to any specified system program, or to call another system program almost as if it were a subroutine. The ProDOS 8 QUIT command can pass control only to a ProDOS 8 program selector.
10. GS/OS can create and deal with *extended files*, but ProDOS 8 cannot. Extended files (sometimes called *resource files*) are made up of two logical parts: a data fork and a resource fork. The data fork generally contains application-specific data, and the resource fork generally contains a group of data structures, called resources, that define such things as icons, text strings, and alert box templates.
11. GS/OS uses file system translators (FSTs) to provide an application with transparent access to disk volumes that use non-ProDOS file systems, such as High Sierra for CD-ROM or Macintosh HFS, as well as the ProDOS file system. ProDOS 8 only works with disks formatted for the ProDOS file system.
12. GS/OS lets an application access character-oriented devices, like the video screen, keyboard, modem, and printer, using the same types of commands you would use to access disk files. Under ProDOS 8, the application must use completely different techniques to access character-oriented devices, many of which require an understanding of the low-level hardware interface.
13. GS/OS accesses disks faster than ProDOS 8 because it uses disk-caching techniques and more efficient 65816 code. It can also format disks with a lower block interleave ratio (2:1 instead of 4:1), thus improving the effective data transfer speed.
14. GS/OS allows an unlimited number of open files and active volumes, and it imposes no limit on the number of devices per slot. ProDOS 8 allows only 8 open files, 14 active volumes, and 2 devices per slot.
15. GS/OS, because it uses file system translators, can access non-ProDOS volumes up to 2048Gb (gigabytes) in size and can deal with files up to 4096Mb long. ProDOS 8 volumes cannot exceed 32Mb, and files cannot be longer than 16Mb.
16. GS/OS does *not* come with a BASIC language interpreter equivalent to ProDOS 8's BASIC.SYSTEM program.

CHAPTER 2

Disk Volumes and File Management

In this chapter, we familiarize you with the concept of a file and explain how the ProDOS file system organizes files on the disk drive medium. You need to know the details of the ProDOS file system if you want to better comprehend the internal GS/OS and ProDOS 8 file-handling commands described in Chapter 4. (GS/OS works with non-ProDOS file systems as well, but most users will be using it with disks formatted for the ProDOS file system.)

The concept of a file is fundamental to all disk operating systems. A file is just a collection of data that can define an executable program, a letter to the editor, a spreadsheet template, or any other document a program can deal with. The general structure of a file is defined by the operating system itself; the operating system also provides the various commands for accessing the file in different ways: create, open, read, write, close, destroy, rename, and so on.

NAMING FILES

When you first save a file to disk, you must assign it a unique filename that a program can use to identify it thereafter. A ProDOS filename can be up to 15 characters long. It must begin with an alphabetic letter (A to Z), but the other characters may be any combination of letters, digits (0 to 9), and periods (.). You can use lowercase letters, too, but ProDOS 8 and GS/OS automatically convert them to uppercase when dealing with the ProDOS file system. 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

A common mistake that arises in naming files is the use of the space as a word separator (as in the second example). This is permitted with DOS 3.3 but not ProDOS. Periods, not spaces, must be used to separate words in a filename to improve readability. Some programs, like AppleWorks, allow users to enter spaces in filenames, but they internally convert the spaces to periods before using the filenames with operating system commands.

GS/OS, of course, can work with disk volumes that have been formatted for foreign operating systems (such as Macintosh HFS, MS-DOS, and High Sierra) if the appropriate file system translator files are on the boot disk. The naming rules for these file systems are different from those for the ProDOS file system. Macintosh HFS, for example, allows names up to 31 characters long; these names can contain any printable ASCII character except the colon. Refer to the appropriate operating system reference manuals for the naming rules for other operating systems.

DIRECTORIES AND SUBDIRECTORIES

When you save a ProDOS file to disk, you can store it in any one of several *directories* that may have been created on the disk. These directories are analogous to file folders in that they are often used to hold groups of related files. (In fact, they are often referred to as folders instead of directories.) For example, you may create one directory to hold word processing documents, and another to hold Applesoft programs. The ability to create separate directories on the same disk makes it much easier to efficiently organize large numbers of files.

When you first format a disk, only one directory, the *volume* directory or *root* directory, exists; you name it as part of the formatting procedure. (The rules for naming directories are the same as for naming standard files.) The volume directory for a ProDOS-formatted disk can hold the names of up to 51 files (whereas a DOS 3.3 directory can hold 105 files).

You can create additional directories (called *subdirectories*) within the volume directory using the GS/OS or ProDOS 8 Create command. Indeed, you can even create subdirectories within subdirectories. A 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. Most modern file systems, including Macintosh HFS, MS-DOS (version 2.x and higher), and CD-ROM's High Sierra, use similar hierarchical directory structures.

To specify the directory a file is to be saved in, you normally add a special prefix to the filename to create a unique identifier called a *pathname*. A pathname comprises the names of a series of directories, beginning with the name of the volume directory and continuing with the names of all the directories you must pass through to reach the target directory, followed by the filename itself. Each directory name is separated from the next by a special separator character, and a separator must precede the name of the volume directory.

Under GS/OS, the separator character can be either a slash (/) or a colon (:). Under ProDOS 8, it must be a slash. We use the slash as the separator character in the following discussion.

The directory names in a pathname chain must define a continuous path—that is, each directory specified must be contained within the preceding directory. For example, suppose a disk has a volume directory called BASEBALL and two subdirectories within BASEBALL called AMERICAN and NATIONAL. (Figure 2-1 shows such a directory arrangement.) If you want to save a file called NY.YANKEES in the AMERICAN subdirectory, you would specify the following pathname:

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

If you had specified the name NY.YANKEES itself, the file would have been saved in the current directory, which is usually the volume directory (unless it has been changed using the SetPrefix command described next).

Under GS/OS, you can specify a device name, instead of a volume directory name, when forming a pathname. Device names begin with a period (.) and can be between 2 and 31 characters long. Examples of device names are .SCSI1, .DEV4, and .APPLEDISK3.5A. If the NY.YANKEES file in the above example is on the disk in the drive whose device name is .SCSI1, you could identify it with the following pathname instead:

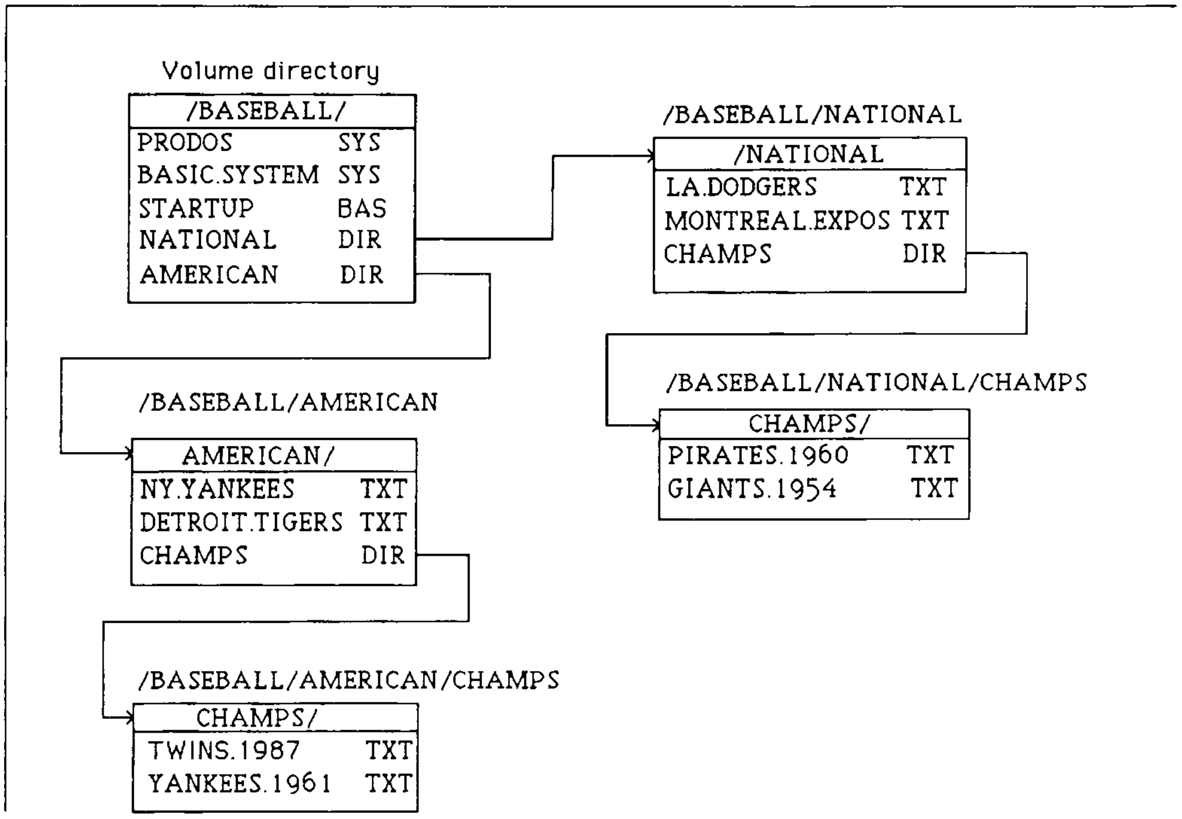
```
.SCSI1/AMERICAN/NY.YANKEES
```

This technique cannot be used with ProDOS 8 because ProDOS 8 does not use device names.

As we saw above, the separator for a GS/OS pathname can be a slash or a colon, but you can't use both as separators in a single pathname. GS/OS determines what the separator is by scanning the pathname from left to right until it finds a slash or colon; the character it finds is the separator.

If the GS/OS separator is a colon, you can use slashes in GS/OS filenames, which is important if you're accessing files on a non-ProDOS disk volume through a GS/OS file system translator. (Macintosh files, for example, can include slashes.) The reverse is not true, however: If the separator is a slash, you cannot use a colon in a filename. Thus it's best to always use the colon as a pathname separator in GS/OS applications.

Figure 2-1 The ProDOS hierarchical directory structure



Prefixes

If most of the files you are using are in the same subdirectory, it becomes annoying to have to specify the same chain of directory names every time you want to access a file.

To abate this annoyance, GS/OS and ProDOS 8 have a SetPrefix command you can use to set the chain of directory names to which any filename specified in a command will be automatically appended. The chain is the *default prefix* and cannot be more than 64 characters long under ProDOS 8 or 8K characters long under GS/OS.

For example, if you set the default prefix to /BASEBALL/AMERICAN/, you can refer to any file in the directory at the end of this path (such as NY.YANKEES) by filename only.

A name that is a continuation of the default prefix could also be specified to access files in lower-level subdirectories; such a name is called a *partial pathname*. If the default prefix has the value just described, and if AMERICAN contains a subdirectory called CHAMPS that contains a file called TWINS.1987, you could access the file by specifying a partial pathname of CHAMPS/TWINS.1987. Here the pathname is *not* preceded by a slash.

Under GS/OS (but not ProDOS 8), the default prefix also goes by the shorthand name of 0/. This means 0/ is equivalent to /BASEBALL/AMERICAN/ if you've used SetPrefix to assign /BASEBALL/AMERICAN/ to the 0/ prefix. As Table 2-1 shows, GS/OS supports 32 different prefixes you can refer to by a number followed by a slash (0/ through 31/) and a boot prefix called */. GS/OS sets */ to the name of the disk you booted from; you cannot change */. 1/ and 9/ identify the directory in which the current application resides, and 2/ identifies the directory containing system library files. You can change 1/, 2/, and 9/ with the GS/OS SetPrefix command, but it's probably best to leave them alone. Use the user-definable prefixes if your application needs to identify a particular directory using the convenient GS/OS shorthand notation.

ProDOS 8 prefixes can be up to 64 characters long, including the preceding slash. Partial pathnames can be up to 64 characters long as well. GS/OS has both short and long prefixes. Short prefixes (*/ and 0/ through 7/) can be up to 64 characters long and long prefixes (8/ through 31/) can be up to about 8192 characters long.

A good feature of GS/OS and ProDOS 8 is that whenever a command must locate a file described by a pathname, it searches every disk available to the system. Contrast this with the DOS 3.3 environment where you must explicitly specify the drive and slot number for the file before you can access it (using the ,S# and ,D# parameters). BASIC.SYSTEM, for reasons of compatibility, also permits the use of the ,S# and ,D# parameters. If you specify a filename or partial pathname in a command line, and no default prefix has yet been defined, or if either the slot or drive parameter is used, BASIC.SYSTEM automatically uses the name of the volume directory for the disk in the specified slot and drive (or their defaults) to create the full pathname.

The advantages of using subdirectories are often not readily apparent to users of floppy disks but are obvious to hard disk users. Hard disks have enough room for hundreds of files. If all the files were held in one directory, you might have to wait a long time to spot your file when the disk was cataloged, and even then you could well miss it among the other files. Fortunately, the hierarchical directory structure ProDOS uses allows related files to be grouped within the same subdirectory for easy access.

FUNDAMENTAL FILE-HANDLING CONCEPTS

As we see in Chapter 4, GS/OS and ProDOS 8 both include a command interpreter that understands a variety of file-handling commands. The most common commands used with existing files are

```
Open     open a file for I/O operations
Read     read data from the file
Write    write data to the file
Close    close the file to I/O operations
```

(Four similar commands are also available from Applesoft when you are using the BASIC.SYSTEM interpreter in a ProDOS 8 environment.) Let's review each of these fundamental file-handling operations.

Table 2-1 Standard prefix numbers for GS/OS

Prefix Number	*Description*
*/	The boot prefix. This is the name of the volume GS/OS was booted from. This prefix cannot be changed by the user.
0/	The default prefix. GS/OS automatically attaches it to any filename or partial (rather than full) pathname you specify.
1/	The application prefix. The pathname of the directory containing the current application program.
2/	The system library prefix. The pathname of the directory containing library modules used by the current application. For a standard GS/OS boot disk, this is /MYDISK/SYSTEM/LIBS.
3/ to 8/	User-definable.
9/	Same as for 1/.
10/ to 31/	User-definable.

Opening a File

You must open a file before you can access it. Do this by using the Open command and specifying the name of the file you wish to open. The operating system opens a file by first locating it on the disk and then setting up a special buffer area for it in memory.

Part of the file buffer holds information that tells the operating system where the file data is located on disk; another part holds the most recently accessed portion of the file. Whenever you request a file I/O operation, the operating system determines whether the portion of the file to be accessed is already sitting in the file buffer. If it is, the operating system 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 occur much more quickly than if unbuffered disk I/O techniques were used.

ProDOS 8 can open a file at one of sixteen different system file levels (numbered from 0 to 15); GS/OS supports 256 different system file levels (0 to 255). Under ProDOS 8, an application can specify the system file level by storing the level number at a particular memory location ($BF94) just before opening the file. Under GS/OS, the application must use the SetLevel command instead. The default system file level is 0.

The main advantage of having different file levels available is to make it easier to write supervisory or executive programs. These types of programs typically open their own work files, pass control to user programs, and regain control when the user programs end. If a supervisory program bumps the file level by one before a user program takes over, its work files can't be inadvertently closed by the user program, even if the program tries to close all open files (unless the user program breaks a rule and decrements the file level).

Reading and Writing a File

When the operating system opens a file, it initializes two important internal pointers it uses for keeping track of the size of the file and the last position in the file that an application accessed. These are called the *EOF* and *Mark* pointers. 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 occurs (the "end of data" error). EOF normally changes only if an application writes information to the end of a file; when this happens, EOF automatically increases by the appropriate number of bytes, and if necessary, the operating system allocates more blocks on the disk. But as we see in Chapter 4, GS/OS and ProDOS 8 also have a SetEOF command you can use to set EOF to any specific value.

Mark is 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 you first open a file, but it automatically increases as information is read from or written to the file. For example, if Mark is currently 10 (that is, it is pointing to the 11th byte in the file), and you read or write 14 more bytes of information, Mark advances to 24.

It is also possible to explicitly set Mark to any position in the file so that you can access the file *randomly*. This means a program can retrieve a record from a file containing fixed-length records very quickly because it is not necessary to read through all preceding records first.

Closing a File

You must close a file when you're finished dealing with it. This ensures that any data written to the file buffer, but not yet stored on the disk itself, is actually stored on the disk. It also updates file information, such as size, in the directory.

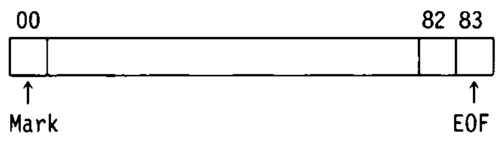
Although it is not necessary to close a file immediately after you're finished with it (you could wait until the program is about to end), it makes good sense to do so to reduce the risk of data loss in the event of an unexpected power loss or a system reset. Moreover, ProDOS 8 allows only so many files 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 you open a file. Another compelling reason to close unused files is to free up memory space; each open file reserves a buffer area that is made available to the system when you close the file.

GS/OS DISK CACHING

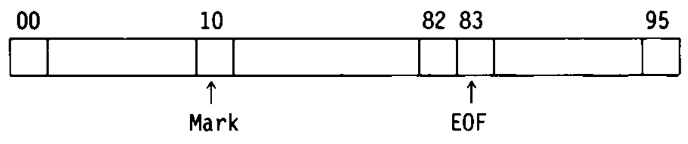
To speed up disk operations like the ones described above, GS/OS supports the caching of disk blocks. The cache is an area of memory where GS/OS saves copies of

Figure 2-2 The ProDOS 8 and GS/OS EOF and Mark pointers

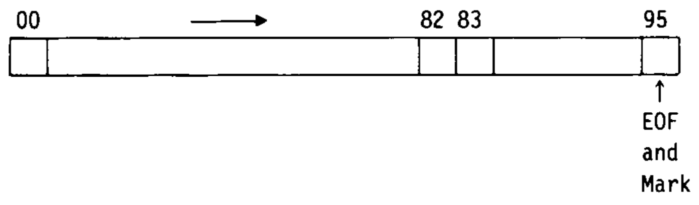
(a) EOF and Mark after an 83-byte file has been opened:



(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.

disk blocks when it first reads them from disk. GS/OS also puts in the cache copies of blocks it writes to disk. Once a block is in the cache, GS/OS can quickly get it from memory whenever it needs to read the block again; GS/OS doesn't have to access the relatively slow disk drive to get it.

The user usually sets the size of the disk cache with the Disk Cache desk accessory. Like any desk accessory, Disk Cache appears in the Apple menu of most applications which use the Apple IIGS Menu Manager, including the Finder. An application can also set the cache size by calling the GS/OS ResetCache command after saving the new cache size to Battery RAM with the WriteBParam function (see Chapter 4). Generally speaking, the larger the cache, the better GS/OS will perform, but less memory will be available to applications.

In most cases, the block cache is not large enough to hold all the blocks which GS/OS may want to cache. When the cache is full, GS/OS throws out the least recently used block to make room for the next block.

The GS/OS Read and Write commands (see Chapter 4) let you specify whether specific disk blocks are to be cached or not. Applications should try to cache blocks they expect to frequently access.

PRODOS FILE MANAGEMENT

Disk operating systems use different methods to organize files on disk and 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 investigate the following topics:

- The structure of a ProDOS-formatted disk
- The structure of the ProDOS volume bit map
- The structure of ProDOS directories and subdirectories
- The structure of a ProDOS directory entry
- The indexing schemes ProDOS uses to locate files

ProDOS uses the same general method to organize files on every block-structured, mass-storage device it works with (such as an Apple 5.25 Drive, an Apple 3.5 Drive, an HD20SC, and the /RAM volume). Specific differences arise because the storage capacities of these different devices vary. Furthermore, the sizes of two important data structures stored on the media, the volume directory and the volume bit map, might be different. We generally focus on the Apple 5.25 Drive (and its 5.25-inch floppy disks) in this section; any specific differences for other devices that are not obvious will be mentioned.

FORMATTING THE DISK MEDIUM

Before you can use a floppy disk (or any other disk medium) with GS/OS or ProDOS 8, it must be formatted into a state that GS/OS or ProDOS 8 recognizes. You can format a disk with the Filer or System Utilities program on Apple's ProDOS 8 master disk or the Apple IIGS Finder. GS/OS also has a Format command that applications can use to format a disk.

The method used to format a disk depends on the nature of the disk device. When you format a 5.25-inch floppy disk, for example, templates for 35 *tracks* on the disk 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 at the outside edge and track 34 at the inside edge. The operating system 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 that activate a *stepping motor* that controls the motion of a metal arm the read/write head is connected to. This arm moves along a radial path beginning at the outside edge of the disk (track 0) and ending at the inside edge (track 34).

Each of the 35 tracks formatted on a disk is subdivided into 16 smaller units, or *sectors*. A sector is the smallest unit of data that can be written to or read from the disk 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 disk 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 disk sectors. An initialized disk 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 disk since the operating system disk driver subroutine automatically maps block numbers to actual physical locations on the disk.

DISK VOLUMES AND DISK DRIVES

A formatted floppy disk that is on line (placed in a system disk drive and ready to be accessed) is often called a *disk volume*. ProDOS-formatted volumes have names that follow the same naming rules as files, but they are often preceded with a slash (/) to make them more recognizable as volume names.

Disk drives themselves also have unique identifiers. ProDOS 8 assigns a *unit number* to each disk device it finds in the system. The value of the unit number is formed from the slot number of the disk drive controller card and the drive number. Figure 2-3 shows the format of the unit number byte.

In Figure 2-3, SLOT may actually be the number of a phantom, or logical, slot if the system contains nonstandard disk devices like RAMdisks. The unit number for the /RAM volume on a IIe, IIc, or IIGS is $B0, for example; in other words, /RAM is the logical slot 3, drive 2 device.

DR indicates the drive number: It is 0 for drive 1 and 1 for drive 2. More than two drives may be connected to the port 5 SmartPort. In this case, ProDOS 8 logically assigns the next two drives to slot 2, drive 1 and slot 2, drive 2. ProDOS 8 ignores all SmartPort drives after the first four.

GS/OS assigns unique device reference numbers to the disk devices (and character devices) it finds—these numbers are consecutive integers beginning with 1. It also assigns device names to each device; examples are .APPLEDISK3.5A, .SCSI1, and .DEV3. (These names can be from 2 to 31 characters long.) GS/OS does not use the unit number scheme that ProDOS 8 uses.

(See Chapter 7 for more detailed information on disk devices and naming conventions.)

DISK VOLUME BLOCK USAGE

We are now ready to examine the method ProDOS uses to manage files on a disk. Our discussion includes an analysis of the structures of the directories that hold information about files, of the volume bit map that keeps track of block usage on the disk, and of the index blocks that contain the locations of the data blocks each file uses.

But before we continue, keep in mind that the following descriptions relate only to the ProDOS file system and not to its predecessor, DOS 3.3, the Apple Pascal file system, or any other foreign operating system.

Figure 2-3 The format of a ProDOS 8 unit number byte

```
 7   6    5    4   3    2    1    0
+---+-----------+-------------------+
|DR |   SLOT    |     [Unused]      |
+---+-----------+-------------------+
```

As we have seen, a total of 280 blocks, holding 140K of data, are available on a ProDOS-formatted 5.25-inch disk. If a standard disk-formatting program is used, however, seven of these blocks (0–6) are not available for use by files because ProDOS reserves them for special purposes. Figure 2-4 shows the usage of blocks on freshly formatted 5.25- and 3.5-inch disks.

Blocks 0 and 1 contain a short assembly-language program that the firmware on the drive controller card loads into memory and executes whenever it boots a disk. This program is called the *boot record*, and it locates, loads, and executes a special system file called PRODOS if it finds it on the disk. (A system file has a file type code of $FF and a CATALOG mnemonic of SYS. We discuss file type codes later in this chapter.) PRODOS is the program ultimately responsible for installing and activating the operating system. (See Chapter 3.)

Blocks 2 through 5 are the blocks containing the volume directory for the disk. We describe the structure of this directory later in this chapter.

Block 6 is the first volume bit map block for the disk. Each bit in the map indicates whether the block it corresponds to is free or in use. ProDOS reserves one bit map block for each 2Mb (4096 blocks) of storage space.

The blocks past the end of the bit map block (or blocks), a total of 273 for a 5.25-inch disk or 1593 for a 3.5-inch disk, are free for use by files stored on the disk.

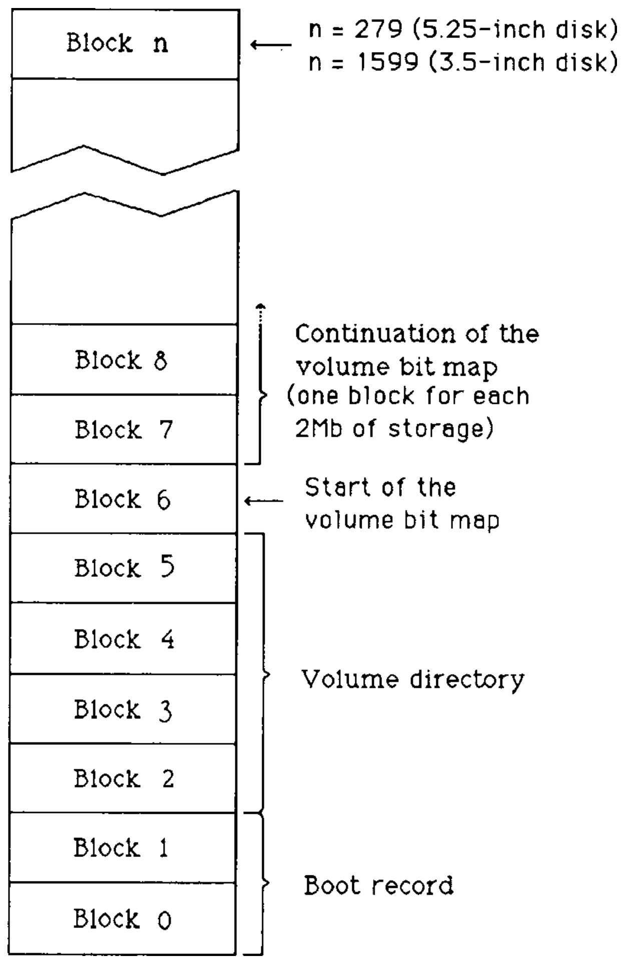
THE VOLUME BIT MAP

The operating system accesses the volume bit map to determine the status of each block on the disk. It reads the bit map whenever it allocates new space to a file so that it can quickly locate free blocks on the disk. It writes to the bit map to reserve new file blocks (this occurs when an existing file grows or a new one is saved) or to free up blocks (this occurs when a file shrinks or is deleted).

Standard formatting routines use block 6 as the first block for a disk's volume bit map. But block 6 is only the conventional location for the bit map; 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 in block 3. As we see in the next section, the block number for the first bit map block appears in the directory header that describes the characteristics of the disk volume.

For a 5.25-inch disk, only the first 35 bytes (280 bits) in the volume bit map block are 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, like a hard disk, a continuation of the bit map can be found in the blocks on the disk immediately following the first one used. For example, the old 9728-block Apple ProFile hard disk

Figure 2-4 Map of block usage on a 5.25-inch disk and a 3.5-inch disk

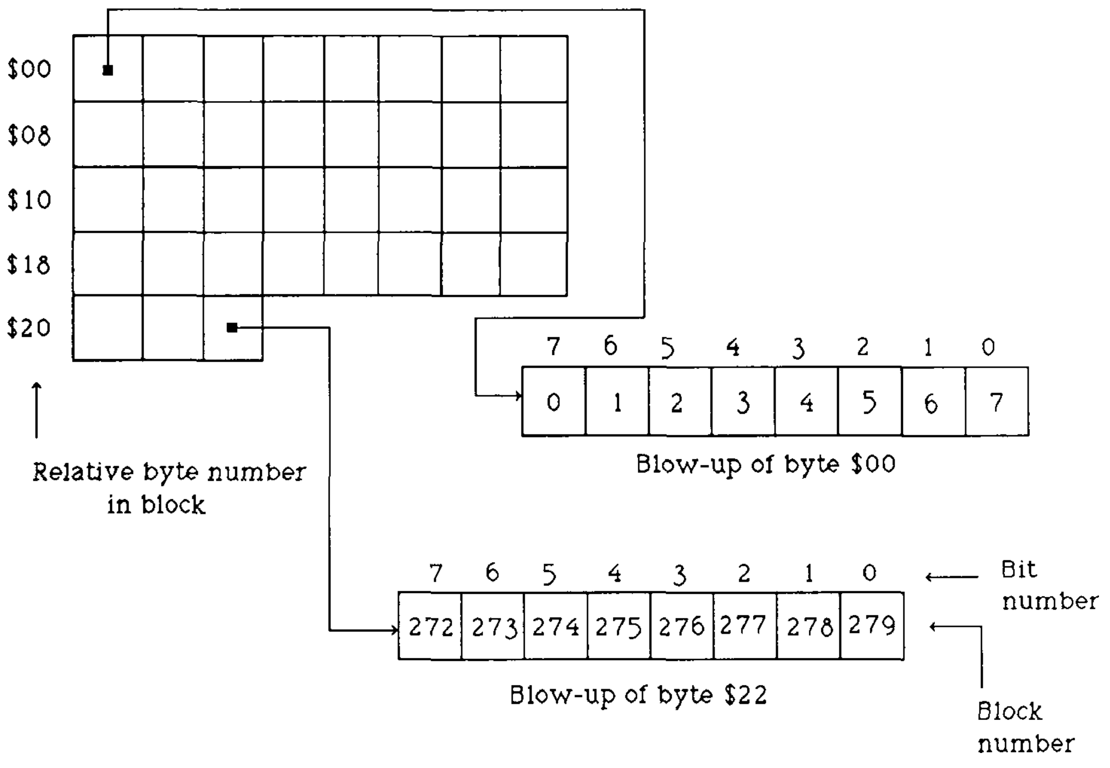


Each block holds 512 bytes.
Total storage capacity is 280 blocks (140K) for a 5.25-inch disk.
Total storage capacity is 1600 blocks (800K) for a 3.5-inch disk.

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. (The operating system determines the size of the volume bit map by examining 2 bytes in the volume directory header that hold the size of the disk; the program used to format the disk places them there. We look at volume directory headers later in this chapter.)

Figure 2-5 shows the structure of the volume bit map for 5.25-inch disks. As you can see, 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

Figure 2-5 The ProDOS volume bit map for a 5.25-inch disk



Each byte in the volume bit map defines the states of eight 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 involved. To get the specific bit number within that byte, subtract the remainder from 7.

lowest-numbered block. If the bit corresponding to a particular block is 0, that block is free. If it is 1, it is being used by a file on the disk.

You can calculate the byte number (from 0 to 34), and the bit number within that byte (from 0 to 7), corresponding to a given block number 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 ProDOS uses to hold important information concerning each file on the disk. This includes the filename, type, size, creation date, 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. Except for the volume directory (the one all the others are accessed through), these directories can occupy just about any area on the disk since ProDOS treats them much like standard files. The volume directory, however, always begins at block 2; if you use a standard disk-formatting program, or the GS/OS Format and EraseDisk commands, it also occupies blocks 3, 4, and 5.

A ProDOS directory is an example of a doubly linked-list data structure. The links are actually pairs of 2-byte pointers 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 directories of any size to be created.

Each block used by a directory can hold up to 13 39-byte file entries. (This means the four-block volume directory used with most ProDOS-formatted disks can hold a total of 52 entries, one of which is an entry for the volume name itself.) Table 2-2 shows the map of a directory block.

The Directory Header

The first block a directory (or subdirectory) uses is the *key block*, and it is configured slightly differently from 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*.

Table 2-3 shows the meanings of each of the 39 bytes making up a directory header. 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 these types of files are virtually identical and are shown in Table 2-4.

File Type Codes

The only way to determine the general nature of the file a particular file entry corresponds to is to examine the file type code at relative position $10 within the entry. Many of the 256 different codes have now been assigned by Apple, and Table 2-5 summarizes their meanings. Table 2-5 also shows the three-character mnemonics often used to represent these file types. All file type codes, except $F1 through $F8, are reserved for operating system use; user programs may freely use the user-defined codes for any purpose.

Table 2-2 Map of a ProDOS file system 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 meaning of the contents of any specific file type actually depends on the program that created the file in the first place. For example, in a BASIC.SYSTEM environment, several file type codes identify files containing specific information useful in an Applesoft environment. Let's look at five of the most common file types used by BASIC.SYSTEM.

TXT (code \$04). A TXT file (Figure 2-6) contains ASCII-encoded text. (Standard ASCII codes, with bit 7 cleared to zero, are used. DOS 3.3 creates textfiles with codes that have bit 7 set to 1.) Each line of text ends with a carriage return code (\$0D), and if it's a standard sequential textfile (one containing consecutive lines of text), the last

Table 2-3 Map of a ProDOS file system directory header

Byte Number in Key Block	*Description*
$04	High 4 bits: storage type code – $F for a volume directory – $E for a subdirectory Low 4 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 this directory was created (format: MMMDDDDD YYYYYYYM, see Figure 8-1)
$1E–$1F	The minute (byte $1E) and hour (byte $1F) this directory entry was created (format: see Figure 8-1)
$20	The version number of ProDOS that created this directory
$21	The lowest version of ProDOS that is capable of using this directory
$22	The access code for this directory (see Figure 2-10)
$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 in which 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
$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)

Table 2-4 Map of a ProDOS file system directory file entry

Relative Byte Number Within Entry	*Meaning of Entry*
$00	High 4 bits: storage type code – $0 for an inactive (or deleted) file – $1 for a seedling file – $2 for a sapling file – $3 for a tree file – $4 for a Pascal area – $5 for an extended file – $D for a subdirectory file Low 4 bits: length of filename
$01–$0F	Filename (in standard ASCII with bit 7 = 0)
$10	File type code (see Table 2-5)
$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 or key 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 this file was created (format: MMMDDDDD YYYYYYYM, see Figure 8-1)
$1A–$1B	The minute (byte $1A) and hour (byte $1B) this file was created (format: see Figure 8-1)
$1C	The version number of ProDOS that created this file
$1D	The lowest version of ProDOS that is capable of using this file
$1E	The access code for this file (see Figure 2-10)
$1F–$20	The auxiliary type code for the file; this code is used for special purposes; 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 for Applesoft program files
$21–$22	The date this file was last modified (format: MMMDDDDD YYYYYYYM, see Figure 8-1)

Table 2-4 Continued

Relative Byte Number Within Entry	*Meaning of Entry*
\$23–\$24	The minute (byte \$23) and hour (byte \$24) 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

Table 2-5 ProDOS file type codes

File Type Code	*Mnemonic*	*Description*
\$00	UNK	Uncategorized file
\$01	+BAD	Bad disk block file
\$02	+PCD	Pascal code file
\$03	+PTX	Pascal textfile
\$04	+*TXT	ASCII textfile
\$05	+PDA	Pascal data file
\$06	*BIN	General binary file
\$07	+FNT	SOS font file
\$08	+FOT	SOS foto file
\$09	+BA3	Business BASIC program file
\$0A	+DA3	Business BASIC data file
\$0B	+WPF	Word processor file
\$0C	+SOS	SOS system file
\$0F	+*DIR	Subdirectory file
\$10	+RPD	RPS data file
\$11	+RPI	RPS index file
\$12	AFD	AppleFile discard file
\$13	AFM	AppleFile model file

Table 2-5 Continued

File Type Code	*Mnemonic*	*Description*
$14	AFR	AppleFile report format file
$15	SCL	Screen library file
$19	+*ADB	AppleWorks database file
$1A	+*AWP	AppleWorks word processing file
$1B	+*ASP	AppleWorks spreadsheet file
$AB	GSB	GS BASIC program file
$AC	TDF	GS BASIC toolbox definition file
$AD	BDF	GS BASIC data file
$B0	+SRC	APW source code file
$B1	+OBJ	APW object code file
$B2	+LIB	APW library file
$B3	+S16	GS/OS system file
$B4	+RTL	APW run-time library file
$B5	+EXE	APW executable code file
$B6	PIF	GS/OS permanent init file
$B7	TIF	GS/OS temporary init file
$B8	+NDA	New desk accessory file
$B9	+CDA	Classic desk accessory file
$BA	+TOL	GS/OS tool set file
$BB	DVR	GS/OS driver file
$BC	GLF	GS/OS generic load file
$BD	FST	GS/OS file system translator
$C0	PNT	Compressed super hi-res picture file
$C1	PIC	Super hi-res picture file
$C8	FON	GS/OS font file
$C9	FND	Finder data file

Table 2-5 Continued

File Type Code	*Mnemonic*	*Description*
$CA	ICN	Finder icon file
$CB	AIF	Audio interchange format file
$EE	R16	EDASM 816 relocatable object file
$EF	*PAS	Pascal area on partitioned disk
$F0	+*CMD	BASIC.SYSTEM command file
$F1–$F8		User-definable files
$F9	O.S	GS/OS operating system
$FA	*INT	Integer BASIC program file
$FB	*IVR	Integer BASIC variables file
$FC	+*BAS	Applesoft BASIC program file
$FD	+*VAR	Applesoft BASIC variables file
$FE	+*REL	EDASM relocatable code file
$FF	+*SYS	ProDOS 8 system file

NOTES:
The mnemonics marked with * are used by the BASIC.SYSTEM CATALOG command.
The mnemonics marked with + or * are used by the Apple Programmer's Workshop (APW) development system.
SOS is the operating system for the Apple III.

byte in the file is 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 in a record is called a *field*. If the number of characters in a record is less than the record size, the rest of the record is padded with $00 bytes; these $00 bytes are *not* end-of-file markers. The record length of a textfile is the auxiliary type code in the directory entry (at relative bytes $1F-$20); if the record length is zero, the file is a sequential textfile.

BAS (code $FC). A BAS file (Figure 2-7) contains an Applesoft program in standard tokenized form. Tokens are 1-byte codes for Applesoft keywords such as PRINT and INPUT. (For a detailed description of this form, see Chapter 4 of *Inside the Apple IIe*.) A BAS file is automatically created when you use the BASIC.SYSTEM SAVE command

Figure 2-6 The structure of a TXT file

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 equal to 0); DOS 3.3 stores text as "negative" ASCII codes (with bit 7 equal to 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 and \$20 in the file's directory entry) is its record length; it is zero for a sequential textfile.

to transfer the image of the Applesoft program from memory to disk. The auxiliary type code for a BAS file is usually \$801, the standard loading address for an Applesoft program.

BIN (code \$06). A BIN file (Figure 2-8) is a general-purpose binary data file that can contain just about anything: programs, data, text, and so on. It is the type of file created by the BASIC.SYSTEM BSAVE command. The exact meaning of the contents of a BIN file cannot be generalized although many of them contain executable code. The auxiliary type code for a BIN file is the address it was BSAVEd to disk from.

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. We describe the characteristics of a standard system program in Chapter 5.

VAR (code \$FD). A VAR file (Figure 2-9) contains a set of Applesoft program variables in a special packed form. It is automatically created when you use the BASIC.SYSTEM STORE command and can be reloaded using the RESTORE command. The first 5 bytes of a VAR file contain the total length of the simple (undimensioned) and array (dimensioned) variable tables created by an Applesoft program (2 bytes), the length of the simple variable space itself (2 bytes), and the HIMEM page

Figure 2-7 The structure of a BAS file

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.

number in effect when the file was saved (1 byte). Following these bytes are 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

Figure 2-8 The structure of a BIN file

This program:

```
         ORG $300
HALFTIME DFB $00
LENGTH   DFB $00

         LDY #255
         LDA LENGTH
         STA $32F
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 this example.

image of the compressed variables was stored. (For a description of the structure of the Applesoft variable tables, see Chapter 4 of *Inside the Apple IIe*.)

File Access Codes

Relative byte $1E within each directory entry is a 1-byte code, 4 bits of which reflect the read (bit 0), write (bit 1), rename (bit 6), and destroy (bit 7) status of the file. If a bit is set to 1, ProDOS allows the operation associated with that bit.

Another bit (bit 2) indicates whether the file is to be considered invisible or not. If the invisible bit is set, cataloging subroutines should ignore the file. Yet another bit (bit 5) indicates whether the file has been modified since the last time it was backed up. (It is the backup program's responsibility to clear this bit to 0 when it makes a copy of the file.) The two remaining bits (bits 3 and 4) are not used and are always 0. Figure 2-10 shows a detailed description of the access code byte.

Figure 2-9 The structure of a VAR file

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                      HIMEM 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.

The BASIC.SYSTEM LOCK and UNLOCK commands also 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 appears to the left of its name in a BASIC.SYSTEM CATALOG listing. The asterisk also appears if only one or two of these three types of access modes is disabled. If the file is just read disabled, the asterisk does not appear, but a "file locked" error message appears if you attempt to read the file with a BASIC.SYSTEM command.

Unfortunately, there is no BASIC.SYSTEM command for setting or clearing individual bits of the file access code so that you can easily attach a particular security level to a file. But as we see in Chapter 4, however, you can do this with the GS/OS or ProDOS 8 SetFileInfo command.

Figure 2-10 Description of the ProDOS access code

7	6	5	4	3	2	1	0
D	RN	B	[Reserved]		I	W	R

D	= destroy-enable bit	I	= invisibility bit
RN	= rename-enable bit	W	= write-enable bit
B	= backup-needed bit	R	= read-enable bit

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

If the D, RN, and W bits are all 1, the file is said to be unlocked; if all three are 0, the file is locked. Any other combination means the file is subject to restricted-access limitations.

The invisibility bit is for the benefit of cataloging subroutines that support the concept of hidden, or invisible, files. If the bit is set, the subroutine should exclude the file from a catalog listing.

ProDOS 8 and GS/OS automatically set the backup-needed bit to 1 whenever they write anything to a file. This makes it possible to develop backup programs that 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 once it has made a copy of the file.

Time and Date Formats

Each ProDOS directory entry contains 8 bytes holding the creation and modification time and date for the file 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 holding the data for any particular nondirectory file on the disk. In the most common implementation of this scheme (the one used for files between 2 and 256 data blocks in length), the key block pointer in the file's directory entry (at relative bytes $11 and $12) points to an index block containing 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, 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 no space on the disk is wasted. The disadvantage is that disk I/O operations take place more slowly than, for example, Apple Pascal because it takes longer to position the disk read/write head over the blocks of a fragmented file.

Indexing Schemes

ProDOS actually uses three variants of this general indexing scheme; the one used depends on the size of the file being dealt with. The following "woodsy" classifications 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 (16Mb - 1) bytes (up to 32,768 data blocks)

You can determine the indexing scheme used by a nondirectory file by examining the storage type code number stored in the high-order 4 bits of the 0th byte 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 identifies a directory entry for a subdirectory file, code $E a subdirectory, and code $F a volume directory. A storage type of $4 identifies a Pascal area on a disk and $5 identifies an extended file.)

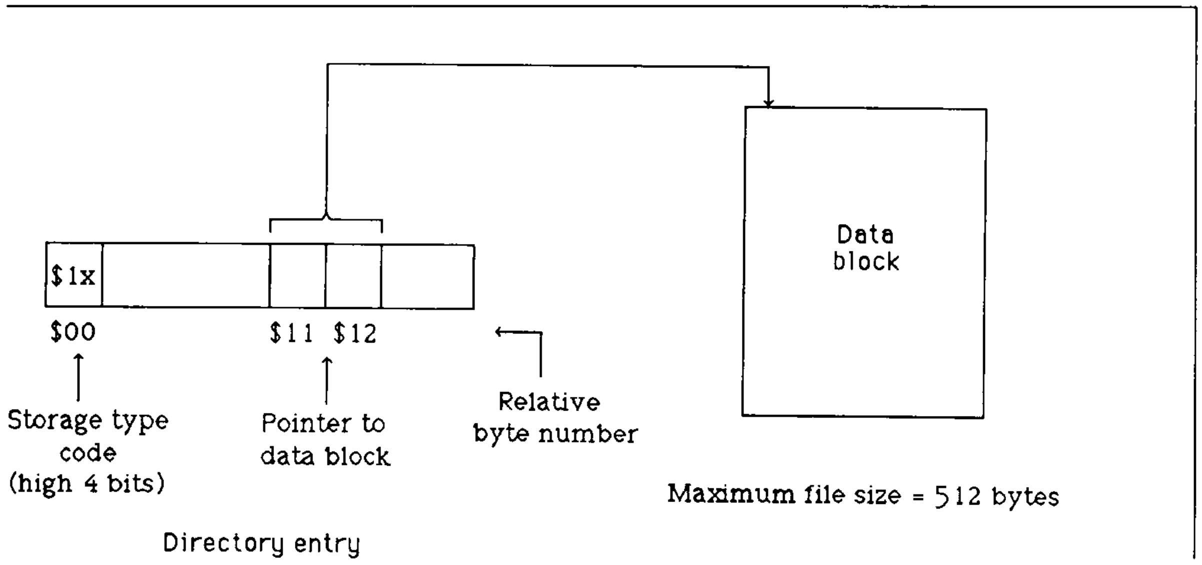
As we have just seen, a file's key pointer (relative bytes $11 and $12 of its directory entry) points to an index block (also called the key block) for the file. Let's look at how ProDOS uses the index block for each of the three types of files.

Seedling File. A seedling file (Figure 2-11) 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 this block is not really an index block at all; it simply holds the contents of the file.

Sapling File. The key pointer of a sapling file (Figure 2-12) holds the block number of a standard index block containing an ordered list of the block numbers used to store that file's data. Table 2-6 shows what an index block for a sapling file looks like. Since block numbers can exceed 255, 2 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 farther into the block. The maximum size of a sapling file is 128K; it cannot be larger than this since an index block can point to only 256 blocks.

Tree File. For a tree file (Figure 2-13), 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-type index blocks. Table 2-7 shows the structure of a master index block. Just as for sapling files, each of the index blocks the master index block points to 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 16Mb (less 1 byte, which is reserved for an end-of-file marker).

Figure 2-11 The structure of a seedling file



ProDOS determines the storage type of an existing file by examining the 4 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.

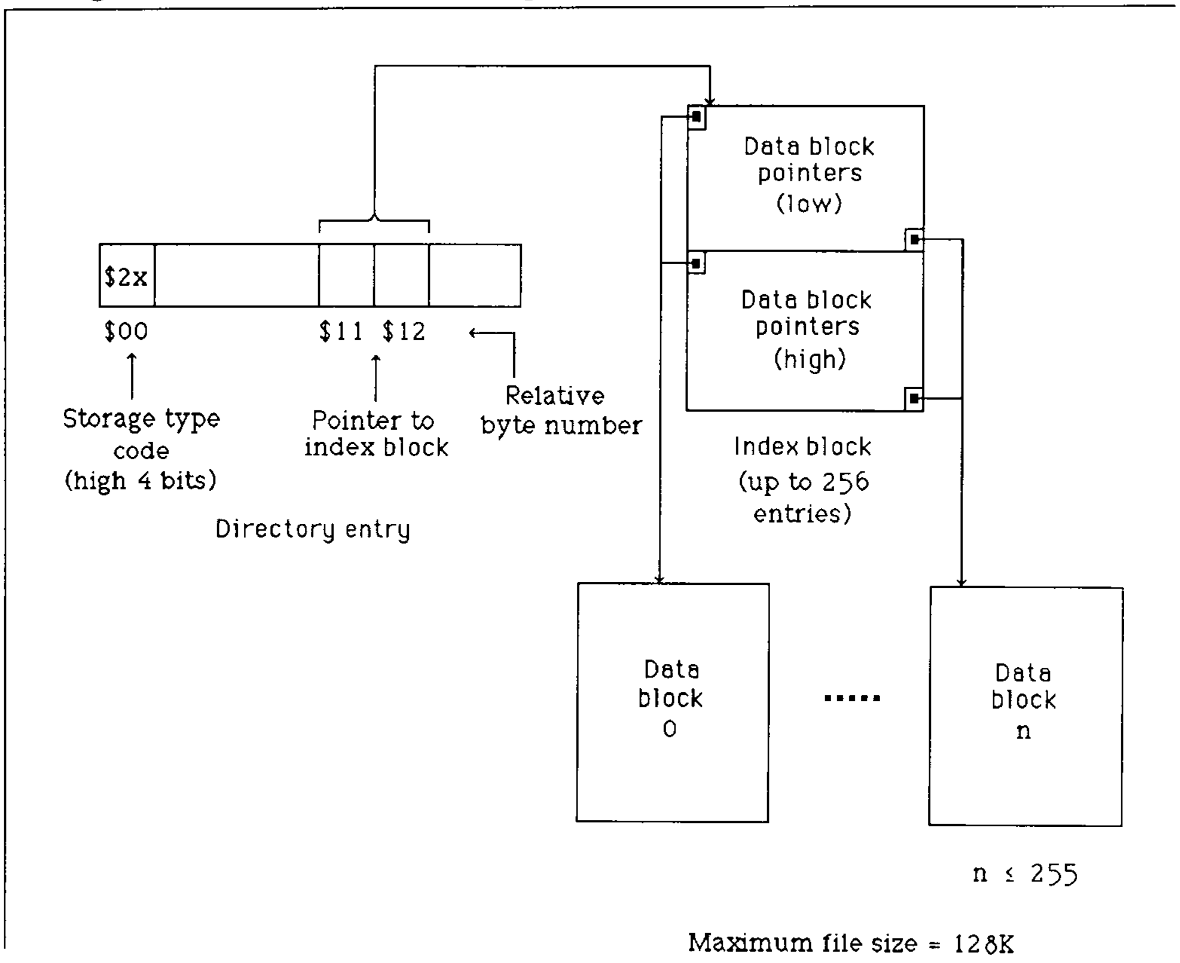
The operating system takes care of all conversions that might become necessary if a file changes its storage type when it changes size. All this happens invisibly, and it is generally not necessary for an application to know the storage type unless it is not using standard operating system commands to access files.

ProDOS uses these three different indexing structures to minimize the disk space needed to manage a file. This permits the operating system to access a file as quickly as possible and frees up disk space for use by other files.

Extended Files

GS/OS (but not ProDOS 8) can also create *extended files* on a ProDOS-formatted disk. These files have a storage type code of $5. An extended file contains two logical data segments, the *data fork* and the *resource fork*. The data fork generally contains application-specific data, and the resource fork generally contains data organized as a series of well-defined data structures; these data structures define such elements as menu definitions, dialog box templates, and cursor definitions. Apple defines the data structures for everyone to use.

Figure 2-12 The structure of a sapling file



The key block for an extended file is really not a key block at all—it's just an extension of the file's directory entry. The first half of the block contains information related to the data fork; the second half contains information related to the resource fork.

GS/OS uses only the first 8 bytes in each half block. The meaning of each of these bytes is as follows:

$00	storage type code for the fork
$01-$02	actual key block number for the fork
$03-$04	size of the fork (in blocks)
$05-$07	size of the fork (in bytes)

Table 2-6 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)

The storage type code for the fork is either $01 (seedling), $02 (sapling), or $03 (tree). The key block for a fork of an extended file (stored at offsets $01–$02) is arranged just like the key block for a regular file of the same size as the fork.

Sparse Files

As we saw in the discussion of TXT files, it is possible to create and use files that are not sequential. That is, you can write information to *any* position within a file even if that position is far away from any other previously used part of the file. To save disk space, ProDOS does not actually allocate 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. ProDOS stores an actual block number in this entry at the time that part of the file is actually written to.

Such a file is called a *sparse* file because it does not take up as much space on disk as its file size indicates it should.

Figure 2-13 The structure of a tree file

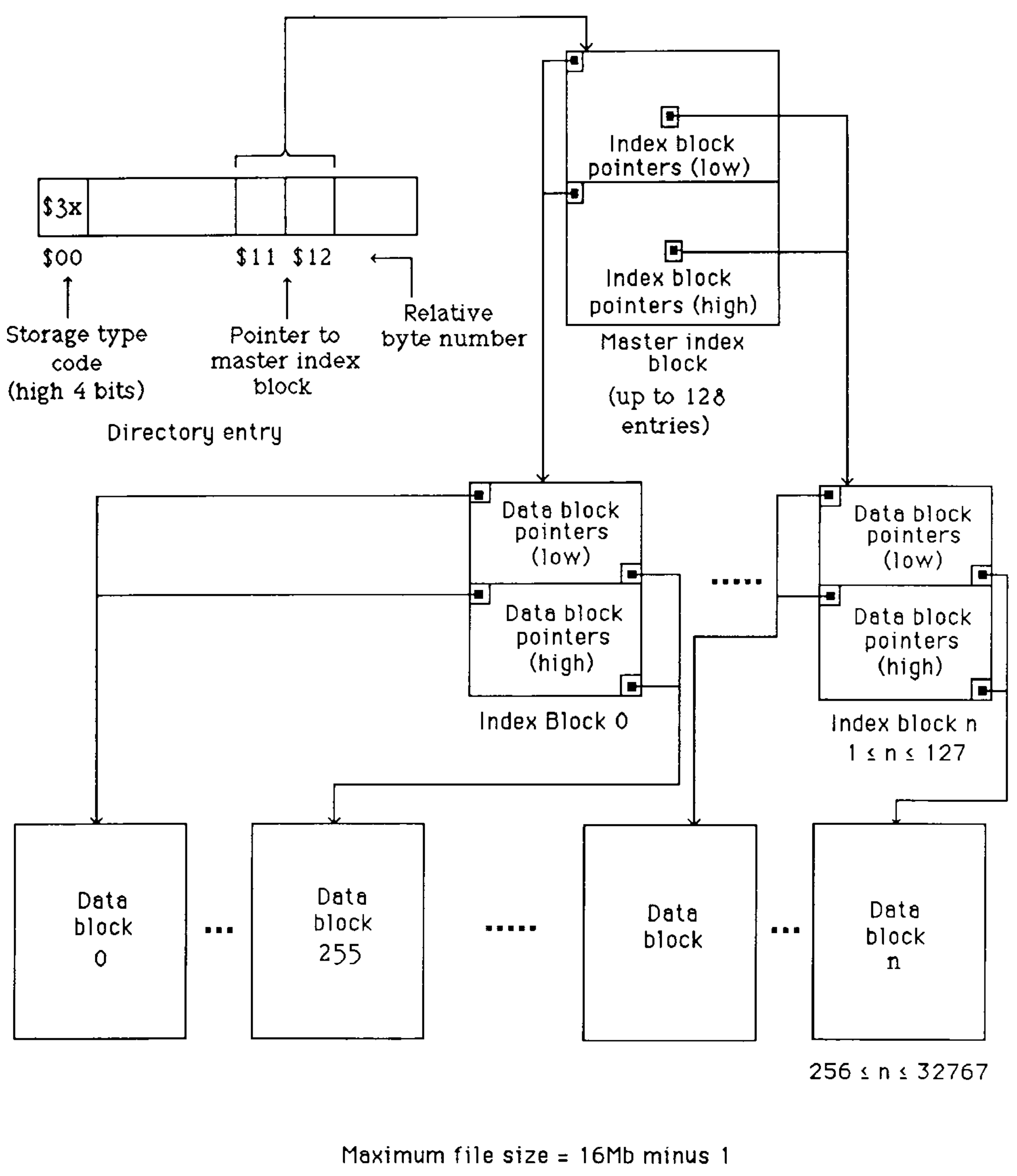

$3x
$00
$11 $12
Storage type code (high 4 bits)
Pointer to master index block
Relative byte number
Directory entry
Index block pointers (low)
Index block pointers (high)
Master index block
(up to 128 entries)
Data block pointers (low)
Data block pointers (high)
Index Block 0
Index block n
1 ≤ n ≤ 127
Data block 0
Data block 255
Data block
Data block n
256 ≤ n ≤ 32767
Maximum file size = 16Mb minus 1

Table 2-7 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)

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. Figure 2-14 shows the structure of such a file. Record 2 is stored beginning at position $100 (2 × 128) in the file; this corresponds to position $100 of the first block allocated to the file (index block entry 0). Record 64 begins at position $2000 (128 × 64) in the file; this corresponds to position $000 of the 16th index block entry. The 15 unused blocks between these two records appear as $0000 entries in the index block. Thus even though the file is logically 17 blocks long, ProDOS needs only 3 data blocks to store it on the disk (1 for the index block and 2 for the data blocks).

THE READ.BLOCK PROGRAM

Table 2-8 shows a useful Applesoft program called READ.BLOCK. You can use it to examine any of the blocks of data on a disk formatted for the ProDOS file system, 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. But you should be careful when writing a

Figure 2-14 The structure of a sparse file

The file created by this program:

```
10 F$ = "RANDOM"
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 that data
0008: 00 00 00 00 00 00 00 00     blocks 0 and 16 are stored
0010: 8E 00 00 00 00 00 00 00     at blocks $008C and $008E
  .                               on this disk.)
  .
0100: 00 00 00 00 00 00 00 00
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
```

Table 2-8 READ.BLOCK, a program to read any block on a ProDOS-formatted disk

```
1  REM "READ.BLOCK"
2  REM COPYRIGHT 1985-1987 GARY B. LITTLE
3  REM DECEMBER 6, 1987
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( 6);
     "COPR. 1985-1987 GARY B. 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
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$ = " "
1065  IF  ASC (A$) >  = 96 THEN A$ =  CHR$ ( ASC (A$) - 32)
1070  PRINT A$
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
```

Table 2-8 Continued

```
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, FN M2(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$: GOTO 1060
1210  GOTO 5000
1220 BL = BL - 1: IF BL < 0 THEN BL = 0
1230  GOTO 190
1240 BL = BL + 1: GOTO 190
1260  TEXT : HOME : END
1270 V = 8:H = 3: VTAB 5: PRINT TAB( 6);: INVERSE :
     PRINT "I=UP M=DOWN J=LEFT K=RIGHT": NORMAL
1280  HTAB 1: VTAB 23: CALL - 958: PRINT TAB( 6);
      "PRESS ";: INVERSE : PRINT "ESC";: NORMAL :
      PRINT " TO LEAVE EDITOR"
1290  REM
1300  GOSUB 1500: GET A$: IF ASC (A$) >  = 96 THEN
      A$ =  CHR$ ( ASC (A$) - 32)
1310 LC = 16384 + 128 * (Q - 1) + 8 * V + H:Y =  PEEK (LC):
     X = ASC (A$)
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$): IF (X >  = 48 AND X <  = 57) OR
     (X >  = 65 AND X <  = 70) THEN  PRINT A$;:
```

Table 2-8 Continued

```
     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
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 169,64,133,7,162,0,160,0,56,165,7,233,64,32,218,253,
     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
```

block to the disk because you may accidentally render the disk unreadable; you should always perform writing experiments with a backup copy of the original disk.

When you first start up READ.BLOCK, you must 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. The program then reads the base block into memory and displays it on the screen in a special format. Because of 40-column screen size limitations, only one quarter of the block appears at once. (You must press the D key to display the other three quarters.)

The contents of a block appear in 64 rows, each of which contains an offset address from the beginning of the block followed by the hexadecimal representations of the 8 bytes stored from that location onward. At the far right of each row are the ASCII representations of each of these 8 bytes. The program displays only 16 rows on the screen at once.

After the program displays the entire block, it asks you to enter one of nine commands:

B	reset the base block number
C	copy the contents of the block to the printer (which 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 you enter the Edit command, the cursor moves to the center of the 8-by-16 array of hexadecimal digits representing the contents of one quarter of the block. To change any entry, use the I, J, K, and M keys to move the cursor up, left, right, and down, respectively, and then type in the new two-digit hexadecimal entry for that position. You can leave editing mode at any time by pressing the Esc key. Once you leave editing mode, you can save the changes to disk using the W (Write) command.

CHAPTER 3

Loading and Installing GS/OS and ProDOS 8

In this chapter, we investigate exactly what happens when GS/OS and ProDOS 8 load into memory from disk, what areas of memory they occupy, and how applications can make use of the areas of memory they don't occupy. This information is important if you're trying to build a bootable distribution disk for your own application or if you want to understand how to develop an application that doesn't interfere with system resources.

For ProDOS 8, we also examine the ProDOS 8 system global page, a 256-byte area of memory residing from $BF00 to $BFFF in main memory. A good understanding of the global page is absolutely necessary if you want to write programs that communicate properly with ProDOS 8 or if you want to install custom drivers for disks and clocks.

THE BOOT RECORD

The first two blocks (numbered 0 and 1) of every standard ProDOS-formatted disk contain an assembly-language program, called the *boot record*, which is placed on the disk when you format the disk. When you boot a disk, the ROM on the disk controller card loads the boot record program into memory at location $0800 in main memory and then executes it by calling its entry point at $0801.

The boot record program can load ProDOS 8 on an Apple II or GS/OS (or ProDOS 16) on an Apple IIGS.

When the boot record program starts executing, it loads the volume directory blocks into the memory area beginning at address $0C00. (It assumes the first volume directory block is block 2.) It then scans the directory entries looking for a system file called PRODOS. If it isn't there, it displays the message:

```
UNABLE TO LOAD PRODOS
```

and the system halts. A bootable disk must contain the PRODOS file; use a file-copying utility to transfer a copy from a ProDOS 8 or GS/OS master disk to the ProDOS-formatted disk you wish to boot from.

> *Note*: Keep in mind that there are three distinct versions of the program called PRODOS. The ProDOS 8 version contains a copy of the ProDOS 8 operating system and the necessary installation code. The ProDOS 16 version contains startup code and the code defining the IIGS System Loader. Finally, the GS/OS version contains startup code and three file-system specific subroutines that the operating system loader and program dispatcher can use to load a file from disk, determine the name of the boot volume, and determine the name of the file system translator associated with the PRODOS file. The equivalent of the ProDOS 8 version of PRODOS is stored in a file called P8 in the SYSTEM/ subdirectory of a GS/OS system disk. You can use P8 to create a bootable ProDOS 8 disk by copying it to the volume directory of a freshly formatted disk and renaming it as PRODOS.

If the PRODOS program file exists, the boot record loads it into memory beginning at location $2000 and runs it by executing a JMP $2000 instruction. What happens next depends on whether you're booting a GS/OS or ProDOS 8 system disk. In the next section, we analyze a ProDOS 8 boot sequence; at the end of the chapter, we do the same for GS/OS.

THE PRODOS 8 BOOT

The ProDOS 8 version of the PRODOS file contains a copy of the code for the ProDOS 8 operating system itself as well as the code necessary to initialize various system parameters (number of disk drives, amount of system memory, and so on) stored in a special data area called the *ProDOS 8 system global page*. When PRODOS gets control, one of the first things it does is relocate the ProDOS 8 image to its execution position in bank-switched RAM. (We describe this RAM area in detail in the next section.)

On version 1.3 or higher of ProDOS 8, PRODOS next looks in the volume directory for a file called ATINIT with a file type code of $E2. If it finds the file, PRODOS loads and executes it. At present, the ATINIT file begins with an RTS instruction, so nothing of interest happens when PRODOS calls it. ATINIT is merely a data file for AppleTalk Networking System utility programs.

If PRODOS finds the ATINIT file, but its file type code is not $E2, or PRODOS can't load it, PRODOS displays the message

```
** UNABLE TO LOAD ATINIT FILE **
```

and the system hangs. If no ATINIT file is present, PRODOS simply goes on to the next step in the boot sequence.

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

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

and the booting procedure stops. Every bootable disk must contain a system file whose name ends in .SYSTEM and it must be in the volume directory.

If PRODOS does find a system file with the .SYSTEM suffix, it loads it into memory beginning at $2000 and executes it with a JMP $2000 instruction. This ends the booting procedure.

To boot into an Applesoft programming environment, the system file must be BASIC.SYSTEM. (It is found on the ProDOS 8 master disk.) As we 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 ProDOS 8 when required.

It should be clear from this discussion that ProDOS 8 is really nothing without a system program like BASIC.SYSTEM to act as a software interface between the user and the low-level ProDOS 8 operating system. It just won't operate without such a program. For this reason, the ProDOS 8–BASIC.SYSTEM environment is commonly referred to as ProDOS 8 even though this is technically not so. Later in this chapter, we examine ProDOS 8 proper; we defer a detailed discussion of BASIC.SYSTEM (and system programs in general) to Chapter 5.

PRODOS 8 MEMORY USAGE

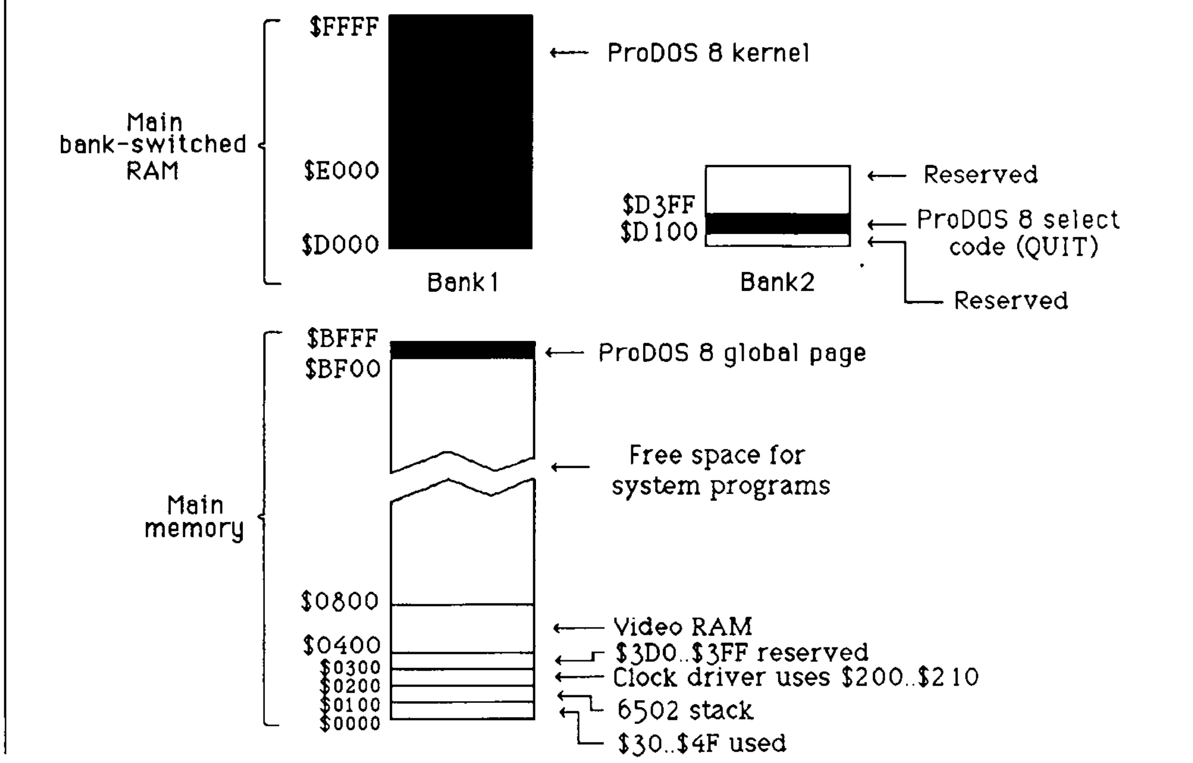
Bank-Switched RAM

After ProDOS 8 has been loaded into memory, it occupies 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 8 system global page.*)

The remaining space in bank-switched RAM ($D400–$DFFF in $Dx bank2) is reserved for future use by ProDOS 8 and must not be used by application programs.

Figure 3-1 ProDOS 8 memory map



You may 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 zero (or an Apple IIe, IIc, or IIGS) has 64K of main RAM memory that is normally used by Applesoft and ProDOS 8. But this memory 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 $0000–$BFFF, but the remaining 16K of memory, the bank-switched RAM, corresponds to one 8K region of memory, $E000–$FFFF, and two 4K regions of memory, $D000–$DFFF (called $Dx bank1 and $Dx bank2, respectively). The address space used by bank-switched RAM is 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 given time.

As Table 3-1 shows, 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 $Dx bank2 is to be used. You can even set these switches so that the RAM area can be read from but not written to or so that it will be active for write operations while the corresponding ROM area is active for read

Table 3-1 Bank-switched RAM soft switches[a]

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

NOTES:
[a]Read a location to perform the indicated function.
[b]Read twice in succession to write-enable bank-switched RAM.

operations. This means you can write data to the RAM area while running a program that uses subroutines in the ROM area (that is, subroutines in Applesoft and the system Monitor program).

To activate the desired mode of operation, you must 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 8 takes care of managing the bank-switched RAM switches whenever you ask it to perform some command. In general, it saves the state of bank-switched RAM when it gets control and then read- and write-enables bank1 of bank-switched RAM before passing control to a subroutine residing there. When it relinquishes control, it restores bank-switched RAM to its original state. Bank1 is active when control passes to a user-installed interrupt handler, disk driver, or clock driver.

Auxiliary Memory

An Apple IIe, with an extended 80-column text card installed, an Apple IIGS, and an Apple IIc all have a 64K *auxiliary* memory space that is mapped to addresses in the same way that the main 64K memory space is. Since most Applesoft programs don't use this space, ProDOS 8 uses it for storing files in the same way it uses a real disk drive for storing files. The name of the volume for this so-called RAMdisk is /RAM; we investigate its characteristics in Chapter 7.

Page Zero Usage

ProDOS 8 uses 22 locations in page zero (of both main and auxiliary memory) for temporary data storage: $3A–$4F. The first 6 locations ($3A–$3F) are used only by the internal ProDOS 8 disk device drivers for 5.25-inch drives and the /RAM volume. This means if ProDOS 8 performs a disk I/O operation, the existing contents of $3A–$3F are overwritten. This is not too serious since these locations are usually used by the Apple II's system Monitor command interpreter only. But if an application program uses them, an irreconcilable conflict will occur, and the program could bomb. Don't use them.

The other 16 locations ($40–$4F) are used by the ProDOS 8 machine language interface (MLI) subroutine. But unlike for the $3A–$3F area, when control passes to the MLI, the current contents of $40–$4F are saved in a safe data area within ProDOS 8 and are restored just before control returns to the caller.

Page Two Usage

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

The standard internal ProDOS 8 clock driver works with clock cards that use the command set first popularized by the Thunderclock. One of the quirks of this command set is that it requires use of the first part of the Apple II's line input buffer ($0200–$0210) to store the time data string whenever ProDOS 8 requests the time. This means an application program must not use this area for any purpose; if it does, it will probably not work properly after ProDOS 8 calls the clock driver.

Other parts of page two may well be used by the system program used with ProDOS 8. BASIC.SYSTEM, for example, uses most of page two as a temporary data buffer area when it executes 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 end of page three of memory ($3D0–$3FF) is used for special purposes on the Apple II. First, ProDOS 8 reserves the $3D0–$3EC area for use by any system program (like BASIC.SYSTEM) that may be active. 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 3-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 investigate BASIC.SYSTEM's use of page three in more detail in Chapter 5.

The rest of page three beyond $3D0 is reserved for storing a set of user-installable vectors and subroutines that service interrupt conditions or provide special commands:

- XFER vector at $3ED–$3EE—IIe, IIc, and IIGS only (This vector facilitates the transfer of data between main and auxiliary memory.)
- 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 (nonmaskable interrupt) vector at $3FB–$3FD
- IRQ (interrupt request) vector at $3FE–$3FF

(See Appendix IV of *Inside the Apple IIe* for a detailed discussion of the meaning of each of these vectors and subroutines.)

ProDOS 8 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. However, BASIC.SYSTEM stores other values in these vectors (except BRK and IRQ) when it first loads. (See Chapter 5 for a description of how BASIC.SYSTEM initializes these vectors.)

THE PRODOS 8 SYSTEM GLOBAL PAGE: $BF00-$BFFF

The page of memory from $BF00 to $BFFF is called the *ProDOS 8 system global page*, and it acts as the gateway to ProDOS 8 proper (that is, the part that resides in bank-switched RAM). It contains several fixed-position jump vectors to standard ProDOS 8 subroutines (the machine language interface, clock driver, 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 8.

The global page also contains the bank-switching subroutines needed to transfer control to and from the parts of the ProDOS 8 machine language interface and interrupt handler that reside in bank-switched RAM. Since you should never need to use these subroutines directly, their addresses, and the code itself, are not guaranteed to stay the same from one ProDOS 8 version to another.

The System Bit Map

One important area in the ProDOS 8 global page is the system bit map; it occupies the area from $BF58 to $BF6F. This map indicates which RAM areas have been reserved and which are free for a file to use. Before ProDOS 8 performs any loading or buffer allocation operations, it examines this map to see if there will be a conflict with a reserved area. If there will be, it does not execute the command, and it reports an error condition.

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 $BF). If a bit is set to 1, the corresponding page has been reserved. The relative byte number (counting from zero) within the system bit map in which the bit for a given page number resides is the whole number calculated by dividing the page number by 8; the bit number within this byte is 7 minus the remainder generated by the division. For example, the bit for page 190 ($BE) is bit 1 of relative byte 23: 190 divided by 8 is 23 (the relative byte number), and the remainder is 6 (meaning the bit number is 7 − 1 = 6).

ProDOS 8 initially marks page zero, the stack page (page 1), the video RAM area (pages 4–7) and its global page (page $BF) as reserved. Other pages can be protected as desired by system and application 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 first part of page 3 ($300–$3CF) because that area is not otherwise used by Applesoft, ProDOS 8, 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 (bit 4 of $BF58).

The Machine Identification Byte

There is a byte in the ProDOS 8 global page called MACHID ($BF98) you can examine to determine the nature of the hardware environment ProDOS 8 is executing in. It contains information on the type of Apple being used (II, II Plus, IIe, IIc, or IIGS), the amount of RAM memory (48K, 64K, or 128K), and whether an 80-column card or clock card is in the system.

The bits in MACHID have the following meanings:

bits 7,6 (if bit 3 = 0)	00 = Apple II
	01 = Apple II Plus
	10 = Apple IIe or IIGS
	11 = Apple III in Apple II emulation mode
bits 7,6 (if bit 3 = 1)	00 = [reserved]
	01 = [reserved]
	10 = Apple IIc
	11 = [reserved]
bits 5,4	00 = [reserved]
	01 = 48K RAM
	10 = 64K system
	11 = 128K system (IIe, IIc, IIGS)
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

It is not possible for an application to determine the exact type of Apple II it's running on by examining the MACHID byte. For a precise identification, an application should instead inspect two identification bytes stored in the Monitor ROM at $FBB3 and $FBC0. An Apple II Plus has $EA stored at $FBB3, whereas the IIe, IIc, and IIGS have $06 stored there. Examine the second location, $FBC0, for a more precise identification: It holds $00 for a IIc, $EA for a IIe, and $E0 for a IIGS or enhanced IIe (the one with the MouseText ROM). To distinguish between the IIGS and enhanced IIe, set the carry flag (with SEC) and call the subroutine at $FE1F in the Monitor. If the carry flag comes back cleared, the system is a IIGS, and the X register contains the ROM version number; otherwise, the system is an enhanced IIe.

Source Listing of the ProDOS 8 Global Page

In later chapters, we analyze in detail all the other important areas in the ProDOS 8 global page. Table 3-2 shows a commented source listing of the code for the global page.

GS/OS SYSTEM DISKS

Certain files must be present on a GS/OS system disk before you can boot it or use it to run both GS/OS and ProDOS 8 applications. The structure of the simplest such system disk is as follows:

```
PRODOS            Operating system startup code
SYSTEM/           Subdirectory: operating system files
  START.GS.OS     GS/OS loader and dispatcher
  GS.OS           GS/OS operating system and System Loader
  ERROR.MSG       GS/OS error messages
START             The startup program
P8                ProDOS 8 operating system
TOOLS/            Subdirectory: RAM-based tool sets
FONTS/            Subdirectory: font files
DESK.ACCS/        Subdirectory: desk accessories
LIBS/             Subdirectory: system library files
DRIVERS/          Subdirectory: device drivers
  APPLEDISK3.5    Driver for 3.5-inch disk drive
  CONSOLE.DRIVER  Console Driver
  SCSI.DRIVER     Driver for SCSI hard disk
SYSTEM.SETUP/     Subdirectory: initialization programs
  TOOL.SETUP      Tool set patching program
  TS2             Patches to ROM version 01 tool sets
FSTS/             Subdirectory: file system translators
  PRO.FST         ProDOS file system translator
  CHAR.FST        Character I/O file system translator
```

In the next section, we see what these files and subdirectories contain.

Table 3-2 Source listing for ProDOS 8 system global page

```
               *************************************
               *     ProDOS 8 System Global Page   *
               *      for ProDOS 8 version 1.7     *
               *                                   *
               * Comments copyright 1985-1988      *
               * Gary B. Little                    *
               *                                   *
               * Last modified: August 28, 1988    *
               *                                   *
               *************************************

               * NOTE: The addresses of the following subroutines
               *       may change in future versions of ProDOS 8.

               CLOCKDR  EQU   $D742      ;ProDOS 8 clock driver
               NODEVICE EQU   $DEAC      ;NO DEVICE CONNECTED vector
               SYSERR1  EQU   $DFFF      ;System error handler
               SYDEATH1 EQU   $E009      ;Critical error handler
               ENTRYMLI EQU   $DE00      ;ProDOS 8 MLI handler
               IRQRECEV EQU   $DF4E      ;ProDOS 8 interrupt handler
               MLIQUIT  EQU   $FCD5      ;QUIT subroutine
               FIX45    EQU   $FFD8

                        ORG   $BF00

               * MLI is the primary entry point to the ProDOS 8
               * machine language interpeter. This interpreter
               * supports a number of commands that can be used
               * to access files. (See Chapter 4.)

BF00: 4C 4B BF MLI      JMP   MLIENT1    ;The gateway to MLI commands

               * QUIT is called whenever the MLI QUIT command is
               * requested. (This is normally done when transferring
               * control from one system program to another.)
               * The standard subroutine asks the user to
               * enter a new prefix and system filename and
               * then executes the program specified. (See Chapter 4.)

BF03: 4C D5 FC QUIT     JMP   MLIQUIT    ;Execute QUIT command

               * DATETIME is called whenever the MLI GET_TIME
               * command is executed. If a clock card is installed,
               * it will read the clock and place the date and
               * time in DATE ($BF90) and TIME ($BF92). See
               * Chapter 8 for details on how this is done.

BF06: 4C 42 D7 DATETIME JMP   CLOCKDR    ;RTS ($60) if no clock

```

Table 3-2 Continued

```
            * ProDOS 8 calls SYSERR if an error occurs during
            * an MLI call. SYSERR takes the error code (that
            * is in the accumulator) and stores it in SERR.
            *
            * SYSDEATH is called whenever a critical error
            * occurs (for example: when important ProDOS 8 data
            * areas are overwritten). The system will have to be
            * restarted if a critical error occurs.
            *
            * (See Chapter 4 for a discussion of MLI system
            * errors and critical errors.)

BF09: 4C FF DF  SYSERR   JMP   SYSERR1     ;System error handler
BF0C: 4C 09 E0  SYSDEATH JMP   SYDEATH1    ;Critical error handler
BF0F: 00        SERR     DFB   $00         ;MLI error code (0 if no error)

            * Disk driver vector table. Each entry in the table
            * corresponds to a unique drive and slot combination
            * as shown. If an entry is unused, its vector
            * points to the ProDOS 8 "no device connected"
            * subroutine. The /RAM device available on an
            * Apple IIe, IIc, or IIGS is mapped to the slot 3,
            * drive 2 device. The entries in the following
            * table are for an Apple IIGS with an Apple 3.5
            * Drive in slot 5, a ProFile hard disk in slot
            * 6, and an Apple II Memory Expansion card in
            * slot 7.

BF10: AC DE     DEVADR01 DA    NODEVICE    ;"No device connected" vector
BF12: AC DE     DEVADR11 DA    NODEVICE    ;Slot 1, drive 1 vector
BF14: AC DE     DEVADR21 DA    NODEVICE    ;Slot 2, drive 1 vector
BF16: AC DE     DEVADR31 DA    NODEVICE    ;Slot 3, drive 1 vector
BF18: AC DE     DEVADR41 DA    NODEVICE    ;Slot 4, drive 1 vector
BF1A: 0A C5     DEVADR51 DA    $C50A       ;Slot 5, drive 1 vector
BF1C: EA C6     DEVADR61 DA    $C6EA       ;Slot 6, drive 1 vector
BF1E: 4E C7     DEVADR71 DA    $C74E       ;Slot 7, drive 1 vector

BF20: AC DE     DEVADR02 DA    NODEVICE    ;"No device connected" vector
BF22: AC DE     DEVADR12 DA    NODEVICE    ;Slot 1, drive 2 vector
BF24: AC DE     DEVADR22 DA    NODEVICE    ;Slot 2, drive 2 vector
BF26: 00 FF     DEVADR32 DA    $FF00       ;Slot 3, drive 2 vector
BF28: AC DE     DEVADR42 DA    NODEVICE    ;Slot 4, drive 2 vector
BF2A: AC DE     DEVADR52 DA    NODEVICE    ;Slot 5, drive 2 vector
BF2C: AC DE     DEVADR62 DA    NODEVICE    ;Slot 6, drive 2 vector
BF2E: AC DE     DEVADR72 DA    NODEVICE    ;Slot 7, drive 2 vector

            * DEVNUM contains the slot and drive code for the
            * last disk device that was accessed. The bit
            * format for this code is as follows:
```

Table 3-2 Continued

```
                 *
                 *    D S S S 0 0 0 0
                 *
                 * where D is the drive number (0 for drive 1
                 * and 1 for drive 2), and SSS is the slot
                 * number (from 1 to 7).

BF30: 60         DEVNUM   DFB   $60        ;Slot, drive of last access

                 * DEVCNT holds the number of active disk devices
                 * installed in the system, less 1.

BF31: 03         DEVCNT   DFB   $03

                 * DEVLST contains a list of the drive and slot
                 * codes for each of the active disk devices
                 * (14 maximum).
                 *
                 * The codes are in the same format as used
                 * for DEVNUM except that the low-order 4
                 * bits contain device characteristics
                 * information. (See Chapter 7.)

BF32: BF         DEVLST   DFB   $BF        ;/RAM in slot 3, drive 2
BF33: 5B                  DFB   $5B        ;3.5" drive in slot 5, drive 1
BF34: 64                  DFB   $64        ;ProFile in slot 6, drive 1
BF35: 74                  DFB   $74        ;RAM card in slot 7, drive 1
BF36: 00                  DFB   $00
BF37: 00                  DFB   $00
BF38: 00                  DFB   $00
BF39: 00                  DFB   $00
BF3A: 00                  DFB   $00
BF3B: 00                  DFB   $00
BF3C: 00                  DFB   $00
BF3D: 00                  DFB   $00
BF3E: 00                  DFB   $00
BF3F: 00                  DFB   $00

BF40: 28 43 29            ASC   +(C)APPLE'83+ ;(+ delimiter only)
BF43: 41 50 50 4C 45 27 38 33

                 * The standard JSR $BF00 MLI call is
                 * routed to this secondary entry point.

BF4B: 08         MLIENT1  PHP
BF4C: 78                  SEI
BF4D: 4C B7 BF            JMP MLICONT

                 * This tiny bit of code is used by the ProDOS 8
```

Table 3-2 Continued

```
                * interrupt-handling subroutine.

BF50: 8D 8B C0           STA   $C08B      ;Turn on RAMcard
BF53: 4C D8 FF           JMP   FIX45
BF56: 00        SAVE45   DFB   $00        ;Contents of $45 upon interrupt
BF57: 00        SAVEDX   DFB   $00        ;ID code for $Dx bank

                * The system bit map. Each bit in this 24-byte
                * (192-bit) table corresponds to a unique page
                * from $00 to $BF. Page $00 corresponds to
                * bit 7 of the first byte, and page $BF
                * corresponds to bit 0 of the last byte.
                * If the page is in use, the corresponding
                * bit will be set to 1. The configuration of
                * the bit map after BASIC.SYSTEM has been
                * loaded is shown.

BF58: CF 00 00  BITMAP   DFB   $CF,$00,$00 ;Pages 0,1,4-7 in use
BF5B: 00 00 00           DFB   $00,$00,$00
BF5E: 00 00 00           DFB   $00,$00,$00
BF61: 00 00 00           DFB   $00,$00,$00
BF64: 00 00 00           DFB   $00,$00,$00
BF67: 00 00 00           DFB   $00,$00,$00
BF6A: 00 3F FF           DFB   $00,$3F,$FF ;Pages $9A-$A7 in use
BF6D: FF FF C3           DFB   $FF,$FF,$C3 ;Pages $A8-$B9,$BE,$BF in use

                * File buffer table. The buffer addresses for each
                * open file (a maximum of 8 are allowed) are stored
                * in this table. A buffer address must be changed
                * by using the MLI SET_BUF command.

BF70: 00 00     BUFFER1  DA    $0000       ;Buffer address for file 1
BF72: 00 00     BUFFER2  DA    $0000       ;Buffer address for file 2
BF74: 00 00     BUFFER3  DA    $0000       ;Buffer address for file 3
BF76: 00 00     BUFFER4  DA    $0000       ;Buffer address for file 4
BF78: 00 00     BUFFER5  DA    $0000       ;Buffer address for file 5
BF7A: 00 00     BUFFER6  DA    $0000       ;Buffer address for file 6
BF7C: 00 00     BUFFER7  DA    $0000       ;Buffer address for file 7
BF7E: 00 00     BUFFER8  DA    $0000       ;Buffer address for file 8

                * Interrupt vector table. This is where the
                * addresses of the user-installed interrupt-
                * handling subroutines are stored (4 maximum).
                * They are installed using the MLI ALLOC_INTERRUPT
                * command and removed using the DEALLOC_INTERRUPT
                * command. Following the vector table is the
                * data area that ProDOS 8 uses to store registers
                * and bank-switching information when an
                * interrupt occurs. (See Chapter 6.)
```

Table 3-2 Continued

```

BF80: 00 00     INTRUPT1 DA    $0000        ;Interrupt vector 1
BF82: 00 00     INTRUPT2 DA    $0000        ;Interrupt vector 2
BF84: 00 00     INTRUPT3 DA    $0000        ;Interrupt vector 3
BF86: 00 00     INTRUPT4 DA    $0000        ;Interrupt vector 4
BF88: 00        INTAREG  DS    1            ;Accumulator
BF89: 00        INTXREG  DS    1            ;X register
BF8A: 00        INTYREG  DS    1            ;Y register
BF8B: 00        INTSREG  DS    1            ;Stack pointer register
BF8C: 00        INTPREG  DS    1            ;Processor status register
BF8D: 00        INTBNKID DS    1            ;ID code for $Dx bank
BF8E: 00 00     INTADDR  DA    $0000        ;Address where IRQ occurred

                * The system date and time are stored in the
                * following two words in a special packed
                * format:
                *
                *      DATE:    year    = bits 15-9 (0..99)
                *               month   = bits 8-5 (1..12)
                *               day     = bits 4-0 (1..31)
                *
                *      TIME:    hours   = bits 12-8 (0..23)
                *               minutes = bits 5-0 (0..59)
                *
                * (See Chapter 8 for more on DATE and TIME.)

BF90: 00 00     DATE     DW    $0000
BF92: 00 00     TIME     DW    $0000

                * LEVEL indicates the level of the files to
                * be acted on by the ProDOS 8 OPEN, FLUSH, and
                * CLOSE commands.

BF94: 00        LEVEL    DFB   $00          ;Level for OPEN, FLUSH, CLOSE
BF95: 00        BUBIT    DFB   $00          ;SET_FILE_INFO backup bit flag
BF96: 00        SAVEP    DS    1            ;P register when MLI called

BF97: 00        SPARE1   DS    1            ;Unused/reserved

                * MACHID identifies the type of Apple being used,
                * the amount of memory available, and whether
                * an 80-column card or ProDOS-compatible clock
                * card is installed. Here is the meaning of the
                * bits in MACHID:
                *
                *       bits 7,6 (if bit 3 = 0)  00 = Apple II
                *                                01 = Apple II Plus
                *                                10 = Apple IIe or IIGS
                *                                11 = Apple III emul.
```

Table 3-2 Continued

```
                *
                *      bits 7,6 (if bit 3 = 1)   00 = [reserved]
                *                                01 = [reserved]
                *                                10 = Apple IIc
                *                                11 = [reserved]
                *
                *      bits 5,4   00 = [reserved]
                *                 01 = 48K
                *                 10 = 64K
                *                 11 = 128K (IIe, IIc, IIGS 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
                *
                * The example given is for an Apple IIGS, which has
                * a built-in 80-column card and clock.

BF98: B3        MACHID   DFB   $B3        ;GS, 80-columns, 128K

                * The high 7 bits of SLTBYT are used as
                * flags to indicate whether there is a
                * peripheral card with ROM on it in
                * slot (bit #). In the following example,
                * a byte is used that indicates ROM in
                * slots 1, 2, 3, 4, 5, 6, and 7.

BF99: FE        SLTBYT   DFB   $FE        ;Binary 11111110

                * PFIXPTR is a flag that indicates whether
                * a filename prefix has yet been defined.
                * If it hasn't, PFIXPTR will be 0, and full
                * (rather than partial) pathnames must be
                * specified when a ProDOS 8 command is
                * requested.

BF9A: 00        PFIXPTR  DFB   $00        ;Prefix flag (0 if no prefix)

                * The following four parameters are set up
                * whenever an MLI call (JSR $BF00) is made.

BF9B: 00        MLIACTV  DFB   $00        ;MLI flag (bit 7=1 if active)
```

Table 3-2 Continued

```
BF9C: 00 00    CMDADR   DA    $0000      ;Address+6 of last JSR to MLI
BF9E: 00       SAVEX    DFB   $00        ;X register when MLI called
BF9F: 00       SAVEY    DFB   $00        ;Y register when MLI called

               * All calls to the MLI eventually exit by
               * calling this subroutine with A = BNKBYT1
               * and bank1 of bank-switched RAM read-enabled.
               * EXIT restores the original state of the RAM
               * switches and returns control to the address
               * stored in CMDADR ($BF9C) via a "simulated" RTI.

BFA0: 4D 00 E0 EXIT     EOR   $E000      ;$E000 same as on entry?
BFA3: F0 05             BEQ   EXIT1      ;Yes, so RAM must be active
BFA5: 8D 82 C0          STA   $C082      ;No, so enable ROM
BFA8: D0 0B             BNE   EXIT2      ;(always taken)
BFAA: AD F5 BF EXIT1    LDA   BNKBYT2    ;Get $Dx bank code
BFAD: 4D 00 D0          EOR   $D000      ;Same as on entry?
BFB0: F0 03             BEQ   EXIT2      ;Yes, so bank1 RAM active
BFB2: AD 83 C0          LDA   $C083      ;Read-enable bank2 RAM
BFB5: 68       EXIT2    PLA
BFB6: 40                RTI              ;(returns to CMDADR)

               * This is a continuation of the standard
               * JSR MLI MLI call. It sets MLIACTV,
               * saves the status of bank-switched RAM,
               * and then enables bank1 of bank-
               * switched RAM before passing control
               * to ENTRYMLI.

BFB7: 38       MLICONT  SEC
BFB8: 6E 9B BF          ROR   MLIACTV    ;Set "MLI active" flag (bit 7)
BFBB: AD 00 E0          LDA   $E000
BFBE: 8D F4 BF          STA   BNKBYT1    ;Save RAM/ROM code
BFC1: AD 00 D0          LDA   $D000
BFC4: 8D F5 BF          STA   BNKBYT2    ;Save $Dx bank code
BFC7: AD 8B C0          LDA   $C08B
BFCA: AD 8B C0          LDA   $C08B      ;Read/Write bank1 RAM
BFCD: 4C 00 DE          JMP   ENTRYMLI   ;Go to RAM to do the rest

               * This is the tail end of the special ProDOS 8
               * interrupt-handling subroutine.

BFD0: AD 8D BF IRQXIT   LDA   INTBNKID   ;Get RAMcard status
BFD3: F0 0D    IRQXIT0  BEQ   IRQXIT2    ;Branch if bank1 $Dx enabled
BFD5: 30 08             BMI   IRQXIT1    ;Branch if bank2 $Dx enabled
BFD7: 4A                LSR              ;Is there a RAM card?
BFD8: 90 0D             BCC   ROMXIT     ;No, so branch
BFDA: AD 81 C0          LDA   $C081      ;Yes, so enable ROM
BFDD: B0 08             BCS   ROMXIT     ;(always taken)
```

Table 3-2 Continued

```
BFDF: AD 83 C0  IRQXIT1  LDA  $C083     ;Read-enable bank2 $Dx
BFE2: A9 01     IRQXIT2  LDA  #1
BFE4: 8D 8D BF           STA  INTBNKID  ;Set flag for ROM
BFE7: AD 88 BF  ROMXIT   LDA  INTAREG   ;Restore accumulator
BFEA: 40                 RTI            ;and finish up

                * The IRQ vector at $3FE/$3FF points here.
                * This code simply read- and write-enables bank1
                * of bank-switched RAM before passing control
                * to the ProDOS 8 interrupt handler that resides
                * there.

BFEB: 2C 8B C0  IRQENT   BIT  $C08B     ;Read- and ...
BFEE: 2C 8B C0           BIT  $C08B     ;... write-enable bank1 RAM
BFF1: 4C 4E DF           JMP  IRQRECEV  ;Go to IRQ handler

BFF4: 00        BNKBYT1  DFB  $00       ;RAM/ROM status stored here
BFF5: 00        BNKBYT2  DFB  $00       ;$Dx RAM bank status stored here

BFF6: 00 00 00           DS   6
BFF9: 00 00 00

                * IBAKVER is the earliest version number of the
                * ProDOS 8 kernel (MLI) that can be used by the
                * currently active system program (interpreter).
                * IVERSION is the version number of the system
                * program. When a system program is first
                * executed, it must set up these two parameters.

BFFC: 01        IBAKVER  DFB  $01       ;Earliest compatible kernel
BFFD: 01        IVERSION DFB  $01       ;Current interpreter version

                * KBAKVER is the earliest version number of the
                * ProDOS 8 kernel (MLI) that is compatible with
                * the current version number stored in KVERSION.

BFFE: 00        KBAKVER  DFB  $00       ;Earliest compatible version
BFFF: 07        KVERSION DFB  $07       ;Current ProDOS 8 version
```

THE GS/OS BOOT

A GS/OS system disk goes through a rather convoluted startup procedure when you boot it. It begins by loading the PRODOS program into memory and executing it. (This is the GS/OS version of PRODOS, of course.)

The first thing PRODOS does is check whether it's running on an Apple IIGS. If it's not, it displays the message

```
GS/OS REQUIRES APPLE IIGS HARDWARE
```

and the system hangs.

It then checks to see whether the IIGS has the correct version of the ROM installed. If it doesn't, it displays the following two lines:

```
GS/OS needs ROM version 01 or greater.
See your dealer for a ROM upgrade.
```

and the system hangs.

If PRODOS is running on an Apple IIGS with ROM version 01 or higher, it loads the file called START.GS.OS in the SYSTEM/ subdirectory and runs it. START.-GS.OS first initializes the state of the system by performing the following steps:

1. It initializes the Apple IIGS tool sets.
2. It installs the GS/OS program dispatcher (the code that handles the QUIT command).
3. It assigns the */ prefix to the name of the boot volume.
4. It saves the name of the startup file system translator.
5. It loads and installs the file called GS.OS from the SYSTEM/ subdirectory; this file contains the Apple IIGS System Loader tool set and the core of GS/OS.

   Note: The IIGS System Loader tool set is the one responsible for bringing GS/OS load files into memory. (Load files are executable applications created by the APW linker.)

START.GS.OS then loads the file called ERROR.MSG in the SYSTEM/ subdirectory; this file contains the text of all GS/OS error messages. By keeping the text in a single file like this, Apple can make foreign-language versions of GS/OS simply by translating the messages contained in this one file.

Next, it loads and installs the startup file system translator from the SYSTEM/FSTS/ subdirectory, usually the ProDOS file system translator, PRO.FST. (This file, like any file system translator file, must have a file type code of $BD.) Any other file system translators in this subdirectory are loaded and installed next. CHAR.FST, the character FST, is the other FST file that should be on the boot disk.

START.GS.OS then scans the system looking for character and disk devices. When it finds one, it tries to find a driver for it in the SYSTEM/DRIVERS/ subdirectory and load it if it is there. If there is no driver, START.GS.OS generates a generic driver in memory. The boot disk should include drivers for the keyboard/video device (CONSOLE.DRIVER), 3.5-inch disk drives (APPLEDISK3.5), and Apple SCSI hard disks

(SCSI.DRIVER); to enable access to 5.25-inch disk drives as well, include the APPLEDISK5.25 file. (SYSTEM/DRIVERS/ should also contain any printer drivers the Print Manager may need.)

START.GS.OS then executes the TOOL.SETUP program in the SYSTEM/SYSTEM.SETUP/ directory. TOOL.SETUP patches and enhances the IIGS's ROM-based tool sets; the patches are contained in the file called TS2.

START.GS.OS continues by loading and executing all the other files in the SYSTEM/SYSTEM.SETUP/ subdirectory that have file type codes of $B6 or $B7. $B6 files are permanent initialization (startup) files, and $B7 files are temporary initialization files. The difference between them is that temporary initialization files remove themselves from memory when they finish executing and permanent initialization files do not.

START.GS.OS then moves to the SYSTEM/DESK.ACCS/ directory and loads into memory any Classic Desk Accessory files (file type $B9) and New Desk Accessory files (file type $B8) it finds. This causes the names of the Classic Desk Accessories to be placed in the menu that appears when you press Control-Open-Apple-Esc. The names of the New Desk Accessories appear when you pull down the Apple menu in a standard desktop application like the Finder.

Next, START.GS.OS searches the SYSTEM/ directory for a file called START that has a file type of $B3 (S16). If it finds this file, it loads and executes it, and the boot process ends. START is usually the Finder, Apple's standard program-launching and disk/file-maintenance program.

If START.GS.OS does not find START, it scans the volume directory until it finds a ProDOS 8 system program (file type $FF) whose name ends with .SYSTEM or a GS/OS system program (file type $B3) whose name ends with .SYS16. It then ends the boot procedure by running the program. But it will not run a ProDOS 8 program unless SYSTEM/P8 is on the disk. P8 contains the code for the ProDOS 8 operating system, and if it's not there, START.GS.OS brings up a window asking the user to enter the pathname of the application to run.

The subdirectories we did not discuss, namely, TOOLS/, FONTS/, and LIBS/, do not participate in the boot procedure. The files they contain are there for the benefit of applications only. TOOLS/ contains IIGS tool set files for RAM-based tool sets; FONTS/ contains files containing font definitions that QuickDraw uses to draw characters in windows on the super hi-res graphics screen; and LIBS/ contains system library files.

GS/OS MEMORY USAGE

It is important for an application to know which areas of memory ProDOS 8 uses because Apple II computers predating the IIGS do not have a memory manager for allocating unused areas of memory and keeping track of used areas. ProDOS 8 applications running on a IIGS could use the IIGS's Memory Manager tool set, but few do because most developers don't want to create a special ProDOS 8 version just for the IIGS.

If an application didn't know what memory areas ProDOS 8 was using, it wouldn't know what areas it could use safely. But since the Apple IIGS does have a memory manager, knowledge of the memory areas GS/OS uses is much less important because the application can call the Memory Manager when it needs a safe block of memory to work with. In fact, all the application really needs to know about GS/OS memory usage is where it keeps important entry points and flags. Refer to *Exploring the Apple IIGS* or *Apple IIGS Toolbox Reference, Volume 1* for instructions on how to use the Memory Manager.

The code for GS/OS, the System Loader, and related IIGS system software occupies most of the language card areas in banks $00, $01, $E0, and $E1 of the 65816 memory space. The language card areas are not managed by the Memory Manager, so an application that uses the Memory Manager will never receive permission to use these areas. An application must not cheat and write to these unmanaged memory areas because they are strictly reserved.

(Other unmanaged memory areas are $0000–$0800 in banks $00 and $01 and $0000–$1FFF in banks $E0 and $E1. They are also reserved and must not be used by the application except for the text-page video RAM area $400–$7FF in banks $00 and $01. An application may store screen data directly to these areas if it needs to bypass the Text Tool Set or Console Driver to improve screen-output speed.)

GS/OS also uses the Memory Manager to allocate a work area in the upper end of bank $00, just below location $C000.

Table 3-3 summarizes the only locations most applications will ever need to know about. This includes the two standard command interpreter entry points we discuss in the next chapter and flags indicating what operating system is currently running, what operating system was originally booted, and whether GS/OS is busy.

Unlike ProDOS 8, GS/OS does not have a system global page that an application can examine to determine how many disk devices are connected to the system, what the system configuration is, what areas of memory have been reserved, and so on. Instead, an application can use GS/OS commands to keep track of disk devices, IIGS tool set functions to determine system configuration, and the Memory Manager to avoid memory conflicts.

Table 3-3 Important GS/OS memory locations

Address	*Meaning*
$E100A8–$E100AB	This is the inline command interpreter entry point. Applications can JSL to this address to perform the GS/OS command whose number and parameter table pointer follow the JSL instruction. (See Chapter 4.)
$E100B0–$E100B3	This is the stack-based machine command interpreter entry point. Applications can JSL to this address to perform the GS/OS command whose number and parameter table pointer have previously been pushed on the stack. (See Chapter 4.)
$E100BC	The OS_KIND byte. The value stored here indicates which operating system is currently running: $00 = ProDOS 8 $01 = GS/OS or ProDOS 16 Any other value indicates that no operating system is current. (This will be the case if the system is in the middle of a switch between ProDOS 8 and GS/OS, for example.) Technically, a $00 value at OS_KIND does not guarantee that ProDOS 8 is running since the user could have subsequently booted another operating system, like DOS 3.3, that does not change the OS_KIND byte. A favorite technique for determining whether ProDOS 8 is actually active is to check for a JMP opcode ($4C) at location $BF00 in bank $00.
$E100BD	The OS_BOOT byte. The value stored here indicates which operating system was initially booted: $00 = ProDOS 8 $01 = GS/OS or ProDOS 16
$E100BE–$E100BF	The GS/OS status flag word. Only bit 15 currently has meaning; if it is 1, GS/OS is busy, and no commands should be requested. This flag is for the benefit of desk accessories and interrupt handlers that may interrupt GS/OS in the middle of executing a command that is not reentrant.

CHAPTER 4

GS/OS and ProDOS 8 Commands

GS/OS and ProDOS 8 both have a low-level command interpreter that serves as an application's gateway to the operating system's commands. (The ProDOS 8 command interpreter is called the *machine language interface* or MLI.) Applications call the interpreter to perform various file-related operations, such as creating, deleting, opening, closing, reading, and writing files.

The command interpreter for GS/OS supports 47 commands, and the one for ProDOS 8 supports 26. (These totals will undoubtedly increase as Apple releases new versions of the operating systems.) You invoke these commands from an assembly-language program in the same general way, using standard calling protocols defined by Apple. The protocols for GS/OS and ProDOS 8 are structurally similar but not identical.

In this chapter, we take a close look at the GS/OS and ProDOS 8 MLI commands and see how to use them in assembly-language programs. In particular, we see how to

- Call specific commands
- Set up command parameter tables
- Identify error conditions
- Interpret error codes

Along the way we look at several brief programming examples which should clarify how to use operating system commands in your own programs.

USING PRODOS 8 MLI COMMANDS

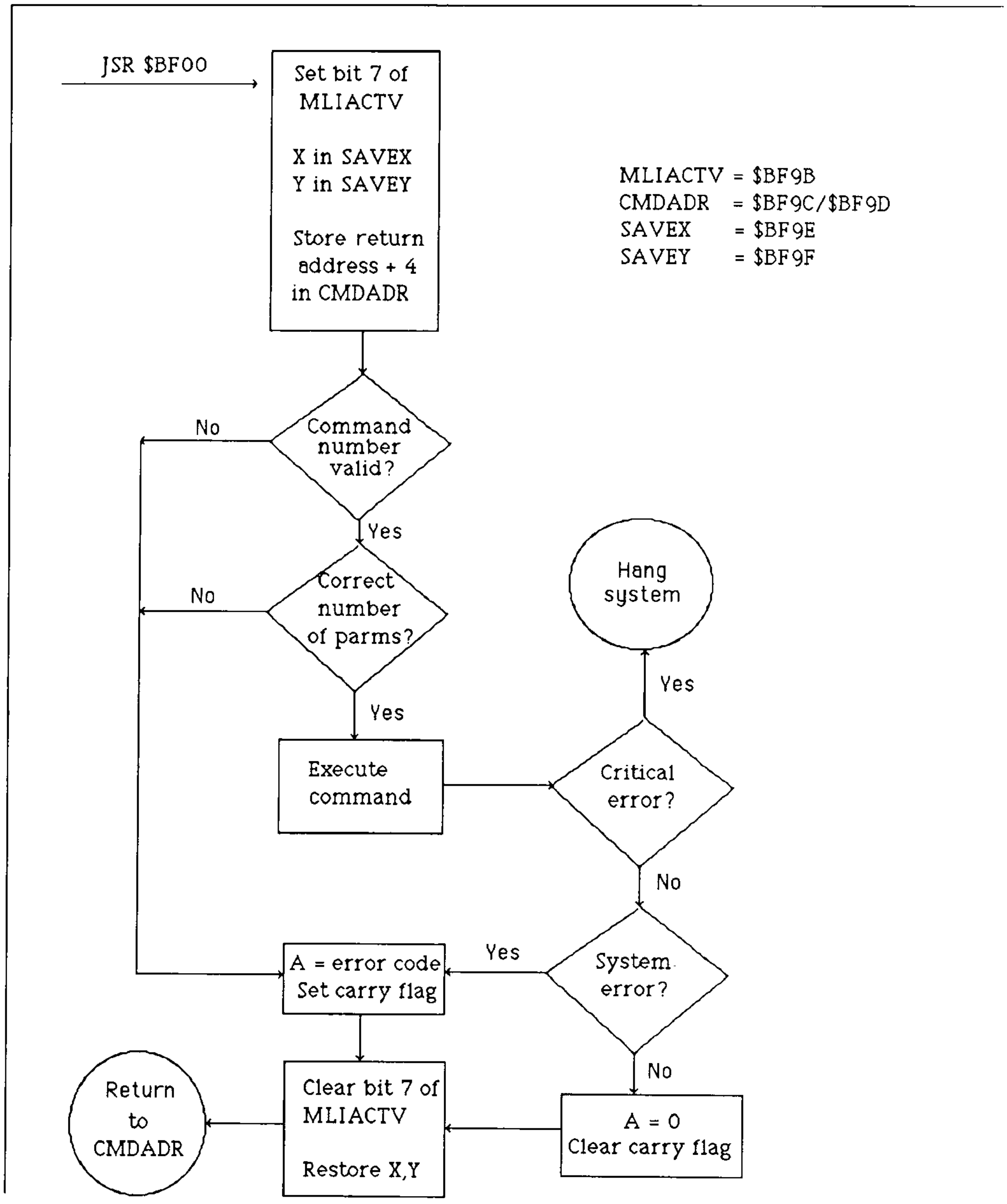
It is very easy to execute a ProDOS 8 MLI command. A typical calling sequence looks something like this:

```
         .
         .
         [place values in the parameter table
          before calling the MLI]
         .
         .
         JSR $BF00          ;$BF00 is the ProDOS 8 MLI entry point
         DFB CMDNUM         ;The MLI command number
         DA  PARMTBL        ;Address of command parameter table
         BCS ERROR          ;Carry is set if error occurred
         .
         .
         [continue your
          program here]
         .
         .
         RTS

ERROR    .
         [put an error
          handler here]
         .
         RTS

PARMTBL DFB NPARMS          ;NPARMS = # of parameters in table

         [place the rest of the parameters
          here in the order the MLI command
          expects]
```

The key instruction here is JSR $BF00. $BF00 is the address of the entry point to the ProDOS 8 MLI interpreter in main memory. This interpreter determines what MLI command the application is requesting and passes control to the appropriate ProDOS 8 subroutine to handle the request.

The flowchart in Figure 4-1 shows what happens when an application executes a JSR $BF00 instruction. As soon as the MLI takes control, it modifies four important variables in the ProDOS 8 global page area: MLIACTV ($BF9B), CMDADR ($BF9C/$BF9D), SAVEX ($BF9E), and SAVEY ($BF9F). First, it changes bit 7 of MLIACTV from 0 to 1 so that an interrupt-handling subroutine can determine if the interrupt condition occurred in the middle of an MLI operation. (We see why it's important to know this information in Chapter 6.) Next, it saves the current values in the X and Y registers in SAVEX and SAVEY. Finally, it stores the address of the instruction immediately following the three data bytes after the JSR $BF00 instruction at CMDADR.

Figure 4-1 Flowchart of ProDOS 8 MLI operations



Control passes to this address after ProDOS 8 executes the MLI command. (The MLI modifies the return address that the JSR places on the stack to ensure that control passes to this address rather than to the address following the JSR $BF00 instruction.)

The MLI determines which command the application is requesting by examining the value stored in the byte immediately following the JSR $BF00 instruction. This byte contains the unique identifier code (or *command number*) associated with the MLI command. If the MLI encounters an unknown command number, a system error occurs. (We see how to identify and handle such errors later in this chapter.) Table 4-1 lists all 26 ProDOS 8 commands and command numbers.

The 2 bytes following the command number contain the address (low-order byte first) of a parameter table the MLI command uses. This table begins with a byte holding the number of parameters in the table; the rest of the table holds data that the MLI command requires to process your request. After the MLI executes the command, the table also holds any results that are returned. We describe the contents of the parameter table for each MLI command later in this chapter.

The parameters an application passes to a ProDOS 8 MLI subroutine are of two types: *pointers* and *values*. A pointer is a 2-byte quantity that holds the address (low-order byte first) of a data structure it is said to be pointing to. (Typical data structures are an I/O buffer or an ASCII pathname preceded by a length byte.) A value is a 1-, 2-, or 3-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 1-, 2-, or 3-byte numeric quantity (with the low-order bytes first), but it can also be a 2-byte pointer, depending on the command involved.

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

While a command is being executed, a *critical error condition* may occur. Critical errors are very rare and occur only if ProDOS 8 data areas have been overwritten by a runaway program or if an interrupt occurs and no interrupt handler is available to deal with 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 is 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

Table 4-1 The ProDOS 8 MLI commands (in numerical order)

Command Name (number)	*Function*
ALLOC_INTERRUPT ($40)	Installs an interrupt-handling subroutine
DEALLOC_INTERRUPT ($41)	Removes an interrupt-handling subroutine
QUIT ($65)	Transfers control to another system program, usually through a dispatcher program
READ_BLOCK ($80)	Reads a data block from disk
WRITE_BLOCK ($81)	Writes a data block to disk
GET_TIME ($82)	Reads the current date and time
CREATE ($C0)	Creates a directory entry for a new file
DESTROY ($C1)	Removes the directory entry for an existing file or subdirectory and frees up the space it uses on disk
RENAME ($C2)	Renames a file
SET_FILE_INFO ($C3)	Changes the attributes for a file
GET_FILE_INFO ($C4)	Returns the attributes for a file
ON_LINE ($C5)	Determines the name of the volume directory for a disk
SET_PREFIX ($C6)	Sets the default pathname prefix
GET_PREFIX ($C7)	Returns the default pathname prefix
OPEN ($C8)	Opens a file for I/O operations
NEWLINE ($C9)	Sets the character that terminates a file read operation
READ ($CA)	Reads data from a file
WRITE ($CB)	Writes data to a file
CLOSE ($CC)	Closes a file
FLUSH ($CD)	Flushes a file buffer
SET_MARK ($CE)	Sets the value of the Mark (position-in-file) pointer
GET_MARK ($CF)	Returns the value of the Mark (position-in-file) pointer

Table 4-1 Continued

Command Name (number)	*Function*
SET_EOF ($D0)	Sets the value of the EOF (end-of-file) pointer
GET_EOF ($D1)	Returns the value of the EOF (end-of-file) pointer
SET_BUF ($D2)	Changes the position of a file buffer
GET_BUF ($D3)	Returns the position of a file buffer

The volume control, file control, and allocation blocks are internal data structures ProDOS 8 uses to handle disk volumes and to open files.

Normally, the MLI command starts finishing up by restoring the values of the X and Y registers (from SAVEX and SAVEY) and then, if a system error has occurred (see the next section), by executing a JSR $BF09 instruction. The subroutine at $BF09 (SYSERR) stores an error code in SERR ($BF0F).

Since the MLI preserves the contents of the X and Y registers, there is no need for the application to do so.

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

USING GS/OS COMMANDS

The general procedure for calling a GS/OS command is similar to the one for calling a ProDOS 8 MLI command. It goes something like this:

```
JSL   $E100A8          ;Call GS/OS entry point
DC    I2'CommandNum'   ;GS/OS command number
DC    I4'ParmTable'    ;Address of parameter table
BCS   Error            ;(Control resumes here after call)
```

$E100A8 is the address of the GS/OS command interpreter entry point. You can call this entry point while the IIGS's 65816 microprocessor is in either native or emulation mode.

Immediately following the JSL $E100A8 instruction is a word containing the identification number of the GS/OS command you wish to use. Table 4-2 lists all the GS/OS commands and command numbers.

Following the command number is the long address (4 bytes, low-order bytes first) of a parameter table containing parameters required by the command and spaces for results returned by the command. The parameters can be one- or two-word numeric values (a word is 2 bytes) or long pointers (4 bytes) and are stored with the low-order bytes first.

Table 4-2 The GS/OS commands (in numerical order)

Command Name (number)	*Function*
Create ($2001)	Creates a directory entry for a new file
Destroy ($2002)	Removes the directory entry for an existing file or subdirectory and frees up the space it uses on disk
OSShutdown ($2003)	Shuts down GS/OS in preparation for a cold reboot or a power down
ChangePath ($2004)	Renames a file or moves a file's directory entry to another subdirectory
SetFileInfo ($2005)	Changes the attributes for a file
GetFileInfo ($2006)	Returns the attributes for a file
Volume ($2008)	Returns the volume name, total number of blocks on the volume, number of free blocks on the volume, and the file system identification number for a given disk device
SetPrefix ($2009)	Sets the pathname prefix for any of the standard GS/OS prefixes (except */)
GetPrefix ($200A)	Returns the pathname prefix for any of the standard GS/OS prefixes (except */)
ClearBackup ($200B)	Clears the backup bit in the file's access code byte
SetSysPrefs ($200C)	Sets system preferences
Null ($200D)	Executes all queued signals
ExpandPath ($200E)	Creates a full pathname string
GetSysPrefs ($200F)	Returns system preferences
Open ($2010)	Opens a file for I/O operations
Newline ($2011)	Sets the character that terminates a file read operation
Read ($2012)	Reads data from a file
Write ($2013)	Writes data to a file
Close ($2014)	Closes a file
Flush ($2015)	Flushes a file buffer
SetMark ($2016)	Sets the value of the Mark (position-in-file) pointer
GetMark ($2017)	Returns the value of the Mark (position-in-file) pointer

Table 4-2 Continued

Command Name (number)	*Function*
SetEOF ($2018)	Sets the value of the EOF (end-of-file) pointer
GetEOF ($2019)	Returns the value of the EOF (end-of-file) pointer
SetLevel ($201A)	Sets the value of the system file level
GetLevel ($201B)	Returns the current value of the system file level
GetDirEntry ($201C)	Returns information about the file entries in a directory
BeginSession ($201D)	Begins a write-deferral session
EndSession ($201E)	Ends a write-deferral session
SessionStatus ($201F)	Returns write-deferral session status
GetDevNumber ($2020)	Returns the device number for a given device name
Format ($2024)	Formats a disk and writes out the boot blocks, volume bit map, and an empty root directory
EraseDisk ($2025)	Writes out the boot blocks, volume bit map, and an empty root directory to a disk
ResetCache ($2026)	Resizes the disk cache to the size stored in Battery RAM
GetName ($2027)	Returns the name of the application that is currently running
GetBootVol ($2028)	Returns the name of the disk GS/OS was booted from; (this is the name assigned to the boot prefix, */)
Quit ($2029)	Transfers control to another system program, usually through a dispatcher program
GetVersion ($202A)	Returns the GS/OS version number
GetFSTInfo ($202B)	Returns information about a file system translator
DInfo ($202C)	Returns the device name corresponding to a given device number
DStatus ($202D)	Returns the status of a device
DControl ($202E)	Sends control commands to a device
DRead ($202F)	Reads data from a device
DWrite ($2030)	Writes data to a device

Table 4-2 Continued

Command Name (number)	*Function*
BindInt ($2031)	Installs an interrupt-handling subroutine
UnbindInt ($2032)	Removes an interrupt-handling subroutine
FSTSpecific ($2033)	Sends FST-specific commands to a file system translator

The exact structure of the parameter table varies from command to command, but it always begins with a parameter count word called pcount. Generally, each GS/OS command allows a range of values for pcount, giving the application the choice of just how much information it wants to provide to the command and just how much it wants returned. The minimum and maximum pcount values for each GS/OS command are in the descriptions of the command table parameters, which we present later in this chapter.

When a command finishes, GS/OS adds 6 to the return address pushed on the stack by the JSL instruction and then ends with an RTL instruction. This causes control to pass to the code beginning just after the pointer to the parameter table. On return, all registers remain unchanged except the accumulator (which contains an error code), the program counter (of course), and the status register. (The m, x, D, I, and e flags are unchanged; N and V are undefined; the carry flag and zero flag reflect the error status.)

At this stage, you can check the state of the carry flag to determine whether an error occurred: If the carry flag is clear, there was no error; if it is not clear, an error did occur. Alternatively, you can check the zero flag; if an error occurred, it will be clear.

An error code indicating the nature of the error comes back in the accumulator; the accumulator will contain 0 if no error occurred. We describe GS/OS and ProDOS 8 error codes in detail in the next section.

The Apple Programmer's Workshop (APW) comes with a set of macros you can use to make it easier to call GS/OS commands. The macros are stored in a file called M16.GSOS on the APW disk. To use a GS/OS command with a macro, use an instruction of the form:

```
_CmdName ParmTbl
```

where CmdName represents the name of the command and ParmTbl represents the address of the parameter table associated with the command. At assembly time, this macro expands into the standard GS/OS calling sequence.

Note: All the macros for GS/OS commands in the M16.GSOS file have names that include a GS suffix. The macro for the Open command, for example, is called OpenGS. The reason for using the suffix is to ensure that the GS/OS macro names are different from their ProDOS 16 counterparts, making it possible to develop programs that use both GS/OS and ProDOS 16 commands. Since it's unlikely you'd ever want to mix commands, consider editing the M16.GSOS file to remove the suffixes. That way you won't have to worry about forgetting to include the suffix. The GS/OS command names used in this book do not include the GS suffix.

The main advantage of using the macros is you do not have to memorize command numbers, only command names. It also makes assembly-language programs that use GS/OS much easier to read.

Stack-Based Calling Method

You can also call a GS/OS command using a stack-based command interpreter entry point at $E100B0. Here is what such a call looks like:

```
PushPtr  ParmTbl      ;Push addr of parameter table
PushWord #CommandNum  ;Push GS/OS command number
JSL      $E100B0      ;Call stack-based entry point
```

To use this method, first push the 4-byte address of the command's parameter table and a 2-byte command number, and then perform a JSL $E100B0 instruction. PushPtr and PushWord are standard APW macros for doing this.

GS/OS AND PRODOS 8 ERROR HANDLING

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

If no system error occurred during execution of a command, the accumulator is 0, the carry flag is clear (0), and the zero flag is set (1).

If an error did occur, the accumulator holds the error code number, the carry flag is set (1), and the zero flag is clear (0). This means you can use a BCS or a BNE instruction to branch to the error-handling portion of your code.

You should always check for error conditions when a ProDOS 8 or GS/OS 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, it is often handy to have a special subroutine available that the application can call to print out helpful status information when an error occurs. Table 4-3 shows such a subroutine for ProDOS 8. When an application calls it, the message

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

Table 4-3 A standard ProDOS 8 MLI error-handling subroutine

```
                ************************************
                * General-Purpose MLI Error Handler *
                *                                  *
                *  Copyright 1985-1988 Gary Little  *
                *                                  *
                * Last modified: August 26, 1988   *
                *                                  *
                ************************************
                CMDADR   EQU   $BF9C      ;Return address for MLI call

                CROUT    EQU   $FD8E      ;Print a CR
                PRHEX    EQU   $FDDA      ;Print byte as two hex digits
                COUT     EQU   $FDED      ;Standard output subroutine

                         ORG   $300

0300: 48        ERROR    PHA              ;Save error code on stack

0301: A0 00              LDY   #0
0303: B9 2E 03  :1       LDA   ERRMSG,Y
0306: F0 06              BEQ   :2
0308: 20 ED FD           JSR   COUT       ;Print first part of message
030B: C8                 INY
030C: D0 F5              BNE   :1         ;(always taken)

030E: 68        :2       PLA              ;Get error code back
030F: 20 DA FD           JSR   PRHEX      ; and print it

0312: A0 00              LDY   #0
0314: B9 3B 03  :3       LDA   ERRMSG1,Y
0317: F0 06              BEQ   :4
0319: 20 ED FD           JSR   COUT       ;Print second part of message
031C: C8                 INY
031D: D0 F5              BNE   :3         ;(always taken)

031F: AD 9D BF  :4       LDA   CMDADR+1
0322: 20 DA FD           JSR   PRHEX      ;Print high part of address
0325: AD 9C BF           LDA   CMDADR
0328: 20 DA FD           JSR   PRHEX      ;Print low part of address
032B: 4C 8E FD           JMP   CROUT

032E: 8D        ERRMSG   DFB   $8D
032F: CD CC C9           ASC   "MLI ERROR $"
0332: A0 C5 D2 D2 CF D2 A0 A4
033A: 00                 DFB   0
033B: A0 CF C3  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                 DFB   0
```

appears on the screen, where xx is the two-digit hexadecimal error code, and yyyy is the address the ProDOS 8 MLI interpreter stored in CMDADR before trying to execute the command. This address is 6 bytes past the JSR $BF00 instruction that caused the error. You can easily adapt this program for use in a GS/OS environment.

Table 4-4 summarizes the system error codes which the GS/OS and ProDOS 8 command interpreters use. It also indicates the Applesoft error messages that BASIC. SYSTEM displays when it encounters an MLI error in a ProDOS 8 environment.

COMMAND DESCRIPTIONS

In the following sections, we examine, in alphabetical order, all the commands that make up GS/OS and ProDOS 8. The GS/OS command name and number appear in a box in the top left-hand corner of the first page of the command description; the ProDOS 8 name and number appear in a box in the top right-hand corner. By convention, ProDOS 8 names are all uppercase and may contain underscore characters; the corresponding GS/OS names contain both uppercase and lowercase characters and do not contain underscores.

Although many of the commands are available in both operating systems, some are unique. If a box contains the word *none*, the command is not available for the operating system to which the box corresponds.

Keep in mind that even where GS/OS and ProDOS 8 have commands that share the same name, the entries in the parameter tables are of different sizes and may be arranged in a different order. For example, GS/OS pointers are always 4 bytes long so that any address in the 65816 memory space may be accessed; ProDOS 8 pointers are only 2 bytes long, long enough to access any byte in the 6502 memory space. Moreover, parameters that are 1 or 2 bytes long in a ProDOS 8 parameter table are usually twice as long in the corresponding GS/OS parameter table.

The description of each command includes a summary of the command's GS/OS and ProDOS 8 parameter tables. These tables indicate the correct order of the parameters, the sizes of the parameters, and whether a parameter is an Input (I) or a Result (R). An Input is a parameter that must be provided before using the command. A Result (R) is a parameter that the command returns.

Class 0 and Class 1 Input Strings

Many commands require a pointer to a character string as an input parameter. ProDOS 8 uses class 0 character strings, where the first *byte* in the string space represents the length of the string (not including the length byte) and is followed by the ASCII-encoded bytes representing the characters. GS/OS uses class 1 character strings, where the first *word* in the string represents the length of the string. As with class 0 input strings, the character string is represented by a sequence of ASCII-encoded bytes.

In this book, an assembler macro called GSString is used to store a string preceded by a length word. The STR macro stores a string preceded by a length byte.

Table 4-4 GS/OS and ProDOS 8 command error codes

Error Code	*BASIC.SYSTEM Error Message*	*Meaning*
$00	[none]	No error occurred.
$01	I/O ERROR	The MLI command number is invalid.
$04	I/O ERROR	An incorrect number of parameters value was specified in the parameter table.
$07	[not applicable]	GS/OS is busy. This error can occur if you try to use GS/OS commands from inside an interrupt handler.
$10	[not applicable]	The specified device cannot be found. GS/OS reports this error after a GetDevNum command if it cannot locate the device.
$11	[not applicable]	The device reference number is invalid. GS/OS reports this error if the device number is not in its list of active devices.
$22	[not applicable]	Bad GS/OS driver parameter.
$23	[not applicable]	GS/OS Console Driver is not open.
$25	I/O ERROR	The ProDOS 8 internal interrupt vector table is full.
$27	I/O ERROR	A disk I/O error occurred that prevented the proper transfer of data. If you get this error, the disk is probably irreparably damaged. You will also get this error if there is no disk in a 5.25-inch disk drive.
$28	NO DEVICE CONNECTED	The specified disk drive device is not present. This error occurs if you try to access a second 5.25-inch drive when only one drive is present, for example.
$2B	WRITE PROTECTED	A write operation failed because the disk is write-protected.
$2E	I/O ERROR	An operation failed because a disk containing an open file has been removed from its drive.

Table 4-4 Continued

Error Code	*BASIC.SYSTEM Error Message*	*Meaning*
$2F	I/O ERROR	The specified device is off-line. This error occurs if there is no disk in a 3.5-inch drive.
$40	SYNTAX ERROR	The pathname syntax is invalid because one of the filenames or directory names specified does not follow the operating system 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 8 allows only eight files to be open at once.
$43	FILE NOT OPEN	The file reference number is invalid. This error occurs 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 one of the subdirectory names, in an otherwise valid pathname, does not exist.
$45	PATH NOT FOUND	The specified volume directory was not found. This means 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 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.

Table 4-4 Continued

Error Code	*BASIC.SYSTEM Error Message*	*Meaning*
$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 the operating system being used.
$4B	FILE TYPE MISMATCH	The storage type code for the file is invalid or not supported.
$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. This byte controls rename, destroy, read, and write operations. The error also occurs if you try to destroy a directory file that is not empty.
$4F	[not applicable]	The size of the GS/OS class 1 output buffer is too small.
$50	FILE BUSY	The command is invalid because the file is open. The OPEN, RENAME, and DESTROY commands operate only on closed files.
$51	I/O ERROR	The directory count is wrong. This error occurs if the file counter stored in the directory header is different from the actual number of files.
$52	I/O ERROR	This is not a ProDOS disk. This error occurs if the MLI senses a directory structure inconsistent with ProDOS.
$53	INVALID PARAMETER	A parameter is invalid because it is out of the allowable range.

Table 4-4 Continued

Error Code	*BASIC.SYSTEM Error Message*	*Meaning*
$54	[not applicable]	Out of memory.
$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 8 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.
$58	[not applicable]	The specified device is not a block device. Certain commands work with block-structured devices only.
$59	[not applicable]	The level parameter (passed to the GS/OS SetLevel command) is out of range.
$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.
$5B	[not applicable]	Illegal pathname change. This error occurs if the pathnames specified in the GS/OS ChangePath command refer to two different volumes. You can move files only between directories on the same volume.
$5C	[not applicable]	The specified file is not an executable system file. GS/OS reports this error if you attempt to use Quit to pass control to a file that is not a GS/OS system file (S16, code $B3) (EXE, code $B5) or a ProDOS 8 system file (SYS, code $FF).

Table 4-4 Continued

Error Code	*BASIC.SYSTEM Error Message*	*Meaning*
$5D	[not applicable]	The operating system specified is not available or not supported. GS/OS returns this error if you try to run a ProDOS 8 system program when the SYSTEM/P8 file is not on the system disk.
$5E	[not applicable]	/RAM cannot be removed.
$5F	[not applicable]	Quit Return Stack overflow. GS/OS returns this error if you try to push another program ID on the Quit Return Stack (using the Quit command) when the stack is already full.
$61	[not applicable]	End of directory. This error can be returned only by the GS/OS GetDirEntry command.
$62	[not applicable]	Invalid class number.
$64	[not applicable]	Invalid file system ID code.
$65	[not applicable]	Invalid FST operation.

NOTE: If the GS/OS Quit command results in an error, the error code is not returned to the application. Instead, the code appears in an interactive dialog box on the screen.

Class 0 and Class 1 Output Buffers

Even though a pointer to a string or a buffer area may be marked as a result in a parameter table, ProDOS 8 or GS/OS does not actually return the pointer. Instead, it returns data in the buffer pointed to by the pointer.

For ProDOS 8, it is the responsibility of the application to preallocate a buffer of the proper size and provide a pointer to it before calling a command. If you don't allocate a large enough buffer, data immediately following the buffer will be overwritten. Such a buffer is called a class 0 output buffer.

GS/OS uses class 1 output buffers to avoid the possibility of the operating system unexpectedly overwriting data areas if the preallocated output buffer is not big enough. A class 1 output buffer begins with a length word that holds the number of bytes in the buffer you've allocated (including the length word). When you call a command that uses a class 1 output buffer, GS/OS inspects the length word to see if the buffer is large enough; if it isn't, the command returns error code $4F ("buffer too small") and returns the size of the buffer it does need in the word following the buffer length word. If the buffer is large enough, the command returns data beginning at the byte following the length word.

(There is an exception. The output buffer you provide to GetDirEntry for returning a filename can be too small to hold the filename, but GetDirEntry does not return an error. Instead, it returns the actual length of the filename but puts only that portion of the filename that will fit in the output buffer.)

Prefixes

Be aware that no default prefix is in effect when ProDOS 8 first boots up. (There is for GS/OS.) This means any pathname specified in a ProDOS 8 MLI command parameter list must be a full pathname and not a partial pathname or a simple filename. To simplify your code, it is a good idea to use the SET_PREFIX command to set the prefix string to a convenient name before calling other ProDOS 8 commands. If you simply want to set the default prefix to the name of the volume directory on a given disk, use the ON_LINE command to get its name before using SET_PREFIX. An example of how to do this is included in the discussion of the SET_PREFIX command.

Access Code

Three of the commands, Create, GetFileInfo, and SetFileInfo, use a parameter called *access code* that describes the types of I/O operations an application may perform on a file as well as some other file attributes. Figure 2-10 in Chapter 2 shows the meaning of each bit in the access code.

Time and Date

Many ProDOS 8 commands accept or return date and time values in their parameter tables. These values are stored in the same special packed form used to store values in the ProDOS 8 system global page TIME and DATE locations. (See Figure 8-1 in Chapter 8 for a description of this format.)

GS/OS uses a different time and date format; it consists of eight bytes in the following order:

```
seconds
minutes
hour         in 24-hour military format
year         year minus 1900
day          day of month minus 1
month        0 = January, 1 = February, and so on
[not used]
weekday      1 = Sunday, 2 = Monday, and so on
```

This format is the same as the one used by the ReadTimeHex function in the IIGS's Miscellaneous Tool Set.

File Type Code

Another common command parameter is the *file type code*. For the ProDOS file system, this is a number from $00 to $FF that identifies the general file type. Table 2-5 in Chapter 2 gives the standard meanings of the ProDOS file type codes.

ProDOS 16 Considerations

The GS/OS commands described in this book are sometimes called class 1 commands. GS/OS also has a set of class 0 commands that are the same as the ProDOS 16 commands documented in the *Apple IIGS ProDOS 16 Reference*. The class 0 commands are not described here since they have been rendered almost obsolete by the class 1 commands. The only good reason for continuing to use class 0 commands is if you're writing a classic desk accessory—the CDA should be flexible enough to use ProDOS 8, GS/OS, or ProDOS 16 commands, depending on what operating system is active when it is called up.

none	ALLOC_INTERRUPT $40
GS/OS	ProDOS 8

Purpose:

To place the address of an interrupt-handling subroutine into the internal ProDOS 8 interrupt vector table. The interrupt vector table can hold up to four such subroutines.

Under GS/OS, use the BindInt command instead.

Parameter table:

ProDOS 8 Offset	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (2)
+1	int_num	R	Interrupt handler reference number
+2 to +3	int_code	I	Pointer to interrupt handler

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 2).
int_num	The reference number ProDOS 8 assigns to the interrupt-handling subroutine. Use this number when you remove the subroutine with the DEALLOC_INTERRUPT command.
int_code	A pointer to the beginning of the interrupt-handling subroutine. ProDOS 8 passes control to this subroutine when an interrupt occurs. The subroutine must begin with a CLD instruction. See Chapter 8 for a discussion of other rules and conventions ProDOS 8 interrupt-handling subroutines must follow.

Important: Install an interrupt-handling subroutine *before* enabling interrupts on the hardware device. If you don't, the system will crash if an interrupt occurs before you've had a chance to install the handler.

Common error codes:

$25	The interrupt vector table is full. Solution: Remove one of the active interrupt-handling subroutines (using DEALLOC_INTERRUPT) and try again.

Other possible error codes are $04, $53.

Programming example:

In Chapter 6, we take a closer look at how ProDOS 8 deals with interrupts and how to write interrupt-handling subroutines. Meanwhile, here's how to install a ProDOS 8 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          ;int_num is returned here
         DA  $300       ;Address of interrupt subroutine
```

Your application should store the returned int_num in a safe place so that it will be available when the interrupt-handling subroutine is removed with the DEALLOC_INTERRUPT command.

BeginSession **$201D**	**none**
GS/OS	**ProDOS 8**

Purpose:

To tell GS/OS to begin deferring all disk write operations that involve updating volume bit map and directory blocks.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (0)

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 0; the maximum is 0.

Common error codes:

[none]

Comments:

Write-deferral sessions are useful where your application wants to transfer a group of files from one disk to another as quickly as possible. If you don't use a write-deferral session, copying operations slow down because the disk read/write head must sweep across the disk medium to access volume bit map and directory blocks before and after each file transfer. (These blocks are usually physically located far from the file's data blocks.) By preventing these time-consuming head movements, you will maximize performance.

At the end of the copying operation, use the EndSession command to write to disk the blocks that were cached during the session. You must always balance every BeginSession call with an EndSession call.

BindInt **$2031**	**none**
GS/OS	**ProDOS 8**

Purpose:

To assign a GS/OS interrupt-handling subroutine to a particular interrupt source. Under ProDOS 8, use the ALLOC_INTERRUPT command instead.

Parameter table:

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (3)
+2 to +3	int_num	R	Interrupt reference number
+4 to +5	vrn	I	Vector reference number
+6 to +9	int_code	I	Pointer to interrupt handler

Descriptions of parameters:

pcount — The number of parameters in the GS/OS parameter table. The minimum value is 3; the maximum is 3.

int_num — The reference number GS/OS assigns to the interrupt-handling subroutine. Use this number when you remove the subroutine with the UnbindInt command.

vrn — A reference number that identifies the type of system interrupt the interrupt handler is to be assigned to:

$0008	AppleTalk (SCC)
$0009	Serial ports (SCC)
$000A	Scan-line retrace
$000B	Ensoniq waveform completion
$000C	Vertical blanking signal (VBL)
$000D	Mouse (movement or button)
$000E	1/4-second timer
$000F	Keyboard
$0010	ADB response byte ready
$0011	ADB service request (SRQ)
$0012	Desk accessory request keystroke
$0013	Flush keyboard buffer request keystroke
$0014	Keyboard micro abort
$0015	1-second timer

```
$0016    Video Graphics Controller (external)
$0017    Other interrupt source
```

(SCC is the Serial Communications Controller; ADB is the Apple Desktop Bus.)

If the interrupt emanates from a source that does not have a specific vrn, set vrn = $0017.

int_code	A pointer to the beginning of the interrupt-handling subroutine. See Chapter 8 for a discussion of rules and conventions GS/OS interrupt-handling subroutines must follow.

Important: Install an interrupt-handling subroutine *before* enabling interrupts on the hardware device. If you don't, the system will crash if an interrupt occurs before you've had a chance to install the handler.

Common error codes:

$25	The interrupt vector table is full. Solution: Remove one of the active interrupt-handling subroutines (using UnbindInt) and try again.

Other possible error codes are $04, $07, $53.

Comments:

See chapter 6 for a discussion of how to handle interrupts in a GS/OS environment.

ChangePath $2004	none
GS/OS	ProDOS 8

Purpose:

To rename a file or a disk volume or to move a file from one directory to another on the same disk volume. You can change the path of any closed file whose rename-enabled access code bit is set to 1.

Under ProDOS 8, use the RENAME command to rename a file or disk volume. There is no command for moving a file between two directories.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (2)
+2 to +5	pathname	I	Pointer to the pathname string
+6 to +9	new_pathname	I	Pointer to the new pathname string

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 2.
pathname	A pointer to a class 1 GS/OS string describing the current pathname of the file whose path is to be changed. If the pathname specified is not preceded by a separator (/ or :), the operating system appends the name to the default prefix (the 0/ prefix) to create a full pathname.
new_pathname	A pointer to a class 1 GS/OS string describing the new pathname of the file whose path is to be changed. If the pathname specified is not preceded by a separator (/ or :), the operating system appends the name to the default prefix (the 0/ prefix) to create a full pathname.

Common error codes:

$2B	The disk is write-protected.
$40	The pathname contains invalid characters, or a full pathname was not specified (and no default prefix has been set up). Verify 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 default prefix has been set up.

$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 default prefix.
$45	The volume directory was not found.
$46	The file was not found.
$47	The new pathname specified already exists. Solution: Give the file a new pathname not used by any other file on the disk volume.
$4E	The file cannot be accessed. Solution: Set the rename-enabled bit of the file's access code to 1 using SetFileInfo.
$50	The file is open. ChangePath works with closed files only.
$5B	The two pathnames indicate different volumes. You can use Change-Path only for moving files within a single volume.

Other possible error codes are $07, $27, $4A, $4B, $52, $57, $58.

Programming example:

Suppose you want to move a file called MY.ACCESSORY from a subdirectory called ASM: on the boot disk to the desk accessory directory on the boot disk. Here is the code you would use:

```
         _ChangePath CP_Parms
         RTS

CP_Parms ANOP
         DC     I2'2'           ;The number of parameters
         DC     I4'Curr_Name'
         DC     I4'New_Name'

Curr_Name GSString '*:ASM:MY.ACCESSORY'

New_Name  GSString '*:SYSTEM:DESK.ACCS:MY.ACCESSORY'
```

Note that when ChangePath moves a file from one subdirectory to another on the same disk, it moves only the file's subdirectory entry. The file's data stays put since the new subdirectory entry for the file still points to it. When the two paths specified describe files in the same subdirectory, ChangePath is equivalent to the ProDOS 8 RENAME command.

Note also that there are restrictions to keep in mind when moving a subdirectory into another subdirectory. The subdirectory you're moving cannot be part of the pathname for the target subdirectory.

ClearBackup $200B	none
GS/OS	ProDOS 8

Purpose:

To clear the backup-needed bit in the access code for the file.

Under ProDOS 8, use the SET_FILE_INFO command instead.

Parameter table:

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +5	pathname	I	Pointer to the pathname string

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
pathname	A pointer to a class 1 GS/OS string describing the current pathname of the file to be used. If the pathname specified is not preceded by a separator (/ or :), the operating system appends the name to the default prefix (the 0/ prefix) to create a full pathname.

Common error codes:

$40	The pathname contains invalid characters or a full pathname was not specified (and no default prefix has been set up). Verify 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 default prefix has been set up.
$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 default prefix.
$45	The volume directory was not found.
$46	The file was not found.

Other possible error codes are $07, $4A, $52, $58.

Programming example:

A file-backup program capable of doing incremental backups acts on only those files that have been modified since the last backup operation. The program checks the state of a file's backup bit to determine whether it needs to be backed up; it does if the bit is set to 1. (GS/OS and ProDOS 8 automatically set the bit after any write operation or any operation that changes the directory entry.) Once the backup copy has been made, the program should clear the backup bit by calling ClearBackup.

Here is the trivial piece of code for doing this:

```
          _ClearBackup CBB_Parms
          RTS

CBB_Parms ANOP
          DC   I2'1'             ;The number of parameters
          DC   I4'Pathname'      ;Pointer to pathname

Pathname  GSString '/DISK/NEW.FILE'    ;The file to act on
```

Close $2014	CLOSE $CC
GS/OS	ProDOS 8

Purpose:

To close an open file. This causes the operating system to write the contents of the data portion of the file's I/O buffer to disk (if necessary) and to update the file's directory entry. Once it does this, the operating system releases the memory used for the file's I/O buffer to the system and prevents further access to the file (until it is reopened).

Parameter table:

ProDOS 8		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0	num_parms	I	Number of parameters (1)
+1	ref_num	I	Reference number for the file

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +3	ref_num	I	Reference number for the file

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 1).
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
ref_num	The reference number the operating system assigned to the file when it was first opened. If you set ref_num to 0, all open files at or above the system file level are closed. To set the value of the file level under ProDOS 8, store the value at LEVEL ($BF94). Under GS/OS, use the SetLevel command.

Common error codes:

$2B	The disk is write-protected.
$43	The file reference number is invalid. You might be using a reference number for a file that you've already closed.

Other possible error codes are $04, $07, $27, $5A.

Programming example:

To close all open files at or above level 1, use SetLevel to set the level and use the Close command with ref_num set to 0. Here's how to do it if GS/OS is active:

```
          _SetLevel SL_Parms  ;Set system file level to 1
          _Close Cl_Parms
          BCS   Error         ;Branch if error occurred
          RTS

SL_Parms  DC    I2'1'
          DC    I2'1'         ;New file level

Cl_Parms  DC    I2'1'         ;Parameter count
          DC    I2'0'         ;reference number = 0 (close all files)
```

If ProDOS 8 is active, set the system file level by storing the new value at LEVEL ($BF94).

Create $2001	CREATE $C0
GS/OS	ProDOS 8

Purpose:

To create a new disk file. The operating system does this by placing an entry for the file in the specified directory. You must create every new file, except the volume directory file, with this command. (GS/OS automatically creates the volume directory when you use the Format or EraseDisk command. ProDOS 8 formatting programs create the volume directory by using the WRITE_BLOCK command to write an image of the four volume directory blocks to disk.)

Parameter table:

ProDOS 8		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0	num_parms	I	Number of parameters (7)
+1 to +2	pathname	I	Pointer to the pathname string
+3	access	I	Access code
+4	file_type	I	File type code
+5 to +6	aux_type	I	Auxiliary type code
+7	storage_type	I	Storage type code
+8 to +9	create_date	I	Creation date
+10 to +11	create_time	I	Creation time

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (7)
+2 to +5	pathname	I	Pointer to the pathname string
+6 to +7	access	I	Access code
+8 to +9	file_type	I	File type code

+10 to +13	aux_type	I	Auxiliary type code
+14 to +15	storage_type	I	Storage type code
+16 to +19	eof	I	Anticipated size of data fork
+20 to +23	resource_eof	I	Anticipated size of resource fork

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 7).
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 7.
pathname	A pointer to a class 0 (ProDOS 8) or class 1 (GS/OS) string describing the pathname of the file to be created. If the pathname specified is not preceded by a separator (/ for ProDOS 8; / or : for GS/OS), the operating system appends the name to the default prefix (in GS/OS, this is the 0/ prefix) to create a full pathname.
access	This field contains several 1-bit codes defining the access attributes of the file to be created. (The other bits must set to zero.) See Figure 2-10 for a description of these bits. The backup-needed bit of the access code is forced to 1 by this command.
file_type	A code indicating the type of data the file holds. See Table 2-5 for a description of the file type codes for the ProDOS file system.
aux_type	This is the auxiliary type code. The meaning of the code depends on the file type code and on the program that created the file in the first place. For SYS, BIN, BAS, and VAR files, it is a default loading address; for TXT files, it is a record length; for SRC files, it is an APW language type code.

storage_type This field indicates how the operating system is to store the file on the disk:

```
$00-$03     standard tree-structured data file
$05         extended file
$0D         linked-list directory file
```

If you specify a code of $00, $02, or $03, ProDOS 8 or GS/OS converts it to a code of $01 and returns that value in this field.

Note that you cannot change the storage_type of a file once it has been created.

create_date This field contains the date (year, month, day) that ProDOS 8 will save as the file's creation date. Figure 8-1 in Chapter 8 shows the

	format of these bytes. If these bytes are both zero, the current date will be used.
create_time	This field contains the time (hour, minute) that ProDOS 8 will save as the file's creation time. Figure 8-1 in Chapter 8 shows the format of these bytes. If these bytes are both zero, the current time will be used.
eof	If the file being created is a standard file (storage_type = $01), this field indicates the anticipated size of the file in bytes. GS/OS preallocates enough blocks on disk to hold a file of this size. If the file is an extended file (storage_type = $05), this field indicates the anticipated size of the data fork, in bytes. GS/OS preallocates enough blocks on disk to hold a data fork of this size. If the file is a subdirectory file (storage_type = $0D), this field indicates the anticipated number of entries in the subdirectory. GS/OS preallocates enough blocks on disk to hold a subdirectory of this size.
resource_eof	If the file being created is an extended file (storage_type = $05), this field indicates the anticipated size of the resource fork in bytes. GS/OS preallocates enough blocks on disk to hold a resource fork of this size.

Common error codes:

$2B	The disk is write-protected.
$40	The pathname contains invalid characters or a full pathname was not specified (and no default prefix has been set up). Verify 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 default prefix has been set up.
$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 default prefix.
$45	The volume directory was not found.
$47	The filename specified already exists. You can't have two files with the same name in the same subdirectory.
$48	The disk is full.
$49	The volume directory is full. Only 51 files can be stored in the volume directory.
$4B	Invalid storage type code. Solution: Set the storage type code to $0D for directory files, to $01 for standard data files, or (for GS/OS only) to $05 for extended files.

Other possible error codes are $04, $07, $10, $27, $52, $53, $58.

Programming example:

Here is a short GS/OS subroutine you can use to create a standard textfile; the filename for the textfile is JUPITER, and the full pathname is :PLANETS:JUPITER.

```
          _Create Cr_Parms
          BCS     Error     ;Branch if error occurred
          RTS

Cr_Parms  DC   I2'5'        ;Only using 5 parameters
          DC   I4'PathName'
          DC   I2'$E3'      ;standard access code (unlocked)
          DC   I2'$04'      ;file type = 4 (textfile)
          DC   I4'0'        ;auxiliary type (0 = sequential)
          DC   I2'$01'      ;storage type = 1 (standard file)
PathName  GSString ':PLANETS:JUPITER' ;Pathname (in ASCII)
```

Note that when you create a file under GS/OS, the date and time of creation is always set to the current date and time. (Under ProDOS 8 you can specify any time you want in the parameter table for CREATE.) To set a different date and time of creation, use the SetFileInfo command.

DControl $202E	none
GS/OS	ProDOS 8

Purpose:

To send control commands to a GS/OS device.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (5)
+2 to +3	dev_num	I	Device reference number
+4 to +5	control_code	I	Control request code
+6 to +9	control_list	I	Pointer to control list
+10 to +13	request_count	I	Size of the control list
+14 to +17	transfer_count	R	Number of bytes transferred

Descriptions of parameters:

pcount — The number of parameters in the GS/OS parameter table. The minimum value is 5; the maximum is 5.

dev_num — The device's reference number.

control_code — A code indicating what control operation is to be performed:

$0000	reset device
$0001	format device medium
$0002	eject device medium
$0003	set configuration parameters
$0004	set wait/no-wait mode
$0005	set format options
$0006	assign partition owner
$0007	arm signal
$0008	disarm signal
$0009	set partition map
$000A-$7FFF	[reserved]
$8000-$FFFF	device-specific operations

control_list	This is a pointer to a buffer that contains any supplementary data that GS/OS may need to perform the control operation.
request_count	The size of the control list buffer.
transfer_count	The number of bytes in the control list buffer that were transferred to the device is returned here.

Common error codes:

$11	The device reference number is invalid.
$53	The parameter is out of range.

Another possible error code is $07.

Programming example:

The only control command you're ever likely to need for a disk device is the eject command. Here is a GS/OS subroutine for ejecting the disk medium from a drive:

```
          _DControl DC_Parms
          RTS

DC_Parms  ANOP
          DC   I2'5'          ;The number of parameters
          DC   I2'2'          ;Device number
          DC   I2'2'          ;Control code (2 = eject)
          DC   I4'Ctrl_List'
          DC   I4'0'
          DS   4

Ctrl_List DS   4              ;Nothing in control list
```

You can determine if the disk medium is removable by doing a DInfo call and examining bit 2 of the characteristics word; if the bit is 1, the medium is removable.

You will use several device-specific control commands to communicate with the Console Driver (see chapter 9). For a detailed discussion of the standard control commands, see *GS/OS Reference, Volume 2.*

GS/OS	ProDOS 8
none	**DEALLOC_INTERRUPT $41**

Purpose:

To remove the address of an interrupt-handling subroutine from the internal ProDOS 8 interrupt vector table.

Under GS/OS, use the UnbindInt command instead.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (1)
+1	int_num	I	Interrupt handler reference number

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 1).
int_num	The identification number for the interrupt handler. ProDOS 8 assigned this number when the handler was installed using the ALLOC_INTERRUPT command.

Important: Do not remove an interrupt-handling subroutine until your application has first told the source of the interrupts to stop generating interrupts. If you remove the subroutine first, the system will crash the next time an interrupt occurs.

Common error codes:

$53	The int_num parameter is not valid. Use the number ALLOC_INTERRUPT returned when you installed the interrupt handler.

Another possible error code is $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 $2002	DESTROY $C1
GS/OS	ProDOS 8

Purpose:

To remove a file from disk. When you destroy a file, the operating system frees up all the disk blocks the file uses and zeros the length byte in the file's directory entry. You can destroy any file (except a volume directory file) whose destroy-enabled access code bit is set to 1; subdirectory files must be empty before you can destroy them, however.

Parameter table:

ProDOS 8		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0	num_parms	I	Number of parameters (1)
+1 to +2	pathname	I	Pointer to the pathname string

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +5	pathname	I	Pointer to the pathname string

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 1).
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
pathname	A pointer to a class 0 (ProDOS 8) or class 1 (GS/OS) string describing the pathname of the file to be destroyed. If the pathname specified is not preceded by a separator (/ for ProDOS 8; / or : for GS/OS), the operating system appends the name to the default prefix (in GS/OS, this is the 0/ prefix) to create a full pathname. If the pathname describes an extended file (storage_type = $05), both forks are destroyed.

Common error codes:

$2B	The disk is write-protected.
$40	The pathname contains invalid characters, or a full pathname was not specified (and no default prefix has been set up). Verify 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 default prefix has been set up.
$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 default 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 default prefix.
$46	The file was not found.
$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. You can destroy closed files only.

Other possible error codes are $04, $07, $10, $27, $4A, $4B, $52, $58.

Programming example:

Consider a situation in which the 0/ prefix is /DEMOS/GAMES. To destroy a file that has a full pathname of /DEMOS/GAMES/TRIVIA.BLITZ, you could use the following GS/OS subroutine.

```
          _Destroy DY_Parms
          BCS Error       ;Branch if error occurred
          RTS

DY_Parms  DC   I2'1'         ;1 parameter
          DC   I4'PathName'

PathName  GSString 'TRIVIA.BLITZ' ;Pathname (in ASCII)
```

Notice that it was not necessary to specify the full pathname in this program. GS/OS automatically appends the name specified to the 0/ prefix to create the full pathname that it acts on.

The ProDOS file system does several things when it destroys a file. First, it zeros the name_length byte in the file's directory entry. (This is the first byte in the entry.) Then it frees up the disk blocks the file uses by setting the appropriate bits in the volume bit map. Finally, it reads in the file's index blocks from disk, reverses the two 256-byte halves of each block (meaning the low-order block number appears in the upper half, and the high-order block number appears in the lower half), and then writes the blocks back to disk. (Versions of ProDOS 8 numbered 1.2 or lower actually zeroed the index blocks, making it impossible for a utility program to recover a deleted file.)

Note that you cannot destroy an extended file (storage_type = $05) with the ProDOS 8 version of the DESTROY command. It can be destroyed only with the GS/OS Destroy command.

DInfo **$202C**	**none**
GS/OS	**ProDOS 8**

Purpose:

To determine information about a device connected to the system.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input* *or* *Result*	*Description*
+0 to +1	pcount	I	Number of parameters (10)
+2 to +3	dev_num	I	Device reference number
+4 to +7	dev_name	R	Pointer to the device name string
+8 to +9	characteristics	R	Device characteristics
+10 to +13	total_blocks	R	Capacity of volume, in blocks
+14 to +15	slot_num	R	Slot number for device
+16 to +17	unit_num	R	Unit number for device
+18 to +19	version	R	Device driver version number
+20 to +21	device_ID_num	R	Device ID number
+22 to +23	head_link	R	First related device
+24 to +25	forward_link	R	Next related device

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 10.
dev_num	The device's reference number.
dev_name	A pointer to a class 1 output buffer in which GS/OS returns the device name. A device name may be up to 31 characters long, so set the buffer size word in the class 1 output buffer to 35 bytes.
characteristics	The bits in this word reflect the characteristics of the device:

bit 15	1 = device is a RAMdisk or ROMdisk
bit 14	1 = device driver was generated
bit 13	[reserved]
bit 12	1 = device is busy
bit 11	[reserved]
bit 10	[reserved]
bit 9	device speed (high)
bit 8	device speed (low)
bit 7	1 = device is a block device
bit 6	1 = write is allowed
bit 5	1 = read is allowed
bit 4	[reserved]
bit 3	1 = format is allowed
bit 2	1 = device contains removable media
bit 1	[reserved]
bit 0	[reserved]

Bits 9 and 8, the device speed bits, indicate the speed at which the device can operate:

00	1 MHz device
01	2.6 MHz device
10	>2.6 MHz device
11	not speed dependent

total_blocks For a block device, the capacity of the volume in blocks. For a character device, this field is zero.

slot_num The slot number of the firmware driver for the device.

unit_num The SmartPort unit number for the device.

version The version number of the device driver:

```
bits 15–12   major version number
bits 11–8    primary minor version number
bits  7–4    secondary minor version number
bits  3–0    version type:
                 $0 = released final
                 $A = alpha
                 $B = beta
                 $E = experimental
                 $F = unreleased final
```

For example, version 2.12 beta would be represented by the version word $212B.

device_ID_num This is a code number that identifies the device type:

$0000	5.25-inch disk drive
$0001	ProFile hard disk (5Mb)
$0002	ProFile hard disk (10Mb)
$0003	3.5-inch disk drive
$0004	generic SCSI device

$0005	SCSI hard disk
$0006	SCSI tape drive
$0007	SCSI CD-ROM drive
$0008	SCSI printer
$0009	serial modem
$000A	console
$000B	serial printer
$000C	serial LaserWriter
$000D	AppleTalk LaserWriter
$000E	RAM Disk
$000F	ROM Disk
$0010	file server
$0011	IBX telephone
$0012	Apple desktop bus device
$0013	generic hard disk drive
$0014	generic floppy disk drive
$0015	generic tape drive
$0016	generic character device
$0017	MFM-style floppy disk drive
$0018	generic AppleTalk network device
$0019	SCSI sequential access device
$001A	SCSI scanner
$001B	non-SCSI scanner
$001C	SCSI LaserWriter
$001D	AppleTalk main driver
$001E	AppleTalk file service driver
$001F	AppleTalk RPM driver

head_link	This is a device number that is the first entry in a linked list of device numbers. The devices in the list are related in that they each have a distinct partition on the same disk medium. If head_link is zero, there is no link.
forward_link	This is a device number that is the next entry in a linked list of device numbers. The devices in the list are related in that they each have a distinct partition on the same disk medium. If forward_link is zero, there is no link.

Common error codes:

$11	Invalid device reference number.

Another possible error code is $07.

Programming example:

You can use DInfo to determine the names of all the devices connected to the system. To do this, make a series of calls to DInfo, incrementing dev_num by 1 after each call, until DInfo returns an error code of $11 ("invalid device reference number"). The first dev_num you pass to DInfo should be 1 since this is the device number GS/OS assigns to the first device it finds when it boots up.

Keep in mind, however, that the number of active devices in the system may change during program execution. For example, server volumes may come on line or go off line at almost any time. As a result, if you're designing a program which has a "list volumes" command, you should form the list each time the user requests it. It is not good enough to form the list once at the beginning of the program.

Here is a GS/OS code fragment that shows how you might do this in an application:

```
         LDA     #1
         STA     DevNum

Get_Name _DInfo  DI_Parms
         BCS     Exit

         LDA     DevName       ;Get length word
         XBA                   ;(Put low-order byte at
         STA     DevName       ;beginning of string)
         PushPtr DevName+1     ;(point to length byte)
         _DrawString           ;Display name in window
         JSR     CRLF          ;(CRLF moves cursor to next line)
         BRA     Get_Name

Exit     RTS

DI_Parms ANOP
         DC      I2'10'        ;The number of parameters
DevNum   DC      I2'1'         ;Device number
         DC      I4'DevSpace'  ;Pointer to device name buffer

DevSpace DC      I2'35'        ;Size of buffer
DevName  DS      33            ;Name stored here
```

Call this subroutine after positioning the cursor with the _Move or _MoveTo macro. _DrawString is the macro for a QuickDraw II tool set function that displays a Pascal-like string (one preceded by a length byte) in the current window.

DRead $202F	none
GS/OS	ProDOS 8

Purpose:

To perform low-level read operations on a GS/OS device.

Under ProDOS 8, use the READ_BLOCK command instead.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (6)
+2 to +3	dev_num	I	Device reference number
+4 to +7	buffer	R	Data buffer
+8 to +11	request_count	I	Number of bytes to read
+12 to +15	starting_block	I	First block to read from
+16 to +17	block_size	I	Number of bytes per block
+18 to +21	transfer_count	R	Number of bytes actually read

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 6; the maximum is 6.
dev_num	The device's reference number.
buffer	A pointer to a class 0 output buffer into which the data is to be read.
request_count	The number of bytes to read.
starting_block	If the device is a block device, this is the number of the block to start reading from. For character devices, this field is not used.
block_size	The size of a block in bytes.
transfer_count	The number of bytes actually read from the device.

Common error codes:

$11	The device reference number is invalid.
$53	Parameter out of range.

Another possible error code is $07.

Programming example:

For block-structured devices, DRead is most often used to read the contents of data blocks on the disk volume. Here is a GS/OS subroutine you could use to read blocks 6 and 7 on a disk volume containing 512-byte blocks:

```
          _DRead DR_Parms
          RTS

DR_Parms  DC   I2'6'          ;The number of parameters
          DC   I2'2'          ;Device number
          DC   I4'Buffer'
          DC   I4'1024'       ;Read 1024 bytes
          DC   I4'100'        ; ... starting with block 100
          DC   I2'512'        ;512 bytes per block
          DS   4              ;transfer_count result

Buffer    DS   1024
```

Note that after DRead reads the 512 bytes in block 100, it proceeds to the next higher-numbered block, 101, to read the next 512 bytes.

DStatus **$202D**	**none**
GS/OS	**ProDOS 8**

Purpose:

To determine the status of a GS/OS device.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (5)
+2 to +3	dev_num	I	Device reference number
+4 to +5	status_code	I	Control request code
+6 to +9	status_list	R	Pointer to control list
+10 to +13	request_count	I	Size of the control list
+14 to +17	transfer_count	R	Number of bytes transferred

Descriptions of parameters:

pcount — The number of parameters in the GS/OS parameter table. The minimum value is 5; the maximum is 5.

dev_num — The device's reference number.

status_code — A code indicating what status request is to be made:

```
$0000        get device status
$0001        get configuration parameters
$0002        get wait/no-wait status
$0003        get format options
$0004        get partition status
$0005-$7FFF  [reserved]
$8000-$FFFF  device-specific status calls
```

status_list — This is a pointer to a class 0 buffer that holds any status data that the status call may return.

request_count — The number of status bytes to be returned in the status list.

transfer_count — The actual number of bytes returned in the status list is returned here.

Common error codes:

\$11	The device reference number is invalid.
\$53	Parameter out of range.

Another possible error code is \$07.

Comments:

Your application should rarely have to use the DStatus command unless it is communicating with the Console Driver (see Chapter 9). For a discussion of the standard low-level status commands, see *GS/OS Reference, Volume 2*.

DWrite $2030	none
GS/OS	ProDOS 8

Purpose:

To perform low-level write operations on a GS/OS device.

Under ProDOS 8, use the WRITE_BLOCK command instead.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (6)
+2 to +3	dev_num	I	Device reference number
+4 to +7	buffer	I	Data buffer
+8 to +11	request_count	I	Number of bytes to write
+12 to +15	starting_block	I	First block to write to
+16 to +17	block_size	I	Number of bytes per block
+18 to +21	transfer_count	R	Number of bytes actually written

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 6; the maximum is 6.
dev_num	The device's reference number.
buffer	A pointer to a buffer in which the data to be written is stored.
request_count	The number of bytes to write.
starting_block	If the device is a block device, this is the number of the block to start writing to. For character devices, this field is not used.
block_size	The size of a block, in bytes.
transfer_count	The number of bytes actually written to the device.

Common error codes:

$11	The device reference number is invalid.
$53	Parameter out of range.

Another possible error code is $07.

Comments:

This command is for low-level transfer of data to a character or block device. The file system on the block device is not relevant.

EndSession **$201E**	**none**
GS/OS	**ProDOS 8**

Purpose:

To perform all disk block write operations that have not been made because a write-deferral session is in progress. EndSession also terminates the current write-deferral session.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (0)

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 0; the maximum is 0.

Common error codes:

[none]

Comments:

You must call EndSession if your application began a disk-deferral session by calling BeginSession and wants to close the session.

EraseDisk **$2025**	**none**
GS/OS	**ProDOS 8**

Purpose:

To write to disk the boot record, volume bit map, and empty root directory for the specified file system. Unlike Format, EraseDisk does not initialize the disk first, so you can use it only with previously initialized disks.

There is no equivalent ProDOS 8 command. Under ProDOS 8, you must use WRITE_BLOCK to perform the required disk-write operations needed to erase a disk.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (4)
+2 to +5	dev_name	I	Pointer to the device name string
+6 to +9	vol_name	I	Pointer to the volume name string
+10 to +11	file_sys_id	R	ID code for selected file system
+12 to +13	requested_fsys	I	ID code for requested file system

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 3; the maximum is 4.
dev_name	A pointer to a class 1 device name string.
vol_name	A pointer to a class 1 disk volume name string. The name must be preceded by a slash.
file_sys_id	If the requested_fsys field is zero, GS/OS displays a dialog box that lets the user pick the file system to be used on the disk volume. On return, the file_sys_id field indicates which file system was selected:

$01 = ProDOS/SOS
$02 = DOS 3.3
$03 = DOS 3.2/3.1
$04 = Apple II Pascal
$05 = Macintosh MFS

$06	Macintosh HFS
$07	Macintosh XL (LISA)
$08	Apple CP/M
$09	[never used]
$0A	MS-DOS
$0B	High Sierra (CD-ROM)
$0C	ISO 9660 (CD-ROM)

If GS/OS returns a zero in this field, the user canceled the operation.

requested_fsys This field contains the ID code for the file system to be written to the disk volume. (The codes are the same as those described for file_sys_id.) If the field is zero, GS/OS displays a dialog box that lets the user pick his or her own file system; GS/OS returns the selected ID in the file_sys_id field.

Common error codes:

$10	The specified device name does not exist.
$40	The volume name specified contains invalid characters or does not start with a valid separator (/ or :).
$5D	The specified file system is not supported.

Other possible error codes are $07, $11, $27.

Programming example:

Suppose you want to erase a disk whose device name is .APPLEDISK3.5A and give it the name :BLANK. Here is the GS/OS subroutine to use:

```
EraseIt  _EraseDisk ED_Parms
         RTS

ED_Parms ANOP
         DC   I2'4'          ;The number of parameters
         DC   I4'DevName'    ;Pointer to device name
         DC   I4'VolName'    ;Pointer to volume name
         DS   2              ;file_sys_id
         DC   I2'0'          ;0 = let user pick

DevName  GSString '.APPLEDISK3.5A'
VolName  GSString ':BLANK'
```

GS/OS	ProDOS 8
ExpandPath **$200E**	**none**

Purpose:

To convert a filename, partial pathname, or full pathname into a full pathname with colon separators.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (3)
+2 to +5	input_path	I	Pathname to be expanded
+6 to +9	output_path	R	Pointer to expanded pathname
+10 to +11	flags	I	Uppercase conversion flag

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 3.
input_path	Pointer to a class 1 pathname string that is to be expanded.
output_path	Pointer to a class 1 output buffer where GS/OS returns the expanded pathname.
flags	Bit 15 of this flag indicates whether lowercase characters are to be converted to uppercase:

```
bit 15       1 = convert to uppercase characters
             0 = don't convert characters

bits 14-0    must be zero
```

Common error codes:

$40	The pathname syntax is invalid.
$4F	The class 1 output buffer is too small to hold the result.

Comments:

The input_path parameter does not have to represent an existing filename on disk.

Flush $2015	FLUSH $CD
GS/OS	ProDOS 8

Purpose:

To force the operating system to write the contents of the data portion of a file's I/O buffer to disk and to update the file's directory entry. The operating system does this without closing the file.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (1)
+1	ref_num	I	Reference number for the file

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +3	ref_num	I	Reference number for the file

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 1).
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
ref_num	The reference number ProDOS 8 or GS/OS assigned to the file when it was first opened. If ref_num is 0, all open files at or above the system file level are flushed. To set the value of the file level under ProDOS 8, store the value at LEVEL ($BF94). Under GS/OS, use the SetLevel command.

Common error codes:

$2B	The disk is write-protected.

$43	The file reference number is invalid. You might be using a reference number for a file that you've already closed.

Other possible error codes are $04, $07, $27, $48.

Programming example:

To flush all open ProDOS 8 files at or above file level 2, use the FLUSH command with ref_num 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)
```

Format $2024	none
GS/OS	ProDOS 8

Purpose:

To format a disk and write out the boot record, volume bit map, and empty root directory for the specified disk operating system.

There is no equivalent ProDOS 8 command. Under ProDOS 8, You must use a utility program like System Utilities to format a disk.

Parameter table:

GS/OS Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (4)
+2 to +5	dev_name	I	Pointer to the device name string
+6 to +9	vol_name	I	Pointer to the volume name string
+10 to +11	file_sys_id	R	ID code for selected file system
+12 to +13	requested_fsys	I	ID code for requested file system

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 3; the maximum is 4.
dev_name	A pointer to a class 1 device name string.
vol_name	A pointer to a class 1 disk volume name string. The name must be preceded by a slash.
file_sys_id	If the requested_fsys field is zero, GS/OS displays a dialog box that lets the user pick the file system to be used on the disk volume. On return, the file_sys_id field indicates which file system was selected:

$01 = ProDOS/SOS
$02 = DOS 3.3
$03 = DOS 3.2/3.1
$04 = Apple II Pascal
$05 = Macintosh MFS
$06 = Macintosh HFS
$07 = Macintosh XL (LISA)

$08	= Apple CP/M
$09	= [never used]
$0A	= MS-DOS
$0B	= High Sierra (CD-ROM)
$0C	= ISO 9660 (CD-ROM)

If GS/OS returns a zero in this field, the user canceled the operation.

requested_fsys This field contains the ID code for the file system to be written to the disk volume. (The codes are the same as those described for file_sys_id.) If the field is zero, GS/OS displays a dialog box that lets the user pick his or her own file system; GS/OS returns the selected ID in the file_sys_id field.

Common error codes:

$10	The specified device name does not exist.
$40	The volume name specified contains invalid characters or does not start with a valid separator (/ or :).
$5D	The specified file system is not supported.

Other possible error codes are $07, $11, $27.

Programming example:

See the example given for the EraseDisk command. The only change to make is to replace the _EraseDisk macro with the _Format macro.

FSTSpecific **$2033**	**none**
GS/OS	**ProDOS 8**

Purpose:

To perform operations which are unique to a particular file system translator.

There is no equivalent ProDOS 8 command. ProDOS 8 does not use file system translators.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (3)
+2 to +3	file_sys_id	I	File system ID code
+4 to +5	command_num	I	FST-specific command number
+6 to +7/9	command_parm	I/R	Command parameter or result

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 3; the maximum is 3.
file_sys_id	This field indicates the file system that the FST implements:

$01	ProDOS/SOS
$02	DOS 3.3
$03	DOS 3.2/3.1
$04	Apple II Pascal
$05	Macintosh MFS
$06	Macintosh HFS
$07	Macintosh XL (LISA)
$08	Apple CP/M
$09	Character FST
$0A	MS-DOS
$0B	High Sierra (CD-ROM)
$0C	ISO 9660 (CD-ROM)

command_num This field contains an FST-specific command code.

command_parm	This can be either an Input or a Result field, depending on command_num. Its meaning depends on which FST you are communicating with.

Common error codes:

$53	Invalid parameter.

Other possible error codes are $04, $54.

Comments:

This command is for performing operations unique to a particular file system. The nature of these operations varies from one FST to another. Consult the technical description of the FST you want to deal with for an explanation of the FSTSpecific calls it supports.

GetBootVol $2028	none
GS/OS	ProDOS 8

Purpose:

To determine the name of the disk volume from which the GS/OS operating system was booted.

There is no equivalent ProDOS 8 command. ProDOS 8 does not keep track of the name of the disk it was booted from.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +5	vol_name	R	Pointer to the volume name string

Note: The volume name GetBootVol returns is the same as the name GS/OS assigns to the */ prefix when it first boots up.

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
vol_name	A pointer to a class 1 output buffer in which GS/OS returns the disk volume name (preceded and followed by a pathname separator). The output buffer should be 35 bytes long.

Common error codes:

[none]

Another possible error code is $07.

Programming example:

An application never really needs to know the actual name of the GS/OS boot volume. If it needs to define a pathname on the boot volume, it should use the */ shorthand notation to identify the root directory.

It may be convenient, however, to display the name for information or when debugging. Here is a GS/OS subroutine that shows how to use GetBootVol:

```
Get_Boot    _GetBootVol GBV_Parms
            RTS
```

```
GBV_Parms  ANOP

           DC   I2'1'          ;The number of parameters
           DC   I4'BootSpace'  ;Pointer to output buffer

BootSpace  DC   I2'35'
BootName   DS   33             ;Space for name
```

On exit from the subroutine, the name is stored at BootName, preceded by a length word.

none	GET_BUF $D3
GS/OS	ProDOS 8

Purpose:

To determine the starting address of the 1024-byte I/O buffer an open file uses.

There is no equivalent GS/OS command. GS/OS takes care of all buffer-management operations internally.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (2)
+1	ref_num	I	Reference number for the file
+2 to +3	io_buffer	R	Pointer to I/O buffer

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 2).
ref_num	The reference number ProDOS 8 assigned to the file when it was first opened.
io_buffer	A pointer to the 1024-byte file buffer 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. You might be using a reference number for a file that you've already closed.

Another possible error code is $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 executes, 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
RTS
```

```
PARMTBL   DFB 2          ;The # of parameters
          DFB 2          ;File reference number
BUFFPTR   DS  2          ;Buffer address is returned here
```

GetDevNumber **$2020**		**none**
GS/OS		**ProDOS 8**

Purpose:

To determine the device reference number corresponding to a specified device name or volume name.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (2)
+2 to +5	dev_name	I	Pointer to device/volume name string
+6 to +7	dev_num	R	Device reference number

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 2.
dev_name	A pointer to a class 1 device name string or the class 1 volume name string. A volume name must be preceded by a pathname separator.
dev_num	The device's reference number.

Note: If dev_name points to a volume name, the dev_num GS/OS returns represents the *current* device reference number for the volume. The volume's dev_num will change if the disk is removed and placed in another disk drive.

Common error codes:

$10	The specified device name does not exist.
$40	The volume name specified contains invalid characters or does not start with a valid separator (/ or :).
$45	The disk with the specified volume name can't be found, or the name, although preceded by a separator, is otherwise invalid.

Other possible error codes are $07, $11.

Programming example:

Here is a GS/OS code fragment you can use to determine the device reference number for a disk whose name is /APPLEWORKS.GS:

```
          _GetDevNumber GDN_Parms
          RTS

GDN_Parms ANOP
          DC   I2'2'          ;The number of parameters
          DC   I4'VolName'
          DS   2              ;Device ref number returned here

VolName   GSString '/APPLEWORKS.GS'
```

Don't forget to include a leading slash (or colon) in the volume name.

GetDirEntry $201C	none
GS/OS	ProDOS 8

Purpose:

To read an open directory file. GS/OS returns entries that contain information about the files in a directory.

There is no equivalent ProDOS 8 command. Under ProDOS 8, you must open the directory file, read it into memory, and interpret the data yourself. This requires an understanding of the structure of a directory file. See Chapter 2.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (17)
+2 to +3	ref_num	I	Reference number for the file
+4 to +5	flags	R	Extended file flag
+6 to +7	base	I	Base code
+8 to +9	displacement	I	Displacement code
+10 to +13	name_buffer	I	Pointer to name buffer
+14 to +15	entry_num	R	Absolute directory entry number
+16 to +17	file_type	R	File type code
+18 to +21	eof	R	Size of the file
+22 to +25	block_count	R	Number of blocks file uses
+26 to +33	create_td	R	Time and date of creation
+34 to +41	modify_td	R	Time and date of modification
+42 to +43	access	R	Access code
+44 to +47	aux_type	R	Auxiliary type code
+48 to +49	file_sys_id	R	Operating system ID code
+50 to +53	option_list	R	Pointer to option list
+54 to +57	res_eof	R	Size of the resource fork
+58 to +61	res_block_count	R	Number of blocks resource fork uses

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 5; the maximum is 15.
ref_num	The reference number GS/OS assigned to the directory file when it was first opened.
flags	Bit 15 of this word indicates whether the file represented by the current directory entry is an extended file (bit 15 = 1) or not (bit 15 = 0).
base	This code tells GS/OS how to calculate the number of the next directory entry to read. If base = 0, displacement is an absolute directory entry; if base = 1, GS/OS adds displacement to the current entry number to get the next entry number; if base = 2, GS/OS subtracts displacement from the current entry number to get the next entry number. Note that GS/OS sets the current entry number to 0 when it first opens a file and updates it each time the application calls GetDirEntry.
displacement	If base = 0, this represents the absolute number of the directory entry to be returned. Otherwise, it represents the displacement to the next directory entry to be returned, which can be positive or negative, depending on the value of base. Note that if base and displacement are both zero, GS/OS returns in the entry_num field the total number of active entries in the subdirectory. It also sets the current entry number to the first entry in the subdirectory. To step through the directory one entry at a time, set both base and displacement to 1 and keep calling GetDirEntry until error $61 (end of directory) occurs.
name_buffer	A pointer to a class 1 output buffer in which GS/OS stores the filename it finds in the directory entry. For volumes formatted for the ProDOS file system, the buffer size should be 19 bytes (15 for the name bytes, 2 for the length word, and 2 for the buffer size word). Since GetDirEntry could also be used to read directories of foreign operating systems that use longer filenames (such as Macintosh HFS or CD-ROM High Sierra), you might want to make the buffer even larger. If the output buffer you provide is too small, GetDirEntry returns as much of the name as will fit in the buffer, but returns the actual length.
entry_num	The absolute directory entry number of the current entry.
file_type	A code indicating the type of data the file holds. See Table 2-5 for a description of the ProDOS file type codes.

eof	A value that holds the current EOF position. This value is equal to the size of the file (in bytes). If the file is an extended file, this field relates to the data fork of the file only.
block_count	This field contains the total number of blocks used by the file for data storage and index blocks. If the file is an extended file, this field relates to the data fork of the file only.
create_td	The time and date of creation. These 8 bytes represent the following parameters in the following order:

```
seconds
minutes
hour          in 24-hour military format
year          year minus 1900
day           day of month minus 1
month         0 = January, 1 = February, and so on
[not used]
weekday       1 = Sunday, 2 = Monday, and so on
```

Note: This format is the same as the one used by the ReadTimeHex function in the IIGS's Miscellaneous Tool Set but is different from the one used in a standard file entry for the ProDOS file system.

modify_td	The time and date of last modification. The ordering of these 8 bytes is the same as for create_time.
access	This field contains several 1-bit codes defining the access attributes of the file. See Figure 2-10 for a description of these bits.
aux_type	This is the auxiliary type code. The meaning of the code depends on the file type code and on the program that created the file in the first place. For SYS, BIN, BAS, and VAR files, it is a default loading address; for TXT files, it is a record length; for SRC files, it is an APW language type code.
file_sys_id	The file system identification code. The currently defined values are

$00 = [reserved]
$01 = ProDOS/SOS
$02 = DOS 3.3
$03 = DOS 3.2/3.1
$04 = Apple II Pascal
$05 = Macintosh MFS
$06 = Macintosh HFS
$07 = Macintosh XL (LISA)
$08 = Apple CP/M
$09 = [reserved]
$0A = MS-DOS

$0B = High Sierra (CD-ROM)
$0C = ISO 9660 (CD-ROM)

All other values are reserved.

option_list	A pointer to a class 1 output buffer where GS/OS returns file information unique to the file system translator used to access the file.
res_eof	A value that holds the current EOF position of the resource fork of an extended file. This value is equal to the size of the resource fork of the file (in bytes).
res_block_count	This field contains the total number of blocks used by the resource fork of an extended file for data storage and for index blocks.

Common error codes:

$4F	The name buffer is too small to hold the filename.
$61	End of directory. When you receive this error, close the subdirectory file you opened before calling GetDirEntry.

Other possible error codes are $07, $27, $43, $4A, $4B, $52, $53, $58.

Programming example:

Here is a GS/OS subroutine that displays the names of all the files in a given subdirectory by repeatedly calling GetDirEntry. On entry to the subroutine, the long-word pointer to the subdirectory pathname must be in the A (high word) and X (low word) registers.

```
Catalog   START

          STX     Name_Ptr          ;Set up pointer to pathname
          STA     Name_Ptr+2

          _Open   Open_Prms         ;Open the subdirectory file
          LDA     ref_num
          STA     ref_num1
          STA     ref_num2

Read_Dir  _GetDirEntry GDE_Parms
          BCS     Exit

          LDA     NameBuff+2        ;Put length in high byte
          XBA                       ; so it's just before the
          STA     NameBuff+2        ; filename

          PushPtr NameBuff+3        ;Point to length byte
          _DrawString               ;Display filename

          JSR     CRLF              ;Move to start of next line
          BRA    Read_Dir

Exit      _Close Close_Prms         ;Close subdirectory file
          RTS

Open_Prms  ANOP
          DC   I2'2'                ;The number of parameters
ref_num   DS   2                    ;Reference number
Name_Ptr  DS   4                    ;Pointer to subdir pathname

Close_Prms ANOP
          DC   I2'1'
ref_num1  DS   2

GDE_Parms  ANOP
          DC   I2'5'
ref_num2  DS   2                    ;reference number
          DS   2                    ;flags
          DC   I2'1'                ;Base = "increment"
          DC   I2'1'                ;displacement = +1
          DC   I4'NameBuff'         ;Pointer to name buffer

NameBuff  DC      I2'19'            ;Buffer size
          DS      2                 ;Length
          DS      15                ;Filename

          END
```

Notice that the values for base and displacement are both set to 1 in the GetDirEntry parameter table so that all active entries in the directory will be returned as GetDirEntry is called again and again. The read loop ends when GetDirEntry returns an error. (This will normally be error code $61—"end of directory.")

Also notice the trickery used to set up a standard Pascal-type string for _DrawString to act on. Pascal strings are preceded by a single length byte, but the length in the GetDirEntry name buffer occupies 2 bytes. The low-order length byte is stored at Name_Buff+3 to set up the Pascal-type string. The subroutine assumes that the file name will not exceed 255 characters.

GetEOF $2019	GET_EOF $D1
GS/OS	ProDOS 8

Purpose:

To determine the value of the current end-of-file pointer (EOF) of an open file. This value represents the size of the file.

Parameter table:

ProDOS 8		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0	num_parms	I	Number of parameters (2)
+1	ref_num	I	Reference number for the file
+2 to +4	eof	R	The end-of-file position

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (2)
+2 to +3	ref_num	I	Reference number for the file
+4 to +7	eof	R	The end-of-file position

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 2).
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 2.
ref_num	The reference number ProDOS 8 or GS/OS assigned to the file when it was first opened.
eof	A 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. You might be using a reference number for a file that you've already closed.

Other possible error codes are $04, $07.

Programming example:

Use the GetEOF command to quickly determine how big an open file is. For example, after you call this GS/OS subroutine, the size of open file #1 is stored at Position (low-order bytes first):

```
          _GetEOF GE_Parms
          BCS Error         ;Branch if error occurred
          RTS

GE_Parms  DC   I2'2'        ;The # of parameters
          DC   I2'1'        ;File reference number
Position  DS   4            ;Current EOF position
```

GetFileInfo **$2006**	**GET_FILE_INFO** **$C4**
GS/OS	**ProDOS 8**

Purpose:

To retrieve the information stored in a file's directory entry. This includes the access code, file type code, auxiliary type code, storage type code, the number of blocks the file uses, and the date and time the file was created and last modified.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (10)
+1 to +2	pathname	I	Pointer to the pathname string
+3	access	R	Access code
+4	file_type	R	File type code
+5 to +6	aux_type	R	Auxiliary type code[a]
+7	storage_type	R	Storage type code
+8 to +9	blocks_used	R	Blocks used by the file[a]
+10 to +11	modify_date	R	Modification date
+12 to +13	modify_time	R	Modification time
+14 to +15	create_date	R	Creation date
+16 to +17	create_time	R	Creation time

[a] When pathname points to the name of a volume directory rather than the name of a standard file, the volume size (in blocks) is returned in the aux_type field, and the number of blocks currently in use by all files on the volume is returned in the blocks_used field.

GS/OS Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (12)
+2 to +5	pathname	I	Pointer to the pathname string
+6 to +7	access	R	Access code
+8 to +9	file_type	R	File type code
+10 to +13	aux_type	R	Auxiliary type code
+14 to +15	storage_type	R	Storage type code
+16 to +23	create_td	R	Creation time and date
+24 to +31	modify_td	R	Modification time and date
+32 to +35	option_list	R	Pointer to option list
+36 to +39	eof	R	Size of the file
+40 to +43	blocks_used	R	Blocks used by the file
+44 to +47	resource_eof	R	Size of resource fork
+48 to +51	resource_blocks	R	Blocks used by resource fork

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 10).
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 12.
pathname	A pointer to a class 0 (ProDOS 8) or class 1 (GS/OS) string describing the pathname of the file to be used. If the pathname specified is not preceded by a separator (/ for ProDOS 8; / or : for GS/OS), the operating system appends the name to the default prefix (in GS/OS, this is the 0/ prefix) to create a full pathname.
access	This field contains several 1-bit codes that define the access attributes of the file. See Figure 2-10 for a description of these bits.
file_type	This code indicates the type of data the file holds. See Table 2-5 for a description of the ProDOS file type codes.
aux_type	This is the auxiliary type code. The meaning of the code depends on the file type code and on the program that created the file in the first

	place. For SYS, BIN, BAS, and VAR files, it is a default loading address; for TXT files, it is a record length; for SRC files, it is an APW language type code. Exception: Under ProDOS 8, if the pathname is a volume directory name, aux_type holds the volume size (in blocks).
storage_type	This code describes the physical organization of the file on the disk: $01 = seedling file $02 = sapling file $03 = tree file $04 = Pascal region on a partitioned disk $05 = extended file $0D = directory file (linked list) $0F = volume directory file (linked list)
blocks_used	This field contains the total number of blocks used by the file for data storage and index blocks. (Use GetEOF to determine the number of bytes in a file.) If the file is an extended file, this is the number of blocks used by the data fork only. This field is undefined for a GS/OS subdirectory file. *Exception*: Under ProDOS 8, if the pathname field points to a volume directory name, blocks_used contains the number of blocks in use on the disk by *all* files.
modify_date	This field contains the date (year, month, day) the file was last modified. Figure 8-1 in Chapter 8 shows the format of these bytes.
modify_time	This field contains the time (hour, minute) the file was last modified. Figure 8-1 in Chapter 8 shows the format of these bytes.
create_date	This field contains the date (year, month, day) the file was created. Figure 8-1 in Chapter 8 shows the format of these bytes.
create_time	This field contains the time (hour, minute) the file was created. Figure 8-1 in Chapter 8 shows the format of these bytes.
create_td	The time and date of creation. These eight bytes represent the following parameters in the following order:

```
seconds
minutes
hour          in 24-hour military format
year          year minus 1900
day           day of month minus 1
month         0 = January, 1 = February, and so on
[not used]
weekday       1 = Sunday, 2 = Monday, and so on
```

	Note: This format is the same as the one used by the ReadTimeHex function in the IIGS's Miscellaneous Tool Set, but is different from the one used in a standard directory entry for the ProDOS file system.
modify_td	The time and date of last modification. The ordering of these eight bytes is the same as for create_td.
option_list	A pointer to a class 1 output buffer where GS/OS returns file information unique to the file system translator used to access the file.
eof	The size of the file in bytes. If the file is an extended file, this is the size of the data fork only. This field has no meaning for a subdirectory file.
resource_eof	If the file is an extended file, this is the size of the resource fork.
resource_blocks	If the file is an extended file, this is the number of blocks the resource fork uses on disk.

Common error codes:

$40	The pathname contains invalid characters, or a full pathname was not specified (and no default prefix has been set up). Verify 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 default prefix has been set up.
$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 default 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 default prefix.
$46	The file was not found.

Other possible error codes are $04, $07, $27, $4A, $4B, $52, $53, $58.

Programming example:

See the example given for the SetFileInfo command.

GetFSTInfo **$202B**		**none**
GS/OS		**ProDOS 8**

Purpose:

To get general information about the characteristics of a GS/OS file system translator. There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (8)
+2 to +3	FST_num	I	FST reference number
+4 to +5	file_sys_id	R	File system ID
+6 to +9	FST_name	R	Pointer to FST name
+10 to +11	version	R	FST version number
+12 to +13	attributes	R	FST attributes
+14 to +15	block_size	R	FST block size
+16 to +19	max_vol_size	R	FST volume size
+20 to +23	max_file_size	R	FST file size

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 8.
FST_num	The FST reference number. GS/OS assigns consecutive reference numbers, beginning with 1, to the FSTs it finds in the system.
file_sys_id	The identification code for the file system that the FST supports:

$01 = ProDOS/SOS
$02 = DOS 3.3
$03 = DOS 3.2/3.1
$04 = Apple II Pascal
$05 = Macintosh MFS
$06 = Macintosh HFS
$07 = Macintosh XL (LISA)

$08 =	Apple CP/M
$09 =	Character FST
$0A =	MS-DOS
$0B =	High Sierra (CD-ROM)
$0C =	ISO 9660 (CD-ROM)

FST_name	A pointer to class 1 output buffer where GS/OS returns the name of the file system translator.
version	The version number of the file system translator:

```
bit 15       1 = prototype version
             0 = final version
bits 14-8    major version number
bits 7-0     minor version number
```

attributes	The attributes of the file system translator:

```
bit 15    1 = FST wants filenames in uppercase
bit 14    1 = character FST; 0 = block FST
bit 12    1 = FST wants the characters in
              filenames to have the high-order
              bit clear
```

block_size	The size (in bytes) of the blocks the FST handles.
max_vol_size	The maximum size (in blocks) of the disk volumes the FST handles.
max_file_size	The maximum size (in bytes) of the files the FST handles.

Common error codes:

$53	Parameter out of range. GS/OS returns this error if the FST reference number does not exist.

Another possible error code is $07.

Comments:

GS/OS provides no simple way to determine how many FSTs are active. To get information on all FSTs, keep calling GetFSTInfo with successively higher FST_num values (beginning at 1) until GS/OS returns an error code of $53.

GetLevel $201B	none
GS/OS	ProDOS 8

Purpose:

To determine the value of the system file level.

• There is no equivalent ProDOS 8 command. The system file level is stored at LEVEL ($BF94) in the ProDOS 8 system global page.

Parameter table:

GS/OS Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +3	level	R	The system file level

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
level	The value of the system file level. The values that can be returned range from $0000 to $00FF.

Common error codes:

[none]

Another possible error code is $07.

Programming example:

Here is a GS/OS subroutine for returning the system file level number:

```
          _GetLevel GL_Parms
          RTS

GL_Parms  ANOP
          DC   I2'1'     ;The number of parameters
theLevel  DS   2         ;System file level returned here
```

GetMark $2017	GET_MARK $CF
GS/OS	ProDOS 8

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:

ProDOS 8 Offset	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (2)
+1	ref_num	I	Reference number for the file
+2 to +4	position	R	The current Mark position

GS/OS Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (2)
+2 to +3	ref_num	I	Reference number for the file
+4 to +7	position	R	The current Mark position

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 2).
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 2.
ref_num	The reference number ProDOS 8 or GS/OS assigned to the file when it was first opened.
position	The current Mark position in bytes.

Common error codes:

$43	The file reference number is invalid. You might be using a reference number for a file that you've already closed.

Other possible error codes are $04, $07.

Programming example:

Here is a ProDOS 8 subroutine that reads and displays 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.

GetName $2027	none
GS/OS	ProDOS 8

Purpose:

To determine the name of the application currently running.

There is no equivalent ProDOS 8 command. Under ProDOS 8, you can deduce the name by examining the pathname or partial pathname stored at $280 when the application starts up.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +5	data_buffer	R	Pointer to application name string

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
data_buffer	A pointer to a class 1 output buffer where the name of the current application is to be returned. The name is an ASCII string preceded by a length word. The output buffer should be 35 bytes long to accommodate the longest filename you might encounter. (Macintosh filenames can be up to 31 characters long.)

Common error codes:

[none]

Other possible error codes are $07, $4F.

Programming example:

A running application sometimes needs to be able to determine what its name is. It would need to know this, for example, if it had to transfer a copy of itself to a RAMdisk when it was started up. The application shouldn't assume a specific name because the user may have renamed the application.

Here is how to determine the name of the application:

```
_GetName GN_Parms
RTS
```

```
GN_Parms  ANOP
          DC   I1'1'          ;The number of parameters
          DC   I4'NameSpace' ;Pointer to class 1 buffer

NameSpace DC   I2'35'         ;Size of buffer
TheName   DS   33             ;Space for class 1 name string
```

GetName returns the filename only, preceded by a length word. The subdirectory it resides in is given by the 1/ prefix, provided the application, or a desk accessory, hasn't changed it since the application was launched. Use GetPrefix to determine the specific value of this prefix.

GetPrefix $200A	GET_PREFIX $C7
GS/OS	ProDOS 8

Purpose:

To determine the name of the default prefix (ProDOS 8) or any of the 32 GS/OS prefixes (0/ through 31/).

Parameter table:

ProDOS 8		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0	num_parms	I	Number of parameters (1)
+1 to +2	prefix	R	Pointer to prefix name string

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (2)
+2 to +3	prefix_num	I	Prefix number (0 to 31)
+4 to +7	prefix	R	Pointer to prefix name string

Note: The GS/OS GetPrefix command uses the colon as a separator character in the prefix strings which it returns. In addition, if the prefix name used with SetPrefix contained lowercase characters, GetPrefix does not convert them to uppercase (but the ProDOS 8 GET_PREFIX command does).

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 1).
prefix	A pointer to a class 0 (ProDOS 8) or class 1 (GS/OS) output buffer in which the operating system returns the prefix name. The name is in ASCII and is preceded and followed by a pathname separator character (/ for ProDOS 8; / or : for GS/OS).
	For ProDOS 8, the buffer must be 67 bytes long to accommodate the longest possible prefix that might be active (64 characters) plus the preceding length byte and the two separator characters.

For GS/OS, a pathname can be up to 8K in size, but it is rare to encounter any longer than 67 characters. You should set the class 1 buffer length word to 69 when you call GetPrefix; if the buffer isn't big enough, GS/OS returns error code $4F, and you can call the command again using the length word returned after the buffer size length word.

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 2.
prefix_num	The GS/OS prefix number (0 to 31). This is a binary number, not an ASCII number string followed by a slash.

Common error codes:

$56	The pathname buffer address is invalid because it has been marked as in use in the ProDOS 8 system bit map. Specify a buffer address that does not conflict with areas already used by ProDOS 8 or its file buffers. Examine the system bit map to determine the free and protected areas.

Other possible error codes are $04, $07, $4F, $53.

Programming example:

This GS/OS subroutine gets the 7/ prefix and stores it in the buffer beginning at PathName (preceded by a length word):

```
          _GetPrefix GP_Parms
          BCS Error           ;Branch if error occurred
          RTS

GP_Parms  DC   I2'2'
          DC   I2'7'          ;Get prefix 7/
          DC   I4'PathBuff'

PathBuff  DC   I2'69'         ;Size of buffer
PathName  DS   67
```

Note that if a 7/ prefix has not yet been set up (with SetPrefix), the prefix length word returned by GetPrefix will be zero.

GetSysPrefs $200F	none
GS/OS	ProDOS 8

Purpose:

To determine the state of the system preferences status word.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +3	preferences	I	System preferences

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
preferences	The system preferences status word:

```
bit 15    1 = display mount volume dialog
          0 = don't display the dialog
```

Common error codes:

[none]

Comments:

GS/OS commands that have pathnames as input parameters normally display a mount volume dialog box (to ask the user to insert a specified disk volume) if the commands can't find the volume they are expecting. If the application wants to handle "volume not found" errors itself, it can use SetSysPrefs to clear bit 15 of the preferences word.

none	GET_TIME $82
GS/OS	ProDOS 8

Purpose:

To read the date and time from the system clock into the ProDOS 8 system global page at DATE ($BF90-$BF91) and TIME ($BF92-$BF93).

There is no equivalent GS/OS command. Use the ReadAsciiTime and ReadTimeHex functions in the IIGS's Miscellaneous Tool Set instead. See Chapter 8.

Parameter table:

[no parameter table, but the caller must point to a dummy table]

Common error codes:

[none]

Programming example:

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

Note, however, that GET_TIME returns the time only if a ProDOS-compatible clock, like the built-in IIGS clock, Thunderware Thunderclock, Prometheus Versacard, or Applied Engineering Timemaster H.O., is installed. When ProDOS 8 first starts up, it installs a special clock driver for reading these types of cards. (We see how to install custom 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
```

Notice the use of a dummy parameter table pointer of $0000.

GetVersion $202A	none
GS/OS	ProDOS 8

Purpose:

To return the GS/OS version number.

There is no equivalent ProDOS 8 command. Under ProDOS 8, the minor release number is stored at $BFFF in the ProDOS 8 system global page. The major release number is always 1.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +3	version	R	GS/OS version number

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
version	The version of the GS/OS operating system currently in use. The low-order byte contains the minor release number, and the high-order byte contains the major release number. (This means, for example, that version 2.1 would be represented by $0201.) Bit 7 of the high-order byte is 1 if the release is a prototype (beta) version.

Common error codes:

[none]

Another possible error code is $07.

Programming example:

Here is a subroutine that will print out the GS/OS version number in ASCII in the current desktop window:

```
Show_Vers  START

           _GetVersion GV_Parms
           LDA     Version          ;Get version word

           PHA                      ;(Save two copies on stack)
```

```
          PHA

          XBA                      ;Swap high/low
          AND     #$007F           ;Isolate major version #
          ORA     #$0030           ;Convert to ASCII
          PHA
          _DrawChar
          PushWord #$2E            ;Period (.)
          _DrawChar

          PLA                      ;Get version word back
          AND     #$00FF           ;Isolate minor version #
          ORA     #$0030           ;Convert to ASCII
          PHA
          _DrawChar

          PLA                      ;Get version word back
          BPL     Exit             ;Branch if prototype bit not 1

          PushWord #$70            ;'p' for prototype
          _DrawChar

Exit      RTS

GV_Parms  ANOP
          DC   I1'1'               ;The number of parameters
Version   DS   2                   ;Version word returned here

          END
```

This subroutine works only if the major and minor version numbers are less than 10.

NewLine $2011	NEWLINE $C9
GS/OS	ProDOS 8

Purpose:

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

Parameter table:

ProDOS 8

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (3)
+1	ref_num	I	Reference number for the file
+2	enable_mask	I	Newline enable mask
+3	newline_char	I	Newline character

GS/OS

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (4)
+2 to +3	ref_num	I	Reference number for the file
+4 to +5	enable_mask	I	Newline enable mask
+6 to +7	num_chars	I	Number of characters in table
+8 to +11	newline_table	I	Pointer to newline character table

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 3).
ref_num	The reference number ProDOS 8 or GS/OS assigned to the file when it was first 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_char (or, for GS/OS, any of the characters in newline_table), the read request terminates; otherwise, the read continues normally. Exception: If enable_mask is zero, newline read mode is disabled, and read operations are not affected.
newline_char	The value of the newline character. Read requests automatically terminate if the logical AND of enable_mask and the character being read equals newline_char.
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 3; the maximum is 3.
num_chars	The number of characters in the newline character table. If enable_mask is not zero, num_chars cannot be zero.
newline_table	A pointer to a table of active GS/OS newline characters. Each character occupies one byte in the table and the table can be up to 256 bytes long.

Common error codes:

$43	The file reference number is invalid. You might be using a reference number for a file that you've already closed.

Other possible error codes are $04, $07.

Programming example:

A common situation is one where you want to read one line at a time from a textfile. Since each line in a standard ProDOS textfile is terminated by $0D, the ASCII code for the carriage return character, you could simply set enable_mask equal to $FF and the newline character to $0D before executing the Newline command. But some applications may use the negative ASCII code for the carriage return character ($8D) for an end-of-line character. If you want to terminate a read operation for either $0D or $8D, use a newline character of $0D and set the enable_mask to $7F.

Here is a GS/OS subroutine that sets the $0D/$8D newline read mode for you:

```
         _NewLine NL_Parms
         BCS Error
         RTL

PARMTBL  DC   I2'4'        ;4 parameters
         DC   I2'1'        ;File reference number (#1 assumed)
         DC   I2'$7F'      ;enable_mask
         DC   I2'1'        ;Number of newline characters
         DC   I4'NL_Table' ;Pointer to table

NL_Table DC   I1'$0D'      ;Carriage return
```

Null $200D	none
GS/OS	ProDOS 8

Purpose:

To execute pending events in the GS/OS signal queue and the Scheduler's task queue. There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (0)

Meanings of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 0; the maximum is 0.

Common error codes:

[none]

Comments:

As explained in Chapter 6, some interrupt handlers place events in the GS/OS signal queue to ensure that they are dealt with when the system isn't busy. They can also place tasks into the Scheduler tool set's task queue if they wish.

The events in the signal and task queues are normally processed when a GS/OS command ends or, if no GS/OS commands are being used, every sixtieth of a second, in response to a task triggered by a vertical blanking interrupt.

If your application isn't making GS/OS commands for extended periods, and interrupts are disabled, it should call the Null command periodically so that signal queue and task queue events may be processed.

none	ON_LINE $C5
GS/OS	ProDOS 8

Purpose:

To determine the volume directory name of a specific disk or the names of all active ProDOS 8 volumes.

Under GS/OS, use the Volume command instead.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (2)
+1	unit_num	I	Unit number
+2 to +3	data_buffer	I	Pointer to data buffer

Descriptions of parameters:

num_parms The number of parameters in the ProDOS 8 parameter table (always 2).

unit_num The slot and drive number for the disk drive to be accessed. The format of this byte is as follows:

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

ProDOS 8 assigns a drive number of 1 or 2 to each drive in the system. DR = 0 for drive 1, and DR = 1 for drive 2. SLOT is usually the actual slot number for the disk controller card (1–7 decimal; 001–111 binary) but may be the number of a phantom, or logical, slot.

The unit_num value for the /RAM volume is $B0, meaning it is the logical slot 3, drive 2 device.

Exception: If unit_num is 0, the volume names of all drives are returned.

data_buffer A pointer to a buffer containing the volume name information for the specified drive. If unit_num is 0, the volume names of all drives are returned. Each volume name entry is 16 bytes long.

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

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

DR and SLOT are defined in the same way as unit_num. Name length contains the length of the volume name for the device defined by DR and SLOT. (If name length is zero, an error occurred; in this case, the error code is stored in the next byte. If the error code is $57 ("duplicate volume"), the third byte of the record contains the unit_num 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_num is 0, the record after the last valid 16-byte record begins with a $00 byte. You must reserve a 256-byte buffer area if you call ON_LINE with unit_num set to 0.

Common error codes:

$27	The disk is unreadable probably because a portion of the disk medium is permanently damaged. This error also occurs if the drive door on a 5.25-inch drive is open or no disk is in the drive.
$28	No device connected. ProDOS 8 returns this error if you do not have a second 5.25-inch drive connected to the drive controller, but you try to access it.
$2E	A disk with an open file was removed from its drive before executing the command. Solution: Close all files on the disk to be removed before executing the ON_LINE command.
$2F	Device not on line. ProDOS 8 returns this error if no disk is in a 3.5-inch drive.
$52	The disk in the drive specified by unit_num is not a ProDOS-formatted disk. Solution: Use only ProDOS-formatted disks with ProDOS 8!
$56	The pathname buffer address is invalid because it has been marked as in use in the ProDOS 8 system bit map. Specify a buffer address that does not conflict with areas already used by ProDOS 8 or its file buffers. Examine the system bit map to determine the free and protected areas.

Other possible error codes are $04, $55.

ON_LINE handles error conditions quite differently from how the other MLI commands do. Generally, 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.

Programming example:

This ProDOS 8 program reads 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_num = 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, 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 $2010	OPEN $C8
GS/OS	ProDOS 8

Purpose:

To prepare a file for subsequent read and write operations. When you open a file, the position-in-file pointer (Mark) points to the start of the file (that is, Mark = 0), and its file level is set equal to the system file level. Under GS/OS open also returns all the file's directory attributes.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (3)
+1 to +2	pathname	I	Pointer to the pathname string
+3 to +4	io_buffer	R	Pointer to I/O buffer
+5	ref_num	R	Reference number for the file

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (15)
+2 to +3	ref_num	R	Reference number for the file
+4 to +7	pathname	I	Pointer to the pathname string
+8 to +9	request_access	I	Access permissions requested
+10 to +11	resource_num	I	Fork designator
+12 to +13	access	R	Access code
+14 to +15	file_type	R	File type code
+16 to +19	aux_type	R	Auxiliary type code
+20 to +21	storage_type	R	Storage type code

+22 to +29	create_td	R	Creation time and date
+30 to +37	modify_td	R	Modification time and date
+38 to +41	option_list	R	Pointer to option list
+42 to +45	eof	R	Size of the file
+46 to +49	blocks_used	R	Blocks used by the file
+50 to +53	resource_eof	R	Size of resource fork
+54 to +57	resource_blocks	R	Blocks used by resource fork

Important: You can usually open a closed file only. But, if a file is open, and its write-enabled access code bit is not set (that is, you aren't allowed to write to it), it may be opened more than once simultaneously.

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 3).
pathname	A pointer to a class 0 (ProDOS 8) or class 1 (GS/OS) string describing the pathname of the file to be used. If the pathname specified is not preceded by a separator (/ for ProDOS 8; / or : for GS/OS), the operating system appends the name to the default prefix (in GS/OS, this is the 0/ prefix) to create a full pathname.
io_buffer	A pointer to a 1024-byte file buffer that the open file can use. 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.
ref_num	The reference number ProDOS 8 or GS/OS assigns to the file. All file operations on open files use this reference number (instead of a pathname) to identify the file. The file level is set to the value of the system file level. (For ProDOS 8, this value is stored at $BF94. For GS/OS, use GetLevel and SetLevel to read and set the system file level.)
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 15. If the file is for a character device, the maximum value is only 3.
request_access	This word describes the requested access permission:

```
bit 1    1 = request write access
bit 0    1 = request read access
```

	You cannot request write access for files on a CD-ROM drive. If this word is $0000, the access granted is the same as allowed by the access_code word.
resource_num	If the file is an extended file, this word tells GS/OS which fork to open:

```
$0000    open data fork
$0001    open resource fork
```

Note: The rest of the parameters in the GS/OS parameter list are the same as those returned by the GetFileInfo command.

Common error codes:

$40	The pathname contains invalid characters, or a full pathname was not specified (and no default prefix has been set up). Verify 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 default prefix has been set up.
$42	An attempt was made to open a ninth file. ProDOS 8 allows only eight open files.
$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 default 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 default prefix.
$46	The file was not found.
$50	The file is open. You can open only files that are closed unless the file is not write-enabled.
$56	The pathname buffer address is invalid because it has been marked as in use in the ProDOS 8 system bit map. Specify a buffer address that does not conflict with areas already used by ProDOS 8 or its file buffers. Examine the system bit map to determine the free and protected areas.

Other possible error codes are $04, $07, $27, $4A, $4B, $52.

Programming example:

The following GS/OS subroutine opens a file called SESAME that resides in the subdirectory identified by 0/:

```
         _Open OP_Parms
         BCS   Error        ;Branch if error occurred
         RTS

OP_Parms DC    I2'2'        ;Only need 2 parameters
         DS    2            ;ref_num returned here
         DC    I4'PathName' ;Pointer to pathname

PathName GSString 'SESAME'     ;Filename
```

GS/OS returns an error code of $46 if the file you try to open does not yet exist.

Once you open a file, you should take the reference number Open returns and store it in the parameter tables of other GS/OS commands which you might use to access the file while it is open.

OSShutdown $2003	none
GS/OS	ProDOS 8

Purpose:

To shut down GS/OS prior to a cold reboot or power down operation.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	 *Symbolic Name*	*Input* *or* *Result*	 *Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +5	shutdown_flag	I	Pointer to next pathname

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
shutdown_flag	The two low-order bits in this flag control the mechanics of the shutdown operation:

```
bit 0 : 1 = GS/OS shuts down and system is rebooted
        0 = GS/OS shuts down and the user is asked
            to either reboot or power down

bit 1 : 1 = RAM disk is left intact upon reboot
        0 = RAM disk is initialized upon reboot
```

Common error codes:

[none]

Comments:

When GS/OS shuts down it writes to disk any blocks in the disk cache, closes all new desk accessories, shuts down the Desk Manager, then disposes of all device drivers and file system translators. The OSShutdown command should be used by program selectors like the Finder, not applications.

Quit $2029	QUIT $65
GS/OS	ProDOS 8

Purpose:

To terminate the current application. Under ProDOS 8, control passes to the ProDOS 8 selector program or, if GS/OS was the boot operating system, to a system program (ProDOS 8 or GS/OS) the application specifies. (The standard selector program asks the user to enter the prefix and pathname of the next ProDOS 8 system program to run.) Under GS/OS, the application can pass control to another system program (ProDOS 8 or GS/OS) or return control to the application that called it (typically the Finder).

Parameter table:

ProDOS 8

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (4)
+1	quit_type	I	Quit type code
+2 to +3	pathname	I	Pointer to next pathname
+4	[reserved]	I	Reserved area
+5 to +6	[reserved]	I	Reserved area

GS/OS

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (2)
+2 to +5	pathname	I	Pointer to next pathname
+6 to +7	flags	I	Return/Restart flags

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 4).
quit_type	The ProDOS 8 quit type code. The only quit types currently defined are $00 (standard quit) and $EE (quit to system program). Type $EE may be used only if the system was first booted up under GS/OS.

pathname	A pointer to the class 0 (ProDOS 8) or class 1 (GS/OS) pathname of the next system program to run. The file type code of the program must be $FF (ProDOS 8 system) or $B3 (GS/OS system). *Note*: The pathname cannot reside in page 2 of memory since the QUIT command handler uses this area. For ProDOS 8, this field must be zero if quit_type is $00.
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 0; the maximum is 2.
flags	The Quit option flags; only bits 15 and 14 are significant. If bit 15 is 1, the program's UserID is to be placed on the Quit Return Stack so that the program can be restarted later. If bit 14 is 1, the program is capable of being restarted from memory.

Note: The reserved areas in the ProDOS 8 parameter table must be zeroed before calling the QUIT command.

Common error codes:

$46	The file with the specified pathname was not found.
$5C	The file with the specified pathname is not an executable program. The pathname must be a ProDOS 8 system program (file type $FF) or a GS/OS system program (file type $B3).
$5D	The specified pathname represents a ProDOS 8 system program, but the P8 system file (which contains the ProDOS 8 operating system) is not present in the SYSTEM/ subdirectory of the GS/OS boot disk.
$5F	The Quit Return Stack has overflowed. This stack can hold only 16 entries.

Other possible error codes are $04, $07, $40, $5E.

Programming example:

All well-designed system programs use QUIT to exit so that control can pass to another system program. Here is the usual calling sequence from a ProDOS 8 application:

```
          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          ;Quit type code
```

```
DA  $0000
DFB 0
DA  $0000
```

When you execute a QUIT command with a quit_type of $00, ProDOS 8 moves the code residing at $D100–$D3FF in the second 4K bank of bank-switched RAM (called the selector code or dispatcher code) to location $1000 in main memory and then executes a JMP $1000 instruction.

When the standard ProDOS 8 selector (the one defined inside the PRODOS file) 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 8 system bit map and marks as in use zero page, the stack (page 1), the video RAM area (pages 4–7), and the ProDOS 8 global page (page $BF).
- It enables the 40-column screen and connects the standard input (keyboard) and output (video) subroutines. (You can do this in your own selector program by executing the following group of instructions:

  ```
  LDA $C082       ;Read-enable monitor ROM
  STA $C000       ;Turn off 80STORE
  STA $C00E       ;Turn off alternate char. set
  STA $C00C       ;Turn off 80 columns
  JSR SETNORM     ;$FE84: normal-video characters
  JSR INIT        ;$FB2F: full-screen text mode
  JSR SETKBD      ;$FE89: connect keyboard
  JSR SETVID      ;$FE93: connect 40-column video
  ```

  The writes to the $C000, $C00E, and $C00C soft switches don't do anything on an Apple II Plus but are required for a IIe, IIc, or IIGS to ensure the system switches to standard 40-column mode. Note that the Monitor ROM must be read-enabled before calling the SETNORM, INIT, SETKBD, and SETVID subroutines because it will have been disabled when the selector first 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 ProDOS 8 selector code if you wish. If you do, it must begin with a CLD instruction and it must perform the steps indicated above.

Table 4-5 shows an alternative selector program that follows the above steps. To install the new selector at $D100 (bank2), BRUN the program file from disk. This selector is not interactive since it always passes control to the same system program:

Table 4-5 A ProDOS 8 selector program

```
                **************************************
                *       ProDOS Selector Program      *
                *                                    *
                * When this selector is called using *
                * the QUIT ($65) command, the system *
                * file called BASIC.SYSTEM on the    *
                * boot volume (given by SLOT) will   *
                * be automatically executed.         *
                *                                    *
                * Copyright 1985-1988 Gary B. Little *
                *                                    *
                * Last modified: August 26, 1988     *
                *                                    *
                **************************************

                SLOT     EQU   6          ;Slot number of boot volume

                PATHNAME EQU   $280       ;Full pathname stored here

                FILEBUFF EQU   $1100      ;1K file buffer

                SYS_LOAD EQU   $2000      ;Start addr of system program

                MLI      EQU   $BF00      ;Gateway to MLI
                BITMAP   EQU   $BF58      ;System bit map

                * Soft switches for IIe, IIc, IIGS:

                XSTORE80 EQU   $C000      ;Enable normal page2 switching
                COL80OFF EQU   $C00C      ;Turn off 80-column hardware
                XALTCHAR EQU   $C00E      ;Disable MouseText characters

                SSPACE   EQU   $D100      ;Selector space (in bank2)

                INIT     EQU   $FB2F      ;Set full-screen text mode
                HOME     EQU   $FC58      ;Clear the screen
                SETNORM  EQU   $FE84      ;Set normal video
                SETKBD   EQU   $FE89      ;Connect keyboard driver
                SETVID   EQU   $FE93      ;Connect video driver

                         ORG   $2000

                * Store selector code at $D100 in bank2 of
                * bank-switched RAM:

2000: AD 81 C0           LDA   $C081
2003: AD 81 C0           LDA   $C081      ;Write-enable bank2 BSR

```

Table 4-5 Continued

```
2006: A2 00             LDX   #0
2008: BD 15 20 MOVECODE LDA   SELECTOR,X ;Move the new code
200B: 9D 00 D1          STA   SSPACE,X   ; to its proper place
200E: E8                INX
200F: D0 F7             BNE   MOVECODE

2011: AD 82 C0          LDA   $C082      ;Write-protect BSR

2014: 60                RTS

               SELECTOR EQU *

               * Here is the actual selector code:

                        ORG   $1000

1000: D8                CLD              ;(Required by ProDOS)

               * Get into plain vanilla 40-column mode:

1001: AD 82 C0          LDA   $C082      ;Read-enable monitor ROM

1004: 8D 0C C0          STA   COL80OFF   ;40-column screen
1007: 8D 0E C0          STA   XALTCHAR   ;No MouseText
100A: 8D 00 C0          STA   XSTORE80   ;Normal page2 switching

100D: 20 84 FE          JSR   SETNORM    ;Normal video
1010: 20 2F FB          JSR   INIT       ;Full text window
1013: 20 93 FE          JSR   SETVID     ;Standard video output
1016: 20 89 FE          JSR   SETKBD     ;Standard keyboard input
1019: 20 58 FC          JSR   HOME       ;Clear the screen

101C: 20 00 BF          JSR   MLI
101F: C6                DFB   $C6        ;Set a null prefix
1020: BE 10             DA    PFX_PRMS

1022: 20 00 BF          JSR   MLI
1025: C5                DFB   $C5        ;ONLINE for the boot volume
1026: 9A 10             DA    OL_PARMS
1028: B0 38             BCS   ERROR

102A: AD 9E 10          LDA   NAME_LEN   ;Get returned length
102D: 29 0F             AND   #$0F       ;Strip slot, drive bits
102F: F0 31             BEQ   ERROR      ;If zero, then error
1031: 8D 9E 10          STA   NAME_LEN   ;Store length

               * Put prefix at $281:

1034: A9 2F             LDA   #'/        ;Start prefix with slash
```

Table 4-5 Continued

```
1036: 8D 81 02             STA   PATHNAME+1

1039: A2 00                LDX   #0
103B: BD 9F 10   PUTNAME   LDA   VOL_NAME,X
103E: 9D 82 02             STA   PATHNAME+2,X
1041: E8                   INX
1042: EC 9E 10             CPX   NAME_LEN
1045: D0 F4                BNE   PUTNAME

1047: A9 2F                LDA   #'/         ;End prefix with slash
1049: 9D 82 02             STA   PATHNAME+2,X
104C: E8                   INX

                 * ... and then tack on the BASIC.SYSTEM filename:

104D: A0 00                LDY   #0
104F: B9 C3 10   PUTSYS    LDA   SYS_NAME,Y
1052: F0 07                BEQ   SAVELEN     ;Done if zero
1054: 9D 82 02             STA   PATHNAME+2,X
1057: E8                   INX
1058: C8                   INY
1059: D0 F4                BNE   PUTSYS      ;(Always taken)

105B: E8         SAVELEN   INX               ;Add 1 for initial slash
105C: 8E 80 02             STX   PATHNAME    ;Store length before pathname

105F: 4C 65 10             JMP   RUN_SYS

1062: 4C 62 10   ERROR     JMP   ERROR

1065: 20 00 BF   RUN_SYS   JSR   MLI
1068: C8                   DFB   $C8         ;Open system file
1069: AE 10                DA    OP_PARMS
106B: B0 F5                BCS   ERROR

106D: AD B3 10             LDA   REFNUM
1070: 8D B5 10             STA   REFNUM1     ;Store ref # in READ table

1073: 20 00 BF             JSR   MLI
1076: CA                   DFB   $CA         ;Read system file
1077: B4 10                DA    RD_PARMS
1079: B0 E7                BCS   ERROR

107B: 20 00 BF             JSR   MLI
107E: CC                   DFB   $CC         ;Close system file
107F: BC 10                DA    CL_PARMS
1081: B0 DF                BCS   ERROR

                 * Initialize the system bit map:
```

Table 4-5 Continued

```

1083: A9 CF               LDA  #$CF       ;Pages 0,1,4..7 in use
1085: 8D 58 BF            STA  BITMAP

1088: A9 00               LDA  #0
108A: A2 16               LDX  #22
108C: 9D 58 BF   INITMAP  STA  BITMAP,X   ;Pages 8..$BE free
108F: CA                  DEX
1090: D0 FA               BNE  INITMAP

1092: A9 01               LDA  #1         ;Page $BF in use
1094: 8D 6F BF            STA  BITMAP+23

1097: 4C 00 20            JMP  SYS_LOAD   ;Execute system file

                 * ONLINE parameter table:

109A: 02         OL_PARMS DFB  2
109B: 60                  DFB  SLOT*16    ;Boot slot * 16
109C: 9E 10               DA   NAME_LEN   ;Pointer to len+name

109E: 00         NAME_LEN DS   1          ;Length (bits 0..3)
109F: 00 00 00   VOL_NAME DS   15         ;Volume name
10A2: 00 00 00 00 00 00 00 00
10AA: 00 00 00 00

                 * OPEN parameter table:

10AE: 03         OP_PARMS DFB  3
10AF: 80 02               DA   PATHNAME   ;Pointer to pathname
10B1: 00 11               DA   FILEBUFF
10B3: 00         REFNUM   DS   1          ;File reference number

                 * READ parameter table:

10B4: 04         RD_PARMS DFB  4
10B5: 00         REFNUM1  DS   1
10B6: 00 20               DA   SYS_LOAD   ;Start of load buffer
10B8: FF FF               DW   $FFFF      ;(Enough for entire file)
10BA: 00 00               DW   $0000

                 * CLOSE parameter table:

10BC: 01         CL_PARMS DFB  1
10BD: 00                  DFB  0          ;All files

10BE: 01         PFX_PRMS DFB  1
10BF: C1 10               DA   PFX_NAME

```

Table 4-5 Continued

```
10C1: 01        195  PFX_NAME DFB   1
10C2: 2F        196           ASC   '/'
                197
10C3: 42 41 53  198  SYS_NAME ASC   'BASIC.SYSTEM' ;Name of system program
10C6: 49 43 2E 53 59 53 54 45
10CE: 4D
10CF: 00        199           DFB   0             ;... followed by zero
```

BASIC.SYSTEM in the volume directory of the slot 6, drive 1 disk device. However, this is the program that many users of ProDOS 8 will always want to select after leaving other system programs. From BASIC.SYSTEM, you can use the - (dash) command to execute any other system program.

In certain situations, your selector code may be permitted to pass the name of a file to the system program it selects so that the system program can work with it when it first starts up. For example, you can pass the name of an Applesoft program to BASIC.SYSTEM, and BASIC.SYSTEM will run it as soon as its starts up. (If the selector does not pass a name, BASIC.SYSTEM runs the STARTUP program.) For the system program to accept a filename, it must adhere to a special auto-run protocol that we look at in Chapter 5.

If you are using a IIGS and you ran the ProDOS 8 application after booting GS/OS, you can take advantage of quit_type $EE to pass control from a ProDOS 8 application directly to a ProDOS 8 or GS/OS system program without going through the selector code. All you have to do is place a pointer to the program's pathname in the QUIT parameter list. These programs have file type codes of $FF (SYS) and $B3 (S16), respectively. GS/OS was the bootup operating system if value at location $E100BD is $01.

GS/OS considerations:

Under GS/OS, an application can use the Quit command to either pass control to a specific system program (ProDOS 8 or GS/OS) or return control to the system program whose UserID is on the top of a Quit Return Stack. (GS/OS assigns a unique UserID to a system program when it starts up the program.)

The Quit Return Stack is where an application places its UserID if it wishes to regain control the next time an application quits without specifying the pathname of the next application to run. The availability of a Quit Return Stack makes it easy for a supervisory program to execute subsidiary programs so that control always eventually returns to the original program. In fact, the IIGS Launcher or Finder always pushes its UserID on the Quit Return Stack before launching an application. If it did not, you would not return to it when an application ended.

If the pathname pointer is 0, and the Quit Return Stack is not empty, GS/OS pulls a UserID from the Quit Return Stack and executes the program with that ID. If the Quit Return Stack is empty, GS/OS executes the program launched when the system was booted.

Only the two high-order bits of the flags parameter are significant. If bit 15 is 1, GS/OS places the current application's UserID on the Quit Return Stack before passing control to the application described by the pathname pointer. This means control eventually will return to the current application as later programs quit with a 0 pathname parameter. If bit 15 is 0, nothing is placed on the Quit Return Stack.

If bit 14 of the flags is 1, the calling program is capable of being restarted without being reloaded from disk. (Programs that initialize all their data areas when they start up should be restartable.) If control returns to the calling program, the program will not be loaded from disk unless it has been purged from memory by the IIGS Memory Manager.

Read $2012	READ $CA
GS/OS	ProDOS 8

Purpose:

To read bytes of data from an open file beginning at the current Mark position. After the read operation, the operating system increases Mark by the number of bytes read from the file. The read operation ends when the specified number of bytes have been transferred, when a newline character is encountered, or when the end of the file has been reached.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (4)
+1	ref_num	I	Reference number for file
+2 to +3	data_buffer	I	Pointer to start of data buffer
+4 to +5	request_count	I	Number of bytes to read
+6 to +7	transfer_count	R	Number of bytes actually read

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (5)
+2 to +3	ref_num	I	Reference number for file
+4 to +7	data_buffer	I	Pointer to start of data buffer
+8 to +11	request_count	I	Number of bytes to read
+12 to +15	transfer_count	R	Number of bytes actually read
+16 to +17	cache_priority	I	Block caching priority level

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 4).
ref_num	The reference number ProDOS 8 or GS/OS assigned to the file when it was first opened.
data_buffer	A pointer to the beginning of a block of memory into which file data is to be read. The size of the buffer must be request_count characters.
request_count	The number of characters to be read from the file and placed in the buffer pointed to by data_buffer.
transfer_count	The number of characters actually read from the file. It usually equals request_count, but it will be less if the operating system reaches the end of the file or if newline read mode is active and a newline character is read. See the discussion of the NewLine command.
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 4; the maximum is 5 (or 4 if the file is a character file).
cache_priority	This code indicates how GS/OS is to handle the caching of disk blocks related to the read operation:

```
$0000    do not cache blocks
$0001    cache blocks
```

This field is not used for character devices.

Common error codes:

$43	The file reference number is invalid. You might be using a reference number for a file that you've already closed.
$4C	The end-of-file position has been reached. Solution: Stop reading from the file. Note that ProDOS 8 or GS/OS flags this error only if no bytes were read from the file. (That is, transfer_count 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 pathname buffer address is invalid because it has been marked as in use in the ProDOS 8 system bit map. Specify a buffer address that does not conflict with areas already used by ProDOS 8 or its file buffers. Examine the system bit map to determine the free and protected areas.

Other possible error codes are $04, $07, $27.

Programming example:

The following GS/OS subroutine reads 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.

```
         _Read RD_Parms
         BCS   Error      ;Branch if error occurred
         RTS

RD_Parms DC    I2'4'      ;Parameter count
         DC    I2'1'      ;File reference number
         DC    I4'Buffer' ;Pointer to data buffer
         DC    I4'$1000'  ;Number of bytes to read
TransCnt DS    4          ;# of bytes actually read

Buffer   DS    $1000      ;Data buffer
```

After every call to this subroutine, you must examine the 4-byte number at TransCnt to determine how many bytes were actually read. This number may be less than $1000 if GS/OS reaches the end-of-file position part way through the reading operation or if it encounters a newline character. (See the discussion of the NewLine command for information on newline characters.)

If the Read command returns error code $4C ("end of file"), no bytes were read, and you can close the file.

none	READ_BLOCK $80
GS/OS	ProDOS 8

Purpose:

To transfer one block (512 bytes) of information from an Apple-formatted disk device to a buffer in memory.

Under GS/OS, use the DRead command instead.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (3)
+1	unit_num	I	Unit number
+2 to +3	data_buffer	R	Pointer to the data input buffer
+4 to +5	block_num	I	Number of block to be read from

Warning: Do not use READ_BLOCK if you want your application to work with an AppleShare file server volume over AppleTalk.

Descriptions of parameters:

num_parms The number of parameters in the ProDOS 8 parameter table (always 3).

unit_num The slot and drive number for the disk drive to be accessed. The format of this byte is as follows:

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

ProDOS 8 assigns a drive number of 1 or 2 to each drive in the system. DR = 0 for drive 1, and DR = 1 for drive 2. SLOT is usually the actual slot number for the disk controller card (1–7 decimal; 001–111 binary) but may be the number of a phantom, or logical, slot.

The unit_num value for the /RAM volume is $B0, meaning it is the logical slot 3, drive 2 device.

data_buffer A pointer to the beginning of a 512-byte block of memory that is to hold the contents of the specified block when READ_BLOCK successfully completes.

block_num	The number of the block to be read. The permitted values for block_num depend on the disk device: • 0–279 for 5.25-inch drives • 0–1599 for 3.5-inch drives • 0–127 for the ProDOS 8 /RAM volume You can determine the volume size for a device using the GET_FILE_INFO command and specifying the name of the volume directory for the disk in the device. The size (in blocks) is returned at relative positions $5 and $6 in the parameter table.

Common error codes:

$27	The disk is unreadable, probably because a portion of the disk medium is permanently damaged. This error also occurs if the drive door on a 5.25-inch drive is open or no disk is in the drive.
$28	No device connected. This error comes back if you do not have a second 5.25-inch drive connected to the drive controller, but you try to access it.

Other possible error codes are $04, $07, $11, $2F, $53, $56.

Programming example:

READ_BLOCK is one of two low-level disk-access commands ProDOS 8 provides. (WRITE_BLOCK is the other.) Use it to read any block on a ProDOS-formatted disk, whether it be a file data block, index block, directory block, or a boot record block.

You can also use READ_BLOCK to read any sector on a DOS 3.3-formatted disk. See Appendix II for suggestions on how to do this.

Here's a short ProDOS 8 program that reads into memory the volume bit map block (block 6) on a 5.25-inch disk in slot 6, drive 1 and then calculates the number of free blocks on the disk:

```
          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 to 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 disk. If the bit is 1, the block is not in use; if it is 0, it is. This program simply scans through these 35 bytes and counts the number of 1 bits. The 2-byte result is stored in COUNTER and COUNTER + 1.

none	RENAME $C2
GS/OS	ProDOS 8

Purpose:

To change the name of a file on disk.

Under GS/OS, use the ChangePath command instead.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (2)
+1 to +2	curr_name	I	Pointer to current pathname
+3 to +4	new_name	I	Pointer to new pathname

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 2).
curr_name	A pointer to a class 0 ProDOS 8 string describing the current pathname of the file to be renamed. If the pathname specified is not preceded by a separator (/), the operating system appends the name to the default prefix to create a full pathname.
new_name	A pointer to a class 0 ProDOS 8 string describing the new pathname for the file. If the pathname specified is not preceded by a separator (/), the operating system appends the name to the default prefix to create a full pathname. The new_name must be the same as curr _name 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/CANADA/BC.LIONS /FOOTBALL/CANADA/VANCOUVER.LIONS but not as /FOOTBALL/USA/DETROIT.LIONS.

Common error codes:

$2B	The disk is write-protected.
$40	The pathname contains invalid characters, or a full pathname was not specified (and no default prefix has been set up). Verify 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 default prefix has been set up.
$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 default 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 default prefix.
$46	The file was not found.
$47	The new filename specified already exists.
$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. You can rename only closed files.

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

Programming example:

Here is a subroutine 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   STR '/SUPER.HEROES/BATMAN' ;Old pathname

PATH2   STR '/SUPER.HEROES/BRUCE.WAYNE' ;New pathname
```

Note that you cannot rename /SUPER.HEROES/BATMAN as /IDENTITIES/BRUCE. WAYNE because this would violate the rule that the two pathnames must identify files in the same directory.

ResetCache $2026	none
GS/OS	ProDOS 8

Purpose:

To force an immediate resizing of the GS/OS disk cache using the size value stored in Battery RAM. (Battery RAM holds system configuration and preferences information even when the Apple IIGS has been turned off.)

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (0)

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 0; the maximum is 0.

Common error codes:

[none]

Comments:

A program can use the Miscellaneous Tool Set's WriteBParam function to change the size of the GS/OS disk cache, as follows:

```
PushWord #newCacheSize
PushWord #$0081         ;Parameter reference number
LDX      #$0B03         ;WriteBParam
JSL      $E10000
```

The newCacheSize value represents the size of the cache (in K units) divided by 32. This means, for example, that you would use a value of 4 to set up a 128K cache. You can only set the cache size to a multiple of 32K.

The new cache size setting usually doesn't take effect until the system is rebooted. If the program calls ResetCache, however, the change takes effect immediately. Utility programs like the Disk Cache desk accessory on the GS/OS system disk use ResetCache.

SessionStatus $201F	none
GS/OS	ProDOS 8

Purpose:

To determine whether a write-deferral session (initiated with a BeginSession command) is in progress.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +3	status	R	Status code

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 0; the maximum is 1.
status	This code indicates whether a write-deferral session is in progress:

```
$0000   write-deferral session not in progress
$0001   write-deferral session in progress
```

Common error codes:

[none]

Comments:

Write-deferral sessions are useful for accelerating file transfer operations. You can begin such a session with BeginSession and end it with EndSession. SessionStatus tells you whether a session is currently in progress.

none	SET_BUF $D2
GS/OS	ProDOS 8

Purpose:

To move the ProDOS 8 file buffer for an open file from its current position to another 1024-byte area in memory.

There is no equivalent GS/OS command. GS/OS takes care of all buffer-management operations internally.

Parameter table:

ProDOS 8		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0	num_parms	I	Number of parameters (2)
+1	ref_num	I	Reference number for the file
+2 to +3	io_buffer	I	Pointer to I/O buffer

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 2).
ref_num	The reference number ProDOS 8 assigned to the file when it was first opened.
io_buffer	A 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. You might be using a reference number for a file that you've already closed.
$56	The pathname buffer address is invalid because it has been marked as in use in the ProDOS 8 system bit map. Specify a buffer address that does not conflict with areas already used by ProDOS 8 or its file buffers. Examine the system bit map to determine the free and protected areas.

Another possible error code is $04.

Programming example:

The following ProDOS 8 program will move the file buffer for file 1 from its current position to $2000. You are responsible for ensuring that the area $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
```

GS/OS	ProDOS 8
SetEOF **$2018**	**SET_EOF** **$D0**

GS/OS **ProDOS 8**

Purpose:

To change the current end-of-file pointer (EOF) of an open file. If you reduce EOF, all data blocks past the end of the new EOF are freed up; if you increase EOF, however, ProDOS 8 and GS/OS do not allocate new blocks for the file until you actually write data to the new part of the file. If the new EOF is less than Mark, Mark is set equal to the new EOF. You can change the EOF of any file whose write-enabled access code bit is set to 1.

Parameter table:

ProDOS 8

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (2)
+1	ref_num	I	Reference number for the file
+2 to +4	eof	I	The new end-of-file position

GS/OS

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (3)
+2 to +3	ref_num	I	Reference number for the file
+4 to +5	base	I	Code for determining new eof
+6 to +9	displacement	I	The new end-of-file position

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 2).
ref_num	The reference number ProDOS 8 or GS/OS assigned to the file when it was first opened.
eof	The new EOF position.

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 3; the maximum is 3.
base	This code tells GS/OS how to determine the new value for the end-of-file pointer:

```
$0000    new EOF = displacement
$0001    new EOF = old EOF + displacement
$0002    new EOF = Mark + displacement
$0003    new EOF = Mark - displacement
```

displacement	GS/OS uses this value in conjunction with the base code to determine the new value for the end-of-file pointer.

Common error codes:

$2B	The disk is write-protected.
$43	The file reference number is invalid. You might be using a reference number for a file that you've already closed.
$4D	The position is out of range.
$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, $07, $27, $4E.

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. If you are not careful, and the new file is smaller than the original, the tail end of the old file (the part 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 SetEOF 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, and then re-creating, a file.

Suppose the new file length is $1534 bytes. To set the EOF for this file, you would call a GS/OS subroutine like this:

```
LDA     #$1534      ;Set up new EOF value
STA     New_EOF
LDA     #$0000
STA     New_EOF+2

_SetEOF EOF_Parms
BCS     Error       ;Branch if error occurred
RTS
```

```
EOF_Parms DC      I2'3'        ;The # of parameters
          DC      I2'1'        ;File reference number
          DC      I2'0'        ;EOF = displacement
New_EOF   DS      4            ;New EOF position
```

SetFileInfo $2005	SET_FILE_INFO $C3
GS/OS	ProDOS 8

Purpose:

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

Parameter table:

ProDOS 8

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (10)
+1 to +2	pathname	I	Pointer to the pathname string
+3	access	I	Access code
+4	file_type	I	File type code
+5 to +6	aux_type	I	Auxiliary type code
+7	[not used]	I	
+8 to +9	[not used]	I	
+10 to +11	modify_date	I	Modification date
+12 to +13	modify_time	I	Modification time

GS/OS

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (12)
+2 to +5	pathname	I	Pointer to the pathname string
+6 to +7	access	I	Access code
+8 to +9	file_type	I	File type code
+10 to +13	aux_type	I	Auxiliary type code
+14 to +15	[not used]	I	

+16 to +23	create_dt	I	Creation date and time
+24 to +31	modify_dt	I	Modification date and time
+32 to +35	option_list	I	Pointer to option list
+36 to +39	[not used]	I	
+40 to +43	[not used]	I	
+44 to +47	[not used]	I	
+48 to +51	[not used]	I	

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 7).
pathname	A pointer to a class 0 (ProDOS 8) or class 1 (GS/OS) string describing the pathname of the file to be used. If the pathname specified is not preceded by a separator (/ for ProDOS 8; / or : for GS/OS), the operating system appends the name to the default prefix (in GS/OS, this is the 0/ prefix) to create a full pathname.
access	This field contains several 1-bit codes that define the access attributes of the file. See Figure 2-10 for a description of these bits. Note, however, that under GS/OS you cannot clear the backup-needed attribute with SetFileInfo; use the ClearBackup command instead. Under ProDOS 8, you can clear the bit but only if you first store $FF at BUBIT ($BF95) in the ProDOS 8 system global page. Backup programs should clear this attribute to indicate that the file has been backed up.
file_type	This code indicates the type of data the file holds. See Table 2-5 for a description of ProDOS file type codes. Under the ProDOS FST, only the low-order byte of file_type is significant.
aux_type	This is the auxiliary type code. The meaning of the code depends on the file type code and on the program that created the file in the first place. For SYS, BIN, BAS, and VAR files, it is a default loading address; for TXT files, it is a record length; for SRC files, it is an APW language type code. Under the ProDOS FST, only the low-order word is significant.
[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.

modify_date	This field contains the date (year, month, day) the file was last modified. The current date should be stored here before executing the command. Figure 8-1 in Chapter 8 shows the format of these bytes.
modify_time	This field contains the time (hour, minute) the file was last modified. The current time should be stored here before executing the command. Figure 8-1 in Chapter 8 shows the format of these bytes.
create_date	This field contains the date (year, month, day) the file was created. Figure 8-1 in Chapter 8 shows the format of these bytes.
create_time	This field contains the time (hour, minute) the file was created. Figure 8-1 in Chapter 8 shows the format of these bytes.
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 12.
create_td	The time and date of creation. These eight bytes represent the following parameters in the following order:

```
seconds
minutes
hour          in 24-hour military format
year          year minus 1900
day           day of month minus 1
month         0 = January, 1 = February, and so on
[not used]
weekday       1 = Sunday, 2 = Monday, and so on
```

Note: This format is the same as the one used by the ReadTimeHex function in the IIGS's Miscellaneous Tool Set but is different from the one used in a standard file entry for the ProDOS file system.

modify_td	The time and date of last modification. The ordering of these 8 bytes is the same as for create_time.
option_list	A pointer to a class 1 input buffer containing information unique to the file system translator used to access the file. The ProDOS FST does not require any such information.

Note: The parameters marked by [not used] must be set to zero.

Common error codes:

$2B	The disk is write-protected.
$40	The pathname contains invalid characters, or a full pathname was not specified (and no default prefix has been set up). Verify 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 default prefix has been set up.

$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 default 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 default prefix.
$46	The file was not found.
$4E	The access code specified for the file is not permitted. Solution: Ensure that the reserved bits of the access code are all zero.

Other possible error codes are $04, $07, $27, $4A, $4B, $52, $53, $58.

Programming example:

The following ProDOS 8 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 default prefix).

```
         LDA #10
         STA PARMTBL      ;Store # of parms for GET_FILE_INFO
         JSR MLI
         DFB $C4          ;GET_FILE_INFO
         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 STR 'PRISONER'  ;Pathname (in ASCII)
```

There are two interesting things to note about this program. First, it uses the GET_FILE_INFO command to read the file's current access code and other directory information. Since the parameter table for this command and 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 Figure 2-10 in Chapter 2 indicates, clearing these bits will disable write, rename, and destroy operations, respectively.

SetLevel **$201A**	**none**
GS/OS	**ProDOS 8**

Purpose:

To set the system file level.

There is no equivalent ProDOS 8 command. To change the value of the system file level, store the new value at LEVEL ($BF94) in the system global page.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +3	level	I	The new system file level

Meanings of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
level	The value of the system file level. Legal values range from $0000 to $00FF.

Common error codes:

$59	Invalid file level. The file level must be a number between $0000 and $00FF.

Another possible error code is $07.

Programming example:

Here is how to set the system file level to 2:

```
         _SetLevel SL_Parms
         RTS

SL_Parms ANOP
         DC   I2'1'     ;The number of parameters
         DC   I2'2'     ;New system file level
```

The system file level affects the performance of subsequent Open, Close, and Flush operations.

SetMark $2016	SET_MARK $CE
GS/OS	ProDOS 8

Purpose:

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

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (2)
+1	ref_num	I	Reference number for the file
+2 to +4	position	I	The new mark position

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (3)
+2 to +3	ref_num	I	Reference number for the file
+4 to +5	base	I	Code for determining new mark
+6 to +9	displacement	I	The new mark position

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 2).
ref_num	The reference number ProDOS 8 or GS/OS assigned to the file when it was first opened.
position	This field holds the new Mark position. This position must not exceed the EOF position for the file.
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 3; the maximum is 3.

base	This code tells GS/OS how to determine the new value for the Mark pointer:
	`$0000  new Mark = displacement`
	`$0001  new Mark = EOF - displacement`
	`$0002  new Mark = old Mark + displacement`
	`$0003  new Mark = old Mark - displacement`

displacement	GS/OS uses this value in conjunction with the base code to determine the new value for the Mark pointer.

Common error codes:

$43	The file reference number is invalid. You might be using a reference number for a file that you've already closed.
$4D	The Mark position is larger than the EOF position.

Other possible error codes are $04, $07, $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) under GS/OS:

```
          LDA #$08CE
          STA NewMark
          LDA #$0000
          STA NewMark+2    ;(high-order word is zero)

          _SetMark SM_Parms
          BCS Error         ;Branch if error occurred
          RTS

PARMTBL   DC   I2'3'       ;The # of parameters
          DC   I2'1'       ;File reference number
          DC   I2'0'       ;New Mark = displacement
New_Mark  DS   4           ;New Mark position
```

Remember that the Mark position cannot exceed the EOF position.

SetPrefix **$2009**	**SET_PREFIX** **$C6**
GS/OS	**ProDOS 8**

Purpose:

To set the default prefix to a specified directory path. When you pass a filename or partial pathname to an MLI command, ProDOS 8 or GS/OS automatically converts it into a full pathname by appending it to the current value of the prefix you're trying to set.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (1)
+1 to +2	prefix	I	Pointer to the new prefix string

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (2)
+2 to +3	prefix_num	I	Prefix number (0 to 31)
+4 to +7	prefix	I	Pointer to the new prefix string

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 1).
pathname	A pointer to a class 0 (ProDOS 8) or class 1 (GS/OS) string describing the pathname of the prefix. If the pathname specified is not preceded by a separator (/ for ProDOS 8; / or : for GS/OS), ProDOS 8 appends the name to the default prefix and GS/OS appends it to the prefix string for the prefix you're trying to set, thus creating a full pathname. An optional separator may be placed at the end of the prefix pathname.
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 2.

prefix_num	The GS/OS prefix number (0 to 31). This is a binary number, not an ASCII number string followed by a slash.

Common error codes:

$40	The pathname contains invalid characters, or a full pathname was not specified (and no default prefix has been set up). Verify 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 default prefix has been set up.
$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 default 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 default prefix.
$46	The file was not found.
$4B	A nondirectory 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, $07, $27, $53.

Programming example:

It is often convenient to be able to set the ProDOS 8 default 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 default 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 discrepancy.

```
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
INC PFXNAME     ;(add 1 for leading slash)

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  1           ;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 by SET_PREFIX, storing it at the previous memory location (PFXNAME), and then overwriting the name length byte with the slash. After this has been done, the data structure beginning with PFXNAME is in the format required by SET_PREFIX.

SetSysPrefs **$200C**	**none**
GS/OS	**ProDOS 8**

Purpose:

To set the GS/OS global system preferences.

There is no equivalent ProDOS 8 command.

Parameter table:

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +3	preferences	I	System preferences

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
preferences	Use bit 15 of this value to indicate whether GS/OS should display a mount volume dialog box if a disk volume can't be found during execution of certain GS/OS commands:

```
bit 15    1 = display mount volume dialog box
          0 = don't display the dialog box
```

Common error codes:

[none]

Comments:

GS/OS commands that have pathnames as input parameters normally display a mount volume dialog box (to ask the user to insert a specified disk volume) if the commands can't find the volume they may need to complete. If the application wants to handle "volume not found" errors itself, it can use SetSysPrefs to clear bit 15 of the preferences word.

UnbindInt $2032		none
GS/OS		ProDOS 8

Purpose:

To remove a GS/OS interrupt handling subroutine.

Under ProDOS 8, use the DEALLOC_INT command instead.

Parameter table:

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (1)
+2 to +3	int_num	I	Interrupt handler reference number

Descriptions of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 1; the maximum is 1.
int_num	The identification number for the interrupt handler. GS/OS assigned this number when the handler was installed using the BindInt command.

Important: Do not remove an interrupt-handling subroutine until your application has first told the source of the interrupts to stop generating interrupts. If you remove the subroutine first, the system will crash the next time an interrupt occurs.

Common error codes:

$53	The int_num parameter is not valid. Use the number BindInt returned when you installed the interrupt handler.

Other possible error codes are $04, $07.

Comments:

See Chapter 6 for a discussion of how to handle interrupts in a GS/OS environment.

Volume $2008		**none**
GS/OS		**ProDOS 8**

Purpose:

To return status information about a disk volume.

Under ProDOS 8, use the ON_LINE command instead.

Parameter table:

GS/OS		*Input*	
		or	
Offset	*Symbolic Name*	*Result*	*Description*
+0 to +1	pcount	I	Number of parameters (6)
+2 to +5	dev_name	I	Pointer to the device name string
+6 to +9	vol_name	R	Pointer to the volume name string
+10 to +13	total_blocks	R	Size of the volume in blocks
+14 to +17	free_blocks	R	Number of unused blocks
+18 to +19	file_sys_id	R	Operating system ID code
+20 to +21	block_size	R	Number of bytes in a block

Meanings of parameters:

pcount	The number of parameters in the GS/OS parameter table. The minimum value is 2; the maximum is 6.
dev_name	A pointer to a class 1 string containing the device name. (Use DInfo to get a list of active device names.)
vol_name	A pointer to a class 1 output buffer where GS/OS returns the disk volume name string. The buffer should be 35 bytes long.
total_blocks	The total number of blocks on the disk volume.
free_blocks	The number of unused blocks on the disk volume. For the High Sierra FST, this value is always zero.
file_sys_id	The identification code for the file system on the disk volume. The currently defined values are:

$00	= [reserved]
$01	= ProDOS/SOS
$02	= DOS 3.3
$03	= DOS 3.2/3.1
$04	= Apple II Pascal
$05	= Macintosh MFS
$06	= Macintosh HFS
$07	= Macintosh XL (LISA)
$08	= Apple CP/M
$09	= [reserved]
$0A	= MS-DOS
$0B	= High Sierra (CD-ROM)
$0C	= ISO 9660 (CD-ROM)

block_size The size of a disk block in bytes.

Common error codes:

$10	The specified device name does not exist.
$27	The disk is unreadable probably because a portion of the disk medium is permanently damaged. This error also occurs if the drive door on a 5.25-inch drive is open or no disk is in the drive.
$28	No device connected. This error is returned if you do not have a second 5.25-inch drive connected to the drive controller, but you try to access it.
$2F	Device not on line. This error is returned if no disk is in a 3.5-inch drive.

Other possible error codes are $07, $11, $2E, $40, $45, $4A, $52, $55, $57, $58.

Programming example:

You can use the DInfo command to determine the GS/OS device names for disks attached to the system. It is these names that Volume requires as inputs.

To get the status for a particular device, say .APPLEDISK3.5A, so that you can determine the number of blocks in use on the disk, use a subroutine like this:

```
_Volume Vol_Parms

SEC                     ;Used blocks = total blocks
LDA  total_blk          ; minus free blocks
SBC  free_blk
STA  used_blk
LDA  total_blk+2
SBC  free_blk+2
STA  used_blk+2
RTS
```

```
Vol_Parms ANOP
          DC   I2'6'          ;The number of parameters
          DC   I4'DevName'    ;Pointer to device name
          DC   I4'VolSpace'   ;Pointer to volume name
total_blk DS   4              ;total blocks
free_blk  DS   4              ;free blocks
sys_id    DS   2              ;file system ID
block_sz  DS   2              ;bytes per block

DevName   GSString '.APPLEDISK3.5A'
VolSpace  DC   I2'35'         ;Size of class 1 buffer
VolName   DS   33             ;Space for volume name

used_blk DS    4
```

Write $2013	WRITE $CB
GS/OS	ProDOS 8

Purpose:

To write bytes of data to an open file. Writing begins at the current Mark position. After you write the data to the file, the operating system increases the Mark position by the number of bytes written. If the new Mark position is greater than EOF, EOF is set equal to Mark.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (4)
+1	ref_num	I	Reference number for the file
+2 to +3	data_buffer	I	Pointer to start of data buffer
+4 to +5	request_count	I	Number of bytes to write
+6 to +7	transfer_count	R	Number of bytes actually written

GS/OS *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	pcount	I	Number of parameters (5)
+2 to +3	ref_num	I	Reference number for the file
+4 to +7	data_buffer	I	Pointer to start of data buffer
+8 to +11	request_count	I	Number of bytes to write
+12 to +15	transfer_count	R	Number of bytes actually written
+16 to +17	cache_priority	I	Block-caching priority code

Descriptions of parameters:

num_parms	The number of parameters in the ProDOS 8 parameter table (always 4).

ref_num	The reference number ProDOS 8 or GS/OS assigned to the file when it was first opened.
data_buffer	A pointer to the beginning of a block of memory that contains the data to be written to the file.
request_count	The number of characters to be written to the file from the buffer pointed to by data_buffer.
transfer_count	This result contains the number of characters actually written to the file and usually equals request_count. However, it will be less than request_count 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.
pcount	The number of parameters in the GS/OS parameter table. The minimum value is 4; the maximum is 5 (or 4 for a character device file).
cache_priority	This code indicates how GS/OS is to handle the caching of disk blocks related to the write operation: `$0000` do not cache blocks; `$0001` cache blocks. This field is not used for character devices.

Common error codes:

$2B	The disk is write-protected.
$43	The file reference number is invalid. You might be using a reference number for a file that you've already closed.
$48	The disk is full.
$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 pathname buffer address is invalid because it has been marked as in use in the ProDOS 8 system bit map. Specify a buffer address that does not conflict with areas already used by ProDOS 8 or its file buffers. Examine the system bit map to determine the free and protected areas.

Other possible error codes are $04, $07, $27.

Programming example:

This GS/OS subroutine writes 256 bytes to file 2; the data buffer begins at location Buffer.

```
         _Write WR_Parms
         BCS    Error        ;Branch if error occurred
         RTS

WR_Parms DC   I2'4'          ;Parameter count
         DC   I2'2'          ;File reference number (assume #2)
         DC   I4'Buffer'     ;Pointer to data buffer
         DC   I4'256'        ;Number of bytes to write
TransCnt DS   4              ;# of bytes actually written

Buffer   DS   256            ;Data buffer
```

If no error occurred, the number stored at TransCnt should be equal to 256, the request_count. But if the disk becomes full during the write, TransCnt will be less than 256.

If you want to append data to the end of an open file, use GetEOF to determine the file size, and then use SetMark to set the Mark pointer to the EOF value. Subsequent Write operations will begin at the end of the file.

none	WRITE_BLOCK $81
GS/OS	ProDOS 8

Purpose:

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

Under GS/OS, use the DWrite command instead.

Parameter table:

ProDOS 8 *Offset*	*Symbolic Name*	*Input or Result*	*Description*
+0	num_parms	I	Number of parameters (3)
+1	unit_num	I	Unit number
+2 to +3	data_buffer	I	Pointer to the data output buffer
+4 to +5	block_num	I	Number of block to be written to

Warning: Do not use WRITE_BLOCK if you want your application to work with an AppleShare file server volume over AppleTalk.

Descriptions of parameters:

num_parms The number of parameters in the ProDOS 8 parameter table (always 3).

unit_num The slot and drive number for the disk drive to be accessed. The format of this byte is as follows:

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

ProDOS 8 assigns a drive number of 1 or 2 to each drive in the system. DR = 0 for drive 1, and DR = 1 for drive 2. SLOT is usually the actual slot number for the disk controller card (1–7 decimal; 001–111 binary) but may be the number of a phantom, or logical, slot.

The unit_num value for the /RAM volume is $B0, meaning it is the logical slot 3, drive 2 device.

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

block_num	The number of the block to be accessed. The permitted values for block_num depend on the disk device: • 0–279 for 5.25-inch drives • 0–1599 for 3.5-inch drives • 0–127 for the ProDOS 8 /RAM volume You can determine the volume size for a device using the GET _FILE_INFO command and specifying the name of the volume directory for the disk in the device. The size (in blocks) is returned at relative positions $5 and $6 in the parameter table.

Common error codes:

$27	The disk is unwriteable probably because a portion of the disk medium is permanently damaged. This error also occurs if the drive door on a 5.25-inch drive is open or no disk is in the drive.
$28	No device connected. This error is returned if you do not have a second 5.25-inch drive connected to the drive controller, but you try to access it.
$2B	The disk is write-protected.

Other possible error codes are $04, $07, $11, $2F, $53, $56.

Programming example:

WRITE_BLOCK is perhaps the most dangerous of all the ProDOS 8 commands since it lets you overwrite any block on the disk with any data you want. It is very useful, however, for trying to recover damaged files and making backup copies of disks.

It is also possible to use WRITE_BLOCK to write to any sector on a DOS 3.3-formatted disk. See Appendix II for suggestions on how to do this.

Here's an interesting ProDOS 8 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 disk. The chore is simplified because the parameter tables for READ_BLOCK and WRITE_BLOCK are identical.

Of course, the preferred way to rename a volume directory is to use the RENAME command.

CHAPTER 5

System Programs

A system program is simply an assembly-language program, requiring the resources of GS/OS or ProDOS 8, that communicates directly with users. It is generally a primary application like a word processor, spreadsheet, or telecommunications program or a programming environment for languages like C, BASIC, or Pascal.

Under ProDOS 8, a system program takes control of the entire Apple II memory space, except for the portion ProDOS 8 uses, and is responsible for managing it properly. This space includes the 64K main memory bank in all Apple IIs and the 64K auxiliary memory bank in a IIc, IIGS, or IIe with an extended 80-column text card.

Under GS/OS, a system program must use the IIGS Memory Manager to allocate blocks of memory it may need. Since the Memory Manager allocates only blocks that are not in use, the system program will peacefully co-exist with other programs that may be in memory at the same time.

The operating system identifies a system program by inspecting its file type code. A ProDOS 8 system program has a file type code of $FF and a directory mnemonic of SYS. A GS/OS system program has a file type code of $B3 and a directory mnemonic of S16. But assigning a file type code of $B3 or $FF to a program file is not enough to convert it to a true system program. As we see, a system program must also follow certain software conventions and take care not to interfere with memory areas used by the operating system or other co-resident programs.

In this chapter, we review the features of well-designed GS/OS and ProDOS 8 system programs. We then examine one very common, and very important, ProDOS 8 system program, BASIC.SYSTEM. The discussion of BASIC.SYSTEM is quite detailed: We see how it installs itself in the system, how it calls ProDOS 8 commands, how its command set can be extended, and how it handles errors. We also take a close look at the global page it uses to manage the communication between ProDOS 8 and Applesoft programs. The analysis of BASIC.SYSTEM should assist you in creating your own ProDOS 8 system programs.

THE STRUCTURE OF A GS/OS SYSTEM PROGRAM

To be considered a true GS/OS system program, a program must possess four basic properties.

1. The executable code for the program must be in 65816/6502 assembly language, and it must be stored to disk as a load file in Apple's object module format (OMF). This is not to say you cannot use a high-level language like C, Pascal, or BASIC to create a system program. You can as long as the language compiler creates native assembly-language object code. The Apple Programmer's Workshop linker takes care of creating load files for you, as does the linker for Merlin 8/16.
2. The program must have a file type code of $B3. You can use the Apple Programmer's Workshop FILETYPE command to set the file type after compiling and linking an application. By assigning the $B3 file type code, you can execute the programs directly from the Apple IIGS Finder.
3. The program must use the IIGS toolbox's Memory Manager tool set to allocate any blocks of memory it may need. By using the Memory Manager, the program can avoid overwriting memory areas used by other co-resident programs, such as desk accessories, printer drivers, or interrupt handlers.
4. The program must end using the GS/OS Quit command. It can use the Quit command to return to the system program that called it (usually the Finder) or to call another system program as if it were a subroutine, regaining control when the other system program ends. (See Chapter 4 for a discussion of the Quit command.)

In general, you can assign any valid name to a system program. If you want to create a disk that automatically boots and runs the system program, you should assign it a name that ends in .SYS16, place the program file in the root directory of the disk, and delete the START program from the SYSTEM/ subdirectory. Alternatively, you can name the system program START and put it in the SYSTEM/ subdirectory.

Entry Conditions

GS/OS launches a system program by first loading it into memory using the System Loader tool set's InitialLoad function. It then uses the Memory Manager to allocate a direct page/stack space for use by the system program.

The size of the direct page/stack space depends on whether the program includes a direct page/stack object segment. If it doesn't (the usual case for most applications you're likely to develop), GS/OS uses the Memory Manager's NewHandle function to allocate a 4096-byte space in bank $00, which begins on a page boundary. (The other important Memory Manager attributes of the block are: locked, fixed, purge level 1, may use special memory, and no fixed starting address.)

If the program file does include a direct page/stack object segment, GS/OS allocates a direct page/stack space that is the same size as the object segment. (See Chapter 7 of the *Apple IIGS Programmer's Workshop Reference* for how to create a direct page/stack object segment.)

In either situation, GS/OS sets the A (accumulator), D (direct page), and SP (stack pointer) registers to the following values before passing control to the program:

A = the User ID the System Loader assigns to the program.

D = the address of the first byte in the direct page/stack space.

SP = the address of the last byte in the direct page/stack space.

Note that the stack occupies the upper end of the direct page/stack space. Since the stack grows downward in memory, it may eventually collide with the portion of the space used for direct page storage. It is the responsibility of the application to ensure it allocates enough direct page/stack space to prevent such a collision.

The direct page/stack space that the System Loader automatically sets up is made purgeable when the system program ends by calling the Quit command. This means the application does not have to explicitly release this memory with the DisposeHandle function before ending.

Your system program can also allocate a direct page/stack space on the fly at execution time. To do this, it should first call DisposeHandle to free up the space the System Loader allocates. Use FindHandle to determine the handle to this space; the high word of the long address that FindHandle requires is $0000; the low word is the value stored in the D or SP register. Here is a piece of code that will do the trick:

```
PHA             ;Space for result (long)
PHA
PEA $0000       ;High word of addr is always zero
PHD             ;Low word of addr in dp/stack space
_FindHandle     ;(leave handle on stack)
_DisposeHandle
```

The program must then use NewHandle to allocate the direct page/stack space it requires (the Memory Manager attributes for this space should be as described earlier in this section), and then put the starting address of the block in the D register and the ending address in the SP register. Here is a subroutine that performs these chores (UserID is a variable that holds the program's master user ID):

```
DP_Hndl    GEQU    $00              ;(Assume $00 is free)

           PHA                      ;Space for result
           PHA
           PushLong #$800           ;2K space
           PushWord UserID          ;Use program's user ID
PushWord #$C105           ;Attributes
PushLong #$00000000       ;(Any bank $00 address)
_NewHandle

PLA                       ;Pop handle
STA      DP_Hndl
PLA
STA      DP_Hndl+2

LDA      [DP_Hndl]        ;Get absolute address
TCD                       ;Set up new direct page

CLC                       ;Calculate address of
ADC      #$800            ;the last byte in space
DEC      A
TAX
TXS                       ;Set up new stack ptr
RTS
```

Note that the user ID for the direct page/stack memory block should be set to the system program's master user ID so that the block will be automatically discarded when the system program ends. The master user ID is in the A register when the system program starts up; the Memory Manager's MMStartup function returns the same value.

THE STRUCTURE OF A PRODOS 8 SYSTEM PROGRAM

A properly designed ProDOS 8 system program is an executable assembly-language program adhering to certain conventions and protocols that relate to its internal structure and the way it takes control of the system.

First, a system program must be designed to be loaded and executed beginning at location $2000 in main memory although it can later relocate itself anywhere else in memory not used by ProDOS 8 or system Monitor routines. The load address of $2000 is mandatory.

You can use the BASIC.SYSTEM - (dash) command to execute a system program. It is also possible to automatically execute a system program when ProDOS 8 first starts up by giving the program a name of the form xxxxxxxx.SYSTEM and ensuring it is the first entry in the volume directory with such a name.

Some system programs follow an optional *auto-run protocol* that allows a ProDOS 8 selector program to pass the name of a file to them. (Recall from Chapter 4 that a selector program gets control when an application calls the QUIT command.) The standard ProDOS 8 selector program does not allow for filename passing, but many independent selectors, such as ProSel and RunRun, do. The description of the QUIT command in Chapter 4 includes instructions on how to write your own selector.

The auto-run 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, the sixth byte ($2005) holds the size of a buffer that begins 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 a file the system file is to work with when it first starts up. (A system program file usually has a default filename stored here.) Thus if the selector program detects the presence of the three identification bytes, it could prompt the user to enter the name of a data file, load the system program, store the length and name of the data file beginning at $2006, and then execute the system program by jumping to $2000.

The BASIC.SYSTEM system program adheres to the auto-run protocol. Here is what the first part 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 file
            .
            .
START1                       ;Main program entry point
```

As you can see, BASIC.SYSTEM defines a default auto-run file called STARTUP. This is the name of the Applesoft program BASIC.SYSTEM loads and runs whenever it starts up unless the selector passes a different name.

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 it is located in. This is helpful for loading subsidiary programs or data files located in the same directory as the system program itself.

Often, a system program defines an interpretive programming environment in which application programs can be written and executed. (BASIC.SYSTEM is the best example of such a program.) In this case, the code for the interpreter should be tucked away in a safe place that will not conflict with memory areas the application program can use. The best position for the code is in a contiguous block at the upper end of main RAM memory, just below the ProDOS 8 global page at $BF00; this leaves the space from $800 to the start of the code free for use as a work area. The system program can protect the code space by setting to 1 those bits in the system bit map corresponding to the pages in use. If this is done, the ProDOS 8 command interpreter will not allow these areas to be inadvertently 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 perform several preliminary housekeeping chores.

- Initialize the microprocessor stack pointer. To ensure 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 with the following two instructions:

```
LDX #$FF
TXS
```

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

- Initialize the reset vector. When reset is pressed on an Apple II, control ultimately passes to the subroutine whose address is stored in the reset vector at SOFTEV ($3F2–$3F3) but only 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 reboots 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 open files and then jump to the cold start entry point of the system program. It is not safe to do anything else 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 8 global page. IBAKVER ($BFFC) must be set equal to the earliest version of ProDOS 8 the system program will work with; store a 0 here if any version will do. IVERSION ($BFFD) must be set equal to the version number of the system program being used.

When these chores have been completed, the system program can begin its main duties. If the system program adheres to the auto-run protocol, it must start working with the file whose name (preceded by a length byte) is stored beginning at $2005. The system program is then free to do almost anything it wants as long as it does not overwrite the ProDOS 8 system global page (page $BF) or data areas in other pages used by ProDOS 8 or system Monitor subroutines the system program might call. (See Chapter 3 for a discussion of ProDOS 8 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 2-5 in Chapter 2 for a description of the file type codes ProDOS uses for standard data files.)

When a system program creates a file, it can use the 2-byte auxiliary type code in its directory entry (at relative bytes $1F and $20; see Chapter 2) to hold miscellaneous information about the file. This code is saved to disk when you first create the file with the CREATE command; you can change it with the SET_FILE_INFO command. Here is the meaning of the auxiliary type code for each type of file BASIC.SYSTEM uses:

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

When the time comes for the system program to quit, the system program should first scramble the PWREDUP byte by decrementing it; this causes the system to reboot if reset is pressed. It should then close all open files and reconnect /RAM if it was earlier disconnected. (See Chapter 7 for instructions on how to do this.) Finally, it should pass control to another system program with the QUIT command. As we saw in Chapter 4, this causes the standard ProDOS 8 selector program to be executed. Here is what the code will look like:

```
        .
        [close all open files]
        [restore /RAM]
        .
        DEC $3F4          ;Scramble PWREDUP byte
        JSR $BF00         ;Call the MLI
        DFB $65           ;QUIT
        DA  PARMTBL
        BCS ERROR
        BRK               ;(shouldn't get here)

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 8 selector asks you to enter the prefix and pathname of the next system program to be loaded and executed.

If your ProDOS 8 application is running on a IIGS, and the bootup operating system was GS/OS, you can also use QUIT to transfer control directly to another ProDOS 8 or GS/OS system program. (See the discussion of the QUIT command in Chapter 4 for how to do this.)

THE BASIC.SYSTEM INTERPRETER

The BASIC.SYSTEM interpreter is probably the most commonly used ProDOS 8 system program. It is the program loaded whenever an Applesoft programming environment is going to be used; it extends the Applesoft command set by providing a group of 32 disk commands an Applesoft program can use. 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 entered while the system is in Applesoft command mode. But if a file has previously been opened for read operations, it gets 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] code; such arguments are assumed to be BASIC.SYSTEM disk commands, and BASIC.SYSTEM tries to interpret them as such. The output subroutine can spot these PRINT statements because BASIC.SYSTEM always operates with Applesoft trace mode on; this means line numbers will be sent to the output subroutine before the line is actually executed, giving BASIC.SYSTEM a chance to check any PRINT statements on that line. (By the way, the line numbers generated in trace mode are not displayed by BASIC.SYSTEM unless the Applesoft TRACE command has been executed.)

Figure 5-1 shows a BASIC.SYSTEM memory map. 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 8 system global page), reserves a 1K general-purpose file buffer from \$9600 to \$99FF, and then 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 \$0800 to \$95FF free for Applesoft program and variable storage.

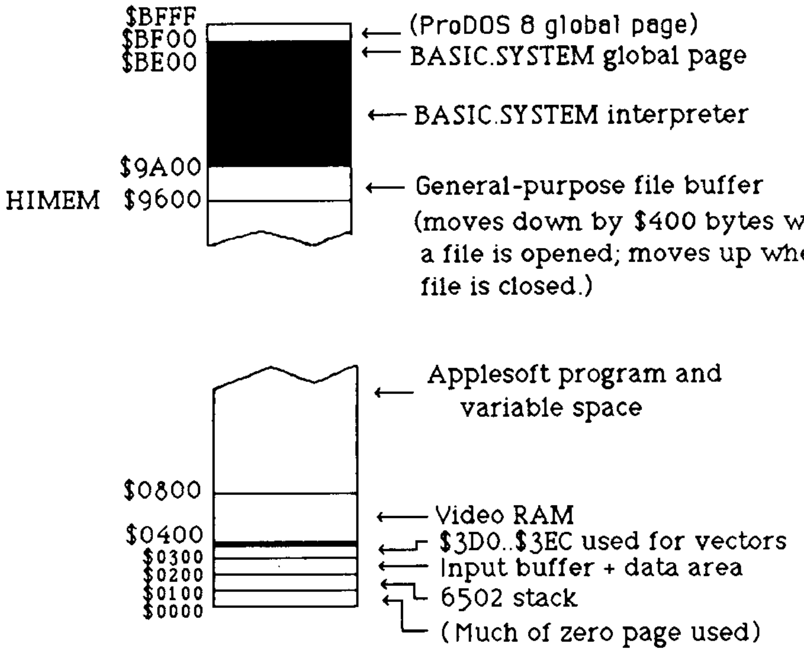
BASIC.SYSTEM also uses the area between \$3D0 and \$3EC for storage of position-independent vectors to some of its internal subroutines. We examine how BASIC.SYSTEM uses page three in more detail later in this chapter.

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 microprocessor stack at \$100–\$1FF, and several locations in zero page. (See Chapter 4 of *Inside the Apple IIe* for a detailed description of how Applesoft uses these areas.) Other areas, such as the video RAM area from \$400 to \$7FF and the system vector area from \$3ED to \$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), general file management (CAT, CATALOG, CREATE, DELETE, LOCK, PREFIX,

Figure 5-1 BASIC.SYSTEM memory map



RENAME, UNLOCK, and VERIFY), or program file loading and execution (-, BLOAD, BRUN, BSAVE, EXEC, LOAD, RUN, and SAVE). There are also commands for effecting I/O redirection (IN# and 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 8 system program (BYE). One command (NOMON) is allowed but does nothing; 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 IIc, 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 command mode, you don't have to worry about the [Control-D]. All 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 its commands to be entered from the keyboard in this way.

Most BASIC.SYSTEM commands support, or require, several parameters for specifying such things as the pathname for the file to be acted on, loading addresses, and lengths. Table 5-1 gives brief descriptions of the 13 different parameters recognized by BASIC.SYSTEM.

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, which requires two pathnames; the second pathname must appear right after the first one.

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. But 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 automatically uses 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 nondefault drive.

Let's now take a quick look at each of the 32 BASIC.SYSTEM commands. Table 5-2 summarizes the command syntax for each of these commands. (See Apple's *BASIC Programming with ProDOS* for detailed information on these commands; see the bibliography in Appendix III.) These commands can be divided into four distinct categories: file management commands, file loading and execution commands, file input/output commands, and miscellaneous commands.

File Management Commands

CAT. This command displays 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

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[b]
,A#[a]	Starting address	\$0000–\$FFFF
,B#	Byte number	\$0000–\$FFFFF
,D#	Disk drive number	1–2[c]
,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[c]
,T#	File type code	\$00–\$FF[d]
,V#	Volume number	\$00–\$FF

NOTES:
[a]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 $).
[b]Hexadecimal values are not allowed for snum.
[c]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 occur.
[d]A three-character file type mnemonic corresponding to a value can be specified with the T parameter instead. Table 2-5 in Chapter 2 shows the mnemonics available.

CAT command are displayed. (If no such parameter is specified, the currently active directory is used.) CAT also displays the type of each file (as a three-character mnemonic such as BAS, BIN, TXT, and SYS; see Table 2-4), the number of blocks it occupies, and the date it was last modified. After the names of all files have been listed, the number of blocks free and blocks used on the disk are displayed.

CATALOG. This command is similar to CAT. It displays the very same information for each file as well as its time of last modification, creation date and time, size (in bytes), and "subtype" entry (the file's auxiliary type code; the entries displayed are the default loading

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#]

Table 5-2 Continued

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 and vertical bars separate alternative parameters.

address for a BIN file and the record length for a TXT file). It also displays the disk capacity in blocks.

CREATE. This command creates a directory entry for a specified file. It is primarily for creating 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, you would enter the command

```
CREATE DEMO.PROGRAMS
```

from the keyboard. After you do this, the subdirectory appears as a file entry when you catalog the directory in which the file was created. 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 deletes a file by removing its entry from the directory and altering the volume bit map to free up the blocks the file uses. Only unlocked files can be erased with the DELETE command.

LOCK. This command protects a 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 command); if the name of the file is preceded by an asterisk (*), it is locked.

PREFIX. This command defines 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 is appended to the default prefix.

RENAME. This command changes 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 checks whether a file exists. If no error occurs, the file does exist. Entering VERIFY by itself (that is, without a pathname) causes Apple's copyright notice to appear.

File Loading and Execution Commands

*- **(dash).*** This is the intelligent run command. Its parameter can be the pathname of an Applesoft program, a binary program, or a textfile, in which cases the - behaves exactly like a RUN, BRUN, or EXEC command, respectively. The - command can also be used to execute ProDOS 8 system (SYS) programs.

BLOAD. This command transfers 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 with the ,Ttype parameter. The BLOAD command can also be used without the ,A# parameter; in this case, the file is loaded at the location from which it was originally saved to 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 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 loads, it is automatically executed. Execution begins at the loading address. The BRUN command can be used with binary (BIN) files only.

BSAVE. This command saves 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 to $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 the write operation is to take place.

EXEC. This command redirects 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 command mode, the screen clears, and the disk is 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 loads an Applesoft program into memory.

RUN. This command is the same as the LOAD command except that after the program is loaded, it is automatically executed. The ,@# parameter can be used to specify the Applesoft line number to be executed first; the default is the first line number. (If RUN is entered without a pathname, the program already in memory is executed.)

SAVE. This command saves an Applesoft program to a file on disk. The file type mnemonic for a program file is BAS.

File Input/Output Commands

OPEN. This command opens a file (by default, a TXT file) for reading and writing. If the pathname specified does not exist, a new file is 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 lines of information are stored one after another, separated only by a carriage return code; if you want to access information anywhere in the file, you usually have to read all the information preceding it.

A random-access textfile 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, the subtype entry would be R=$007F.

READ. This command redirects 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 accessed 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) using the ,F# parameter or a byte number using the ,B# parameter. If more than one of these parameters is used, READ first skips 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 sets the position in the file at which subsequent read and write operations will take place. The number of fields to skip over is specified by the ,F# or ,R# parameter.

WRITE. This command redirects subsequent output to an open file instead of the video screen. It works much like the READ command except in the opposite direction.

APPEND. This command opens a file and redirects subsequent output to the end of the file. The default file type is a textfile, but you can override this with the ,Ttype parameter.

FLUSH. When BASIC.SYSTEM opens a file, it allocates a file buffer for it in memory. Data written to the file is stored in this buffer and is not transferred to disk until the buffer fills up or another file block needs to be accessed. The FLUSH command forces any data stored in the buffer to be saved to 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 you use FLUSH without a pathname, all open files are flushed.

CLOSE. This command closes a file that was opened with the OPEN or APPEND command. When you close a file, its buffer is automatically flushed, and its directory entry is updated. If you use CLOSE without a pathname, all open files are closed.

Miscellaneous Commands

BYE. This command disconnects BASIC.SYSTEM and passes control to a ProDOS 8 system program by calling the QUIT command. This invokes the ProDOS 8 selector program (as discussed earlier in this chapter). The standard selector prompts you to enter the prefix and partial pathname of the next system program to run; once you provide this information, the program is executed.

CHAIN. This command transfers 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 using the ,@# parameter.

FRE. This command forces garbage collection of Applesoft string variables. This command is much faster than the one of the same name built in to the Applesoft interpreter. (See Chapter 4 of *Inside the Apple IIe* for more information on the garbage collection procedure.)

IN#. This command redirects subsequent requests for input to a peripheral card subroutine at $CnOO (where n is a slot number) or to a user-installed subroutine. If a slot number of 0 is specified, the standard keyboard input subroutine at KEYIN ($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, an IN#n command can be used to direct later requests for input to this subroutine rather than to $CnOO.

NOMON. This command is allowed but does nothing. Under DOS 3.3 it disables the display of disk commands and I/O operations; under BASIC.SYSTEM, these commands and operations are never displayed.

PR#. This command redirects subsequent output to a peripheral card subroutine at $CnOO or to a user-installed subroutine. If a slot number of 0 is specified, the standard 40-column video output subroutine at COUT1 ($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 $CnOO.

RESTORE. This command initializes the names and values of the variables in an Applesoft program to those contained in the file specified in the argument. This file must have a file type code of VAR (the type created by the STORE command).

STORE. This command saves the names and current values of all the variables in an Applesoft program to a disk file. The mnemonic for the file type code BASIC.SYSTEM assigns to the file is VAR.

BASIC.SYSTEM AND THE INPUT AND OUTPUT LINKS

Applesoft programs sometimes need to redirect input or output requests to a device in one of the Apple's expansion slots (called *ports* on the IIGS or the slotless Apple IIc). The easiest way to do this is to use 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 located anywhere in memory. The only restriction on its use is that the first byte of the new subroutine must be a 6502 CLD (clear decimal flag) instruction. To direct I/O to any such 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.

Problems can arise if you try 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 the input and output links, respectively. 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 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.)

You can avoid this problem in one of two ways. You 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 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 (nondisk) I/O operations.

Alternatively, you 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).

Any other method 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 Figure 5-1 shows, once you install BASIC.SYSTEM, it occupies the memory space from $9A00 to $BEFF in main memory. It also sets up a $400-byte (1K) general-purpose buffer that initially sits just below this area, beginning at $9600. To prevent the space above $9600 from being overwritten by Applesoft programs, BASIC.-SYSTEM sets the Applesoft HIMEM pointer to $9600; this forces Applesoft to store string variables below $9600. (HIMEM refers to the address stored in the Applesoft end-of-string pointer at $73–$74.)

The general-purpose buffer always occupies the 1K area just above HIMEM even if HIMEM changes. BASIC.SYSTEM uses it as a temporary storage area for directory blocks when it needs to catalog the disk.

BASIC.SYSTEM automatically adjusts HIMEM whenever files are opened or closed with the OPEN, APPEND, and CLOSE commands. It is not immediately obvious why a change is necessary, so let's examine how BASIC.SYSTEM manages files in a bit more detail. When BASIC.SYSTEM opens a file, it 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 it opens another file (up to eight files can be open at once), it repeats the process, meaning the new buffer fits in just below the first one. (*Exception:* If you open a file with the EXEC command, BASIC.-SYSTEM always places its buffer immediately above the highest-addressed active buffer.) When you close a file, ProDOS 8 removes the file's buffer 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 are not overwritten despite the fluctuations in HIMEM.

It is often convenient to reserve a safe area of memory where assembly-language programs may be stored without fear of being overwritten by either BASIC.SYSTEM or Applesoft itself. One such area is from $300 to $3CF in page three, but there is room for only very short programs there. Under DOS 3.3, an alternative area can be reserved simply by lowering HIMEM and storing the program between the new and old HIMEM locations. But you can't do this with BASIC.SYSTEM because of the way it manages buffers when files are opened or closed.

When you're using BASIC.SYSTEM, you can reserve a safe area above the $400-byte directory buffer beginning at HIMEM. To do this, follow these steps:

1. Close all files with the BASIC.SYSTEM CLOSE command.
2. Lower HIMEM by a multiple of $100 (256) bytes with the Applesoft HIMEM: command. (The HIMEM: command simply places the address specified in its argument directly into the HIMEM pointer.)

You must perform these steps before any Applesoft string variables have been defined since the existing Applesoft string space will be overwritten. After completing these two steps, the area from HIMEM + $400 to $99FF can be used for storing assembly-language programs without danger of their being overwritten by BASIC.SYSTEM operations.

Be very careful when using the Applesoft HIMEM: command because no checks are made to ensure the address specified in the command is an integral multiple of 256. BASIC.SYSTEM does not operate properly if HIMEM does not point to 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. Do this by placing the number of pages to be reserved in the accumulator and then calling GETBUFR; on exit, the carry flag is clear if there was enough free space available, or set if there wasn't. If all went well, the number of the first page reserved is in the accumulator. We see an example of how to use GETBUFR later in this chapter in the installation code for a user-defined command called ONLINE.

You can deallocate space reserved with GETBUFR by calling the FREEBUFR ($BEF8) subroutine. This subroutine frees up *all* buffers that GETBUFR has reserved since bootup by setting HIMEM back to its original value stored at PAGETOP ($BEFB). (You can selectively free up the most recently allocated buffers by setting PAGETOP to the page number, less 4, of the start of the buffer you want to remain.)

Whenever you reserve space above HIMEM, it is usually a good idea to modify the system bit map to indicate that the memory pages reserved are in use. If you do this, the ProDOS 8 command interpreter will not permit these pages to be used as buffer areas when ProDOS 8 commands are requested. But if you want to use part of the space as an I/O buffer when opening a file, don't mark the pages as in use; if you do, you will get an error when you try to open a file.

BASIC.SYSTEM PAGE THREE USAGE

We saw in Chapter 3 that ProDOS 8 reserves the area from $3D0 to $3EC for use by system programs like BASIC.SYSTEM. As Table 5-3 shows, BASIC.SYSTEM uses only the first six locations; these locations contain two 3-byte JMP instructions to the BASIC.SYSTEM warm-start entry point.

BASIC.SYSTEM also initializes most of the system vectors from $3ED to $3FF when it starts up. Table 5-4 shows the contents of this area of page three.

Table 5-3 ProDOS 8–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 reconnects BASIC.SYSTEM without destroying the Applesoft program in memory. Use the 3D0G 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 8 system program.

Table 5-4 Initialization of page 3 system vectors by ProDOS 8 and BASIC.SYSTEM

Vector Name	*Address*	*Contents*	*Description*
XFERLOC	$3ED–$3EE	[not initialized]	Address control passes to when XFER ($C314) is called (IIe, IIc, IIGS)
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 8 interrupt handler (see Chapter 6)

NOTE: The addresses stored at each vector location are stored with the low-order byte first.

BASIC.SYSTEM does not use the rest of page three (from $300 to $3CF), so it is a convenient area for holding short assembly-language subroutines you can call from an Applesoft program.

THE BASIC.SYSTEM GLOBAL PAGE: $BE00–$BEFF

The BASIC.SYSTEM global page occupies locations $BE00 to $BEFF, just beneath the ProDOS 8 global page. It contains several fixed-position subroutines and data areas that assembly-language programs can use to communicate easily with BASIC.-SYSTEM. For example, the global page contains entry points for executing ASCII command strings, handling user-installed commands, handling errors, and executing MLI commands. Table 5-5 is a source listing for the BASIC.SYSTEM global page.

The GOSYSTEM Subroutine

Most of the global page supports the GOSYSTEM ($BE70) subroutine that the BASIC.-SYSTEM code calls whenever it needs to execute an MLI command. On entry, GOSYS-TEM constructs a standard JSR MLI call by storing the MLI command number (passed in the accumulator) at SYSCALL ($BE85) and the address of the command's parameter table at SYSPARM ($BE86). (As Table 5-5 shows, each command BASIC.SYSTEM uses has 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 is properly invoked when the JSR MLI is actually executed.

You can use GOSYSTEM in your own assembly-language programs to execute MLI commands. 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 here]
.
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 because BASIC.SYSTEM has already reserved space for the command parameter tables in the global page. Furthermore, GOSYSTEM automatically sets up the JSR MLI/DFB CMDNUM/DA PARMTBL calling block and converts MLI error codes to the more familiar BASIC.SYSTEM error codes. We talk more about error handling in the next section.

Note, however, that you can use GOSYSTEM to execute MLI commands only from $C0 to $D3. Other commands you must execute using the standard JSR MLI technique.

Table 5-5 Source listing for the BASIC.SYSTEM global page

```
                *************************************
                *      BASIC.SYSTEM Global Page     *
                *    for BASIC.SYSTEM version 1.2   *
                *                                   *
                * Comments copyright 1985-1988      *
                * Gary B. Little                    *
                *                                   *
                * Last modified: August 26, 1988    *
                *                                   *
                *************************************

                * Note: these addresses are valid for
                *       BASIC.SYSTEM version 1.2 only!

                TXBUF2   EQU   $280
                SYSOUT   EQU   $9A2F       ;Internal output subroutine
                SYSIN    EQU   $9ABA       ;Internal input subroutine
                NODEVERR EQU   $9AEE
                ERROR    EQU   $9AF0
                PRTERR   EQU   $9F88
                PAGEGET  EQU   $A2B5
                PAGEFREE EQU   $A301
                SYNTAX   EQU   $A677
                WARMDOS  EQU   $ABF1
                DOSOUT   EQU   $B7F1       ;Character out intercept
                DOSIN    EQU   $B7F4       ;Character in intercept
                SYSCTBL  EQU   $B805       ;Table of parm table addresses
                MLIERTBL EQU   $B9EE       ;Table of MLI error codes
                BIERRTBL EQU   $BA01       ;Table of Applesoft error codes
                CALLX    EQU   $BCA8
                TXBUF    EQU   $BCBD

                MLI      EQU   $BF00

                COUT1    EQU   $FDF0       ;Video output (40 column)
                KEYIN    EQU   $FD1B       ;Keyboard input (40 column)
                COUT80   EQU   $C307       ;Video output (80 column)
                KEYIN80  EQU   $C305       ;Keyboard input (80 column)

                         ORG   $BE00

BE00: 4C F1 AB  BIENTRY  JMP   WARMDOS     ;Connect BASIC.SYSTEM I/O links
BE03: 4C 77 A6  DOSCMD   JMP   SYNTAX      ;Execute command string at $200
BE06: 4C 9E BE  EXTRNCMD JMP   XRETURN     ;User command handler

                **************************************************
                * ERROUT is called by BASIC.SYSTEM whenever a     *
                * disk error condition is detected. (The error    *
                * code -- 2..22 -- is stored in the accumulator.) *
```

```
                 * ERROUT stores the error code in ERRCODE and in  *
                 * $DE (required by Applesoft), and then if ONERR  *
                 * is active, it passes control to the Applesoft   *
                 * error-handling subroutine; if it isn't, an      *
                 * error message is printed by calling PRINTERR.   *
                 ***************************************************
BE09: 4C F0 9A   ERROUT   JMP   ERROR       ;Applesoft error handler
BE0C: 4C 88 9F   PRINTERR JMP   PRTERR      ;Print error message
BE0F: 00         ERRCODE  DFB   0           ;Error code

                 **********************************************
                 * The following table holds the addresses to *
                 * be placed in the output link whenever a    *
                 * PR#s command is entered. If a peripheral   *
                 * card is in a particular slot, the entry    *
                 * will be of the form Cs00; if no card is    *
                 * installed, the address of the subroutine   *
                 * that generates a "no device connected"     *
                 * error code is stored instead. Any address  *
                 * can be placed in the table using the       *
                 * Applesoft PRINT CHR$(4);"PR# s,A#"         *
                 * construct.                                 *
                 **********************************************
BE10: F0 FD      OUTVECT0 DA    COUT1       ;Standard video output
BE12: 00 C1      OUTVECT1 DA    $C100       ;(Assume printer card)
BE14: 00 C2      OUTVECT2 DA    $C200       ;(Assume modem card)
BE16: 00 C3      OUTVECT3 DA    $C300       ;(Assume 80-column card)
BE18: 00 C4      OUTVECT4 DA    $C400       ;(Assume mouse card)
BE1A: 00 C5      OUTVECT5 DA    $C500       ;(Assume 3.5-inch drive)
BE1C: 00 C6      OUTVECT6 DA    $C600       ;(Assume 5.25-inch drive)
BE1E: 00 C7      OUTVECT7 DA    $C700       ;(Assume RAMdisk card)

                 **********************************************
                 * The following table holds the addresses to *
                 * be placed in the output link whenever a    *
                 * IN#s command is entered. If a peripheral   *
                 * card is in a particular slot, the entry    *
                 * will be of the form Cs00; if no card is    *
                 * installed, the address of the subroutine   *
                 * that generates a "no device connected"     *
                 * error code is stored instead. Any address  *
                 * can be placed in the table using the       *
                 * Applesoft PRINT CHR$(4);"IN# s,A#"         *
                 * construct.                                 *
                 **********************************************
BE20: 1B FD      INVECT0  DA    KEYIN       ;Standard keyboard input
BE22: 00 C1      INVECT1  DA    $C100       ;(Assume printer card)
BE24: 00 C2      INVECT2  DA    $C200       ;(Assume modem card)
BE26: 00 C3      INVECT3  DA    $C300       ;(Assume 80-column card)
```

Table 5-5 Continued

```
BE28: 00 C4    INVECT4  DA   $C400      ;(Assume mouse card)
BE2A: EE 9A    INVECT5  DA   NODEVERR
BE2C: 00 C6    INVECT6  DA   $C600      ;(Assume 5.25-inch drive)
BE2E: EE 9A    INVECT7  DA   NODEVERR

               ************************************************
               * The BASIC.SYSTEM I/O links are stored here. *
               * These are the addresses control will pass   *
               * to if the input or output is not handled    *
               * internally.                                 *
               ************************************************
BE30: 07 C3    VECTOUT  DA   COUT80     ;ProDOS output link
BE32: 05 C3    VECTIN   DA   KEYIN80    ;ProDOS input link

               * Miscellaneous internal BASIC.SYSTEM parameters:

BE34: F1 B7    VDOSIO   DA   DOSOUT     ;Character out intercept
BE36: F4 B7             DA   DOSIN      ;Character in intercept

BE38: 2F 9A    VSYSIO   DA   SYSOUT     ;Internal output subroutine
BE3A: BA 9A             DA   SYSIN      ;Internal input subroutine

BE3C: 06       DEFSLT   DFB  6          ;Default slot #
BE3D: 01       DEFDRV   DFB  1          ;Default drive #

BE3E: 00       PREGA    DFB  0          ;Temporary storage for A
BE3F: 00       PREGX    DFB  0          ;Temporary storage for X
BE40: 00       PREGY    DFB  0          ;Temporary storage for Y

BE41: 00       DTRACE   DFB  0          ;bit 7=1 ==> Applesoft trace on

BE42: 00       STATE    DFB  0          ;0=direct, <>0=in program
BE43: 00       EXACTV   DFB  0          ;bit 7=1 ==> EXEC file open
BE44: 00       IFILACTV DFB  0          ;bit 7=1 ==> input file active
BE45: 00       OFILACTV DFB  0          ;bit 7=1 ==> output file active
BE46: 00       PFXACTV  DFB  0          ;bit 7=1 ==> prefix input active
BE47: 00       DIRFLG   DFB  0          ;bit 7=1 ==> dir. file active
BE48: 00       EDIRFLG  DFB  0          ;bit 7=1 ==> end of directory
BE49: 00       STRINGS  DFB  0          ;Counter for free space calc.
BE4A: 00       TBUFPTR  DFB  0          ;Character count for WRITE
BE4B: 00       INPTR    DFB  0          ;Char. count for kbd input
BE4C: 00       CHRLAST  DFB  0          ;Last character printed
BE4D: 00       OPENCNT  DFB  0          ;Number of open files (not EXEC)
BE4E: 00       EXFILE   DFB  0          ;EXEC file close flag
BE4F: 00       CATFLAG  DFB  0          ;Directory input flag

               ****************************************************
               * The following three locations will be used if   *
               * you are adding user commands to BASIC.SYSTEM.   *
```

Table 5-5 Continued

```
                 ***************************************************
BE50: 00 00      XTRNADDR DA    0          ;Address of user command handler
BE52: 00         XLEN     DFB   0          ;Length of user command - 1
BE53: 00         XCNUM    DFB   0          ;Command number in use (0=user)

                 * Notes on PBITS and FBITS:
                 *
                 * Once BASIC.SYSTEM has identified a valid command,
                 * it stores a number in PBITS and PBITS+1 that
                 * reflects the syntax of the command. It then calls
                 * the command parser, which updates FBITS and
                 * FBITS+1 to reflect the parameters actually found.
                 *
                 * 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

BE54: 00 00      PBITS   DW 0             ;Permitted parameter bits BE56:
00 00            FBITS    DW   0         ;Found parameter bits

                 ***************************************************
                 * The following table is where command parameters  *
                 * are stored during a parsing operation. The       *
                 * entries for unspecified parameters are not       *
                 * changed.                                         *
                 ***************************************************
BE58: 00 00      APARM    DA    0          ;A (address) parameter
BE5A: 00 00 00   BPARM    DS    3          ;B (byte #) parameter
BE5D: 00 00      EPARM    DA    0          ;E (end addr) parameter
BE5F: 00 00      LPARM    DW    0          ;L (length) parameter
BE61: 00         SPARM    DFB   0          ;S (slot) parameter
```

Table 5-5 Continued

```
BE62: 00        DPARM    DFB  0           ;D (drive) parameter
BE63: 00 00     FPARM    DW   0           ;F (field #) parameter
BE65: 00 00     RPARM    DW   0           ;R (record #) parameter
BE67: 00        VPARM    DFB  0           ;V (volume #) parameter
BE68: 00 00     @PARM    DW   0           ;@ (line #) parameter
BE6A: 00        TPARM    DFB  0           ;T (file type code) parameter
BE6B: 00        SLPARM   DFB  0           ;slot (for IN#, PR#) parameter
BE6C: BC BC     PATH1    DA   TXBUF-1     ;Pointer to first pathname
BE6E: 80 02     PATH2    DA   TXBUF2      ;Pointer to second pathname

                *****************************************************
                * All BASIC.SYSTEM MLI calls are routed to GOSYSTEM *
                * with the command number in the accumulator.       *
                * Prior to calling GOSYSTEM, BASIC.SYSTEM           *
                * sets up the appropriate parameter table in the    *
                * global page as required by the call. GOSYSTEM     *
                * handles all MLI calls from $C0..$D3 inclusive. If *
                * an error occurs, an Applesoft error code is       *
                * returned in A with the carry flag set.            *
                *****************************************************
BE70: 8D 85 BE  GOSYSTEM STA  SYSCALL     ;Save MLI command number
BE73: 8E A8 BC           STX  CALLX       ;Save X register
BE76: 29 1F              AND  #$1F        ;# mod 32
BE78: AA                 TAX
BE79: BD 05 B8           LDA  SYSCTBL,X ;Get address of parm table
BE7C: 8D 86 BE           STA  SYSPARM     ; (low) and save it
BE7F: AE A8 BC           LDX  CALLX       ;Restore X
BE82: 20 00 BF           JSR  MLI         ;Do the MLI call
BE85: 00        SYSCALL  DFB  0           ;MLI command # stored here
BE86: 00        SYSPARM  DFB  0           ;Address of parm table (low)
BE87: BE                 DFB  $BE         ;High address always $BE
BE88: B0 01              BCS  BADCALL     ;Branch if error
BE8A: 60                 RTS

                *******************************************
                * The BADCALL subroutine converts the MLI *
                * error code to a corresponding Applesoft *
                * error code.                             *
                *******************************************
BE8B: A2 12     BADCALL  LDX  #$12
BE8D: DD EE B9  MLIERR1  CMP  MLIERTBL,X ;Is it a "known" MLI error?
BE90: F0 05              BEQ  MLIERR2     ;Yes, so branch
BE92: CA                 DEX              ;Check all 19 possibilities
BE93: 10 F8              BPL  MLIERR1
BE95: A2 13              LDX  #$13        ;Not known, so "I/O error"

BE97: BD 01 BA  MLIERR2  LDA  BIERRTBL,X ;Convert to Applesoft error code
BE9A: AE A8 BC           LDX  CALLX       ;Restore X
BE9D: 38                 SEC              ;==> error
```

Table 5-5 Continued

```
BE9E: 60        XRETURN  RTS
BE9F: 00                 DFB   $00        ;Unused byte

                *****************************************************
                * The parameter tables for each of the MLI functions *
                * supported by BASIC.SYSTEM follow. These tables     *
                * must be filled in before calling GOSYSTEM.         *
                *****************************************************
                * Parm table for CREATE:
BEA0: 07                 DFB   $07        ;Number of parameters
BEA1: BC BC              DA    TXBUF-1    ;Pathname pointer
BEA3: C3                 DFB   $C3        ;Access code
BEA4: 00                 DFB   0          ;File type code
BEA5: 00 00              DW    $0000      ;Auxiliary type code
BEA7: 00                 DFB   0          ;Storage type code (usually 1)
BEA8: 00 00              DW    0          ;Create date
BEAA: 00 00              DW    0          ;Create time

                * Parm table for DESTROY, SET_PREFIX, GET_PREFIX:
BEAC: 01                 DFB   $01        ;Number of parameters
BEAD: BC BC              DA    TXBUF-1    ;Pathname pointer

                * Parm table for RENAME:
BEAF: 02                 DFB   $02        ;Number of parameters
BEB0: BC BC              DA    TXBUF-1    ;Old pathname pointer
BEB2: 80 02              DA    TXBUF2     ;New pathname pointer

                * Parm table for SET_FILE_INFO and GET_FILE_INFO:
BEB4: 00                 DFB   $00        ;=7 (SFI) or =10 (GFI)
BEB5: BC BC              DA    TXBUF-1    ;Pathname pointer
BEB7: 00                 DFB   $00        ;Access code
BEB8: 00                 DFB   $00        ;File type code
BEB9: 00 00              DW    $0000      ;Auxiliary type code
BEBB: 00                 DFB   $00        ;Storage type code (GFI only)
BEBC: 00 00              DW    $0000      ;Blocks used (GFI only)
BEBE: 00 00              DW    $0000      ;Modification date
BEC0: 00 00              DW    $0000      ;Modification time
BEC2: 00 00              DW    $0000      ;Create date (GFI only)
BEC4: 00 00              DW    $0000      ;Create time (GFI only)

                * Parm table for ON_LINE, SET_MARK, GET_MARK,
                * SET_EOF,GET_EOF,SET_BUF,GET_BUF:
BEC6: 02                 DFB   $02        ;Number of parameters
BEC7: 00                 DFB   $00        ;Unit or reference number
BEC8: 00                 DFB   $00        ;2-byte pointer to data buffer
BEC9: 00                 DFB   $00        ; (BUF, ON_LINE), or 3-byte
BECA: 00                 DFB   $00        ; position (MARK, EOF)

                * Parm table for OPEN:
```

Table 5-5 Continued

```
BECB: 03                 DFB   $03        ;Number of parameters
BECC: BC BC              DA    TXBUF-1    ;Pathname pointer
BECE: 00 00              DA    $0000      ;Buffer pointer (1K)
BED0: 00                 DFB   0          ;Reference number

                * Parm table for NEWLINE:
BED1: 03                 DFB   $03        ;Number of parameters
BED2: 00                 DFB   0          ;Reference number
BED3: 7F                 DFB   $7F        ;Ignore state of high bit
BED4: 0D                 DFB   $0D        ;Newline is $0D or $8D

                * Parm table for READ and WRITE:
BED5: 04                 DFB   $04        ;Number of parameters
BED6: 00                 DFB   $00        ;Reference number
BED7: 00 00              DA    $0000      ;Buffer pointer
BED9: 00 00              DW    $0000      ;Number of bytes to read/write
BEDB: 00 00              DW    $0000      ;Actual number read/written

                * Parm table for CLOSE and FLUSH:
BEDD: 01                 DFB   $01        ;Number of parameters
BEDE: 00                 DFB   0          ;Reference number

BEDF: 00                 DFB   0          ;Unused byte

BEE0: C3 CF D0           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

                *****************************************************
                * Call GETBUFR to free up "A" pages above HIMEM. If *
                * the carry flag is set upon exit, there was not    *
                * enough memory to do so; otherwise, "A" will       *
                * contain the number of the first page of the       *
                * buffer. Call FREEBUFR to remove the buffer, and   *
                * restore HIMEM to its bootup value (that value is  *
                * stored at PAGETOP).                               *
                *****************************************************
BEF5: 4C B5 A2  GETBUFR  JMP   PAGEGET    ;Reserve "A" pages above HIMEM
BEF8: 4C 01 A3  FREEBUFR JMP   PAGEFREE   ;Restore original HIMEM
BEFB: 96        PAGETOP  DFB   $96        ;HIMEM page on boot

BEFC: 00 00 00           DS    4          ;Unused bytes
BEFF: 00
```

Important: When using GOSYSTEM, be careful not to disturb the values of certain parameter table entries that BASIC.SYSTEM sets up as constants. 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 temporarily change any of these parameters, save their values first, then restore them after the GOSYSTEM call.

In the following section we discuss some of the other important areas of the BASIC.SYSTEM global page.

BASIC.SYSTEM ERROR HANDLING

If a call to GOSYSTEM results in a system error, GOSYSTEM branches to BADCALL ($BE8B), a subroutine that converts the MLI error code in the accumulator into a BASIC.SYSTEM (Applesoft) error code. Table 5-6 shows the correspondence between a given MLI code and a BASIC.SYSTEM code.

Note that only 19 MLI error codes are specifically dealt with by BADCALL. It automatically converts all others to error code 8 ("I/O Error"). Moreover, four 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.

After BASIC.SYSTEM converts the MLI error code, it calls ERROUT ($BE09) to handle the error. This subroutine first stores the error code in ERRCODE ($BE0F) and at $DE (the Applesoft interpreter expects to find an error code at $DE) and then checks if the Applesoft ONERR GOTO error-trapping feature is active. If it is, control passes to the internal Applesoft error-handling subroutine. If it isn't, BASIC.SYSTEM calls PRINTERR ($BE0C) to print the error message corresponding to the error code (see Table 5-6).

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. But you must ensure 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 error code conversion.

EXECUTING DISK COMMAND STRINGS FROM ASSEMBLY LANGUAGE

An assembly-language program can use the DOSCMD ($BE03) subroutine in the BASIC.SYSTEM global page to interpret and execute a standard BASIC.SYSTEM

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

disk command stored in the Apple input buffer at $200 as an ASCII string followed by a carriage return code ($8D). DOSCMD is effective only when an Applesoft program is actually running, so an Applesoft program must use the CALL command to access the assembly-language program.

(Under DOS 3.3, assembly-language programs can execute disk commands by sending code $04 (Control-D) to the standard character output subroutine, COUT ($FDED), followed by the ASCII codes for the command and a carriage return code. BASIC.SYSTEM does not support this technique.)

DOSCMD can execute most, but not all, BASIC.SYSTEM disk commands. The commands it does *not* handle properly are - (dash), RUN, LOAD, CHAIN, READ, WRITE, APPEND, and EXEC. When you call DOSCMD to execute a command it can handle, it returns a BASIC.SYSTEM error code in the accumulator. If no error occurred, the code is 0, and the carry flag is clear. If an error did occur, the carry flag is set. Handle an error condition by calling ERROUT ($BE09) or PRINTERR ($BE0C) (as described in the previous section) or by passing control to your own error-handling code.

Important: Just before a program using DOSCMD ends, it must clear the carry flag and execute a CLC instruction. If it ends with the carry flag set, the Applesoft program that called it may not work properly.

ADDING COMMANDS TO BASIC.SYSTEM

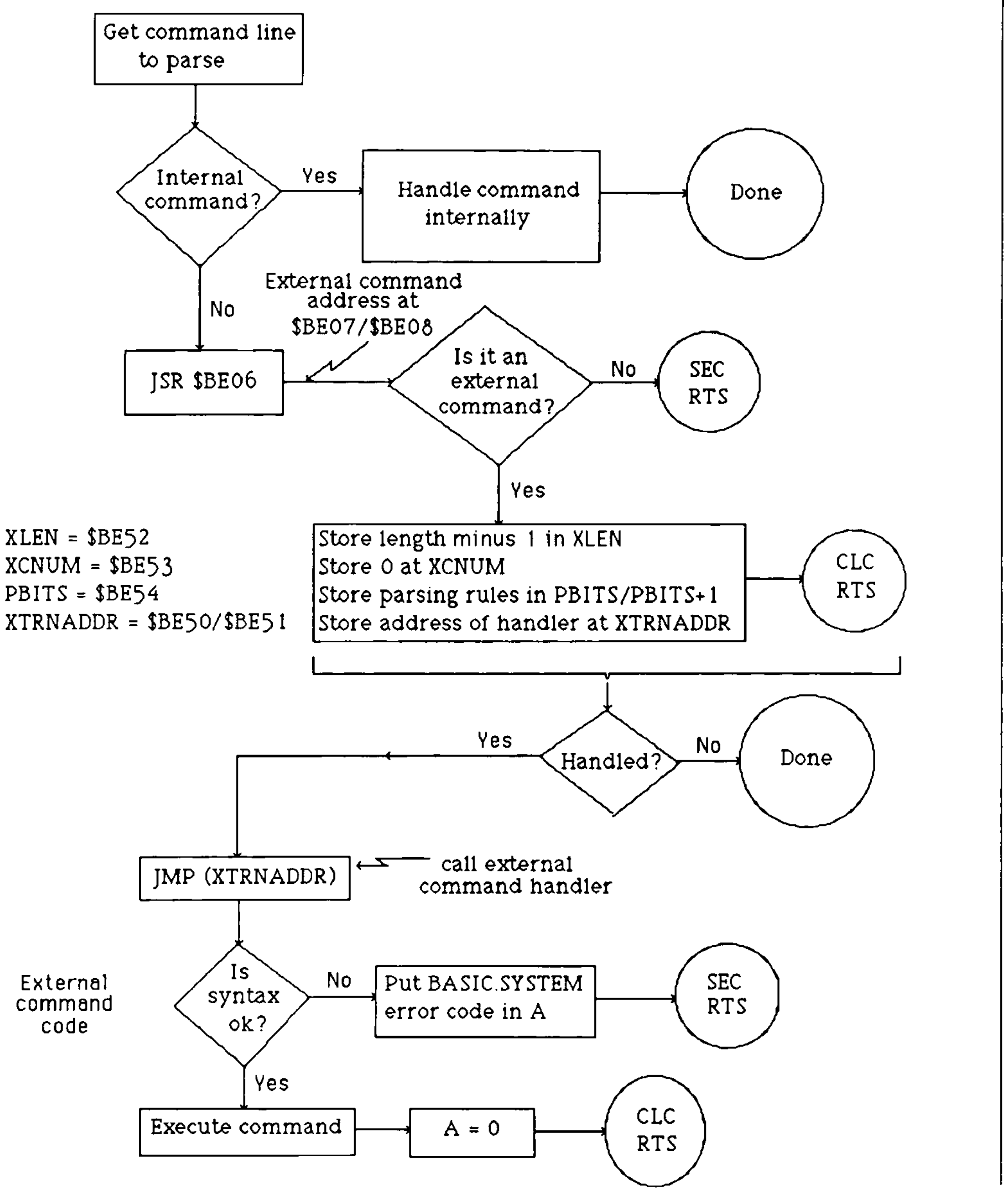
One of the best features of BASIC.SYSTEM is its support of user-defined external commands. To see how to extend BASIC.SYSTEM's standard command set, let's take a look at exactly what happens when BASIC.SYSTEM encounters a string of characters that may represent a valid command. Figure 5-2 shows a flowchart of this procedure.

The first thing BASIC.SYSTEM does is check if one of its 32 standard commands has been specified (CATALOG, OPEN, WRITE, and so on). If one has been, it handles it internally.

But if the command can't be identified, BASIC.SYSTEM does not immediately return an error code; rather, it calls a subroutine in its global page, EXTRNCMD ($BE06), to see if a user-installed external command handler will claim the command. (The handler's address is always stored at $BE07 and $BE08.) If no external command handler has been installed, EXTRNCMD simply jumps to a "do-nothing" RTS instruction at XRETURN ($BE9E). If the external command handler does not claim the command, and if the command was issued from within a program, a BASIC.SYSTEM syntax error condition occurs. If, on the other hand, the command was entered in Applesoft command mode, it is passed on for consideration by the Applesoft interpreter. Only if the interpreter does not recognize it does an Applesoft syntax error occur.

Let's assume an external command handler has been installed so that a call to EXTRNCMD will pass control to it. Such a handler first executes a CLD instruction, which Apple says it will use as an identification byte in future versions of BASIC.SYSTEM. The handler then determines 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

Figure 5-2 A flowchart showing how BASIC.SYSTEM executes external commands


Get command line to parse
Internal command?
Yes
Handle command internally
Done
No
External command address at $BE07/$BE08
JSR $BE06
Is it an external command?
No
SEC RTS
Yes
XLEN = $BE52
XCNUM = $BE53
PBITS = $BE54
XTRNADDR = $BE50/$BE51
Store length minus 1 in XLEN
Store 0 at XCNUM
Store parsing rules in PBITS/PBITS+1
Store address of handler at XTRNADDR
CLC RTS
Yes
Handled?
No
Done
JMP (XTRNADDR)
call external command handler
External command code
Is syntax ok?
No
Put BASIC.SYSTEM error code in A
SEC RTS
Yes
Execute command
A = 0
CLC RTS

buffer beginning at $200 in ASCII form with the high bit of each code set to 1.) If they don't match, the subroutine must end with the carry flag set to indicate that it did not claim the command.

If the handler detects the correct command, the handler can do one of two things. 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 execute the command. Alternatively, if all the possible parameters are capable of being recognized by BASIC.SYSTEM, the handler can ask BASIC.SYSTEM to 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 it detects an error, BASIC.SYSTEM handles the error itself. Table 5-1 shows the command line parameters supported by BASIC.SYSTEM and the range of values that they can take on.

With three exceptions, each bit in PBITS and PBITS + 1 is a flag indicating 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 command 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 (for 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 command 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 selects the file type code
        for a DIR file ($0F).
bit 1   A second pathname is required (for example,
        RENAME). Two pathnames must be specified, or
        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	**($BE55)**
bit 7	The A parameter is allowed.
bit 6	The B parameter is allowed.
bit 5	The E parameter is allowed.
bit 4	The L parameter is allowed.
bit 3	The @ parameter is allowed.
bit 2	The S and D parameters are allowed. The S (slot) and D (drive) parameters must correspond to an existing disk drive if preceded by a filename; if preceded by a slot number specification (see bit 6 of PBITS), they do not. If the S and D parameters are allowed, but not specified, their values default to those stored at DEFSLT ($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 drive is fetched and used to create a full pathname whenever a filename or 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	The F parameter is allowed.
bit 0	The R parameter is allowed.

(One other parameter is always allowed and always parsed: the V (volume) parameter. BASIC.SYSTEM commands tolerate this parameter but do not use it; it has been included to maintain compatibility with DOS 3.3 commands that do use it.)

The descriptions for PBITS and PBITS + 1 apply 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 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 1 in XLEN ($BE52), store 0 (the code number for an external command) in XCNUM ($BE53), and then store at XTRNADDR ($BE50–$BE51) the address control is to pass to 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 according to 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

Table 5-7 BASIC.SYSTEM parameter table[a]

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[b]
$BE68–$BE69	@PARM	@ (line #) parameter[b]
$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

NOTES:
[a]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.
[b]A bug in BASIC.SYSTEM (versions 1.1 and 1.2) causes the V parameter to be stored in @PARM rather than VPARM (as shown). This means 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.

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). (Table 5-7 describes a BASIC.SYSTEM version 1.1 and 1.2 bug that hinders the proper parsing of a command line that uses both the V and @ parameters.)

Note that the first pathname parsed from a command line is stored in a buffer pointed to by VPATH1 ($BE6C) and the second is stored in a buffer pointed to by VPATH2 ($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 an external command handler can use GOSYSTEM to perform MLI commands without first having to modify these pointers.

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 a zero in the accumulator 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 (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

In this section, we see how to design and install the handler for a new BASIC.-SYSTEM command called ONLINE. This command displays the names of any, or all, of the disk volumes currently available to the system. ONLINE is useful if you habitually forget the name of a disk microseconds after putting it into a disk drive.

The syntax of the ONLINE command is

```
ONLINE [,S#] [,D#]
```

where the brackets mean the enclosed parameter (slot number or drive number) is optional. If a specific slot or drive number is specified, only the name of the volume for the corresponding disk device is displayed. But if both parameters are omitted, the volume names for all disk devices are displayed. The ONLINE command can be typed in while in Applesoft command mode, or it can be executed within a program using a PRINT CHR$(4);"ONLINE" statement.

Table 5-8 shows the ONLINE installation program, which is executed with the BRUN command. The first part of the program installs 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

Table 5-8 Adding the ONLINE command to BASIC.SYSTEM

```
                 *************************************
                 *    BASIC.SYSTEM "ONLINE" COMMAND   *
                 *                                    *
                 *         ONLINE [,Sn] [,Dn]         *
                 *                                    *
                 * Copyright 1985-1988 Gary B. Little *
                 *                                    *
                 * Last modified: August 26, 1988     *
                 *                                    *
                 *************************************
                 SBLOCK   EQU  $3C        ;Parameters for block move
                 EBLOCK   EQU  $3E
                 FBLOCK   EQU  $42
                 HIMEM    EQU  $73        ;Use this as ON_LINE buffer

                 IN       EQU  $200       ;Command input buffer

                 EXTRNCMD EQU  $BE06      ;External command JMP opcode
                 ERROUT   EQU  $BE09      ;Error handler
                 XTRNADDR EQU  $BE50      ;Start of external cmd handler
                 XLEN     EQU  $BE52      ;External cmd name length (-1)
                 XCNUM    EQU  $BE53      ;Command # (0 for external)
                 PBITS    EQU  $BE54      ;Command parameter bits
                 FBITS    EQU  $BE56      ;Parameters found in parse
                 VSLOT    EQU  $BE61      ;Slot parameter specified
                 VDRIV    EQU  $BE62      ;Drive parameter specified
                 GETBUFR  EQU  $BEF5      ;Get a free space

                 MLI      EQU  $BF00      ;Entry point to MLI

                 CROUT    EQU  $FD8E      ;Print a CR
                 COUT     EQU  $FDED      ;Character output subroutine
                 MOVE     EQU  $FE2C      ;Block move subroutine

                          ORG  $2000

                 * Calculate # of pages that we need to reserve:

2000: 38                  SEC
2001: A9 22               LDA  #>END
2003: E9 21               SBC  #>CMDCODE
2005: 8D 74 20            STA  PAGES
2008: EE 74 20            INC  PAGES

200B: AD 74 20            LDA  PAGES      ;Reserve the pages for the
200E: 20 F5 BE            JSR  GETBUFR    ; command handler
2011: 90 05               BCC  INSTALL    ;Carry clear if OK

2013: A9 0E               LDA  #14        ;"PROGRAM TOO LARGE" error
```

Table 5-8 Continued

```
2015: 4C 09 BE          JMP   ERROUT

2018: 8D 75 20  INSTALL STA   PGSTART    ;Save starting page #

                * Install the new command handler:

201B: AD 07 BE          LDA   EXTRNCMD+1 ;Set up link to
201E: 8D 26 21          STA   NEXTCMD+1  ; existing external command
2021: AD 08 BE          LDA   EXTRNCMD+2
2024: 8D 27 21          STA   NEXTCMD+2

                *****************************************
                * Install the external command handler *
                * by storing its address after the     *
                * JMP at EXTRNCMD.                     *
                *****************************************

2027: A9 00             LDA   #0
2029: 8D 07 BE          STA   EXTRNCMD+1
202C: AD 75 20          LDA   PGSTART
202F: 8D 08 BE          STA   EXTRNCMD+2

                * Relocate the code:

2032: AD 75 20          LDA   PGSTART    ;Get new page #
2035: 8D 0F 21          STA   CMDCODE+$0F
2038: 8D 1A 21          STA   CMDCODE+$1A
203B: 8D 32 21          STA   CMDCODE+$32
203E: 8D 49 21          STA   CMDCODE+$49
2041: 8D 4E 21          STA   CMDCODE+$4E
2044: 8D 55 21          STA   CMDCODE+$55
2047: 8D 6F 21          STA   CMDCODE+$6F
204A: 8D 75 21          STA   CMDCODE+$75
204D: 8D 8A 21          STA   CMDCODE+$8A
2050: 8D A4 21          STA   CMDCODE+$A4
2053: 8D D5 21          STA   CMDCODE+$D5

                * Set up parameters for block move to final location:

2056: A9 00             LDA   #<CMDCODE
2058: 85 3C             STA   SBLOCK
205A: A9 21             LDA   #>CMDCODE
205C: 85 3D             STA   SBLOCK+1

205E: A9 03             LDA   #<END
2060: 85 3E             STA   EBLOCK
2062: A9 22             LDA   #>END
2064: 85 3F             STA   EBLOCK+1

```

Table 5-8 Continued

```
2066: A9 00                LDA   #0
2068: 85 42                STA   FBLOCK
206A: AD 75 20             LDA   PGSTART
206D: 85 43                STA   FBLOCK+1

206F: A0 00                LDY   #0
2071: 4C 2C FE             JMP   MOVE       ;Move it!

2074: 00          PAGES    DS    1          ;Length of command handler
2075: 00          PGSTART  DS    1          ;Starting page of cmd handler

2076: 00 00 00             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
20F1: 00 00 00 00 00 00 00 00
20F9: 00 00 00 00 00 00 00

                  CMDCODE  EQU   *

                  ************************************
                  * This is the command checker. It  *
                  * scans the input buffer to see    *
                  * if the command has been entered. *
                  ************************************
2100: D8                   CLD
2101: A0 00                LDY   #0
2103: A2 00                LDX   #0
2105: BD 00 02    CHKCMD   LDA   IN,X       ;Get command character
2108: E8                   INX
2109: C9 A0                CMP   #$A0       ;Is it a blank?
210B: F0 F8                BEQ   CHKCMD     ;If it is, ignore it
210D: D9 EE 21             CMP   CMDNAME,Y  ;Same as our command?
2110: F0 0B                BEQ   CHKCMD1    ;Yes, so branch
2112: C9 E0                CMP   #$E0       ;Lowercase?
2114: 90 0E                BCC   NOTFOUND   ;No, so branch
2116: 29 DF                AND   #$DF       ;Convert to uppercase
```

Table 5-8 Continued

```
2118: D9 EE 21            CMP  CMDNAME,Y  ;OK now?
211B: D0 07               BNE  NOTFOUND   ;No, so branch
211D: C8         CHKCMD1  INY
211E: C0 06               CPY  #CMDLEN-CMDNAME ;At end?
2120: D0 E3               BNE  CHKCMD     ;No, so branch
2122: F0 04               BEQ  SETRULES   ;Yes, so branch

2124: 38         NOTFOUND SEC             ;Set carry to indicate failure
2125: 4C 00 00   NEXTCMD  JMP  $0000      ;(Fill in when installed)

2128: 88         SETRULES DEY
2129: 8C 52 BE            STY  XLEN       ;Store command length-1

212C: A9 51               LDA  #<EXECUTE  ;Put address of command handler
212E: 8D 50 BE            STA  XTRNADDR   ; into XTRNADDR
2131: A9 21               LDA  #>EXECUTE
2133: 8D 51 BE            STA  XTRNADDR+1

2136: A9 00               LDA  #0
2138: 8D 53 BE            STA  XCNUM      ;External cmd number = 0

                 * Set up string parsing rules:

213B: A9 10               LDA  #$10       ;Pathname is optional
213D: 8D 54 BE            STA  PBITS
2140: A9 04               LDA  #$04       ;Slot, drive allowed
2142: 8D 55 BE            STA  PBITS+1

2145: A5 73               LDA  HIMEM      ;Set ON_LINE buffer (at least
2147: 8D EC 21            STA  BUFFER     ; 256 bytes) to free area
214A: A5 74               LDA  HIMEM+1    ; beginning at HIMEM
214C: 8D ED 21            STA  BUFFER+1

214F: 18                  CLC             ;Clear carry to indicate success
2150: 60                  RTS

                 * BASIC.SYSTEM comes here after it has
                 * successfully parsed the command line:

2151: A9 00      EXECUTE  LDA  #0
2153: 8D EB 21            STA  UNITNUM    ;(Assume all volumes)
2156: AD 57 BE            LDA  FBITS+1    ;Examine result of parse
2159: 29 04               AND  #$04       ;Slot, drive specified?
215B: F0 13               BEQ  DOCALL     ;No, so check everything
215D: AD 61 BE            LDA  VSLOT      ;Get slot # specified
2160: 0A                  ASL
2161: 0A                  ASL
2162: 0A                  ASL
2163: 0A                  ASL             ;Slot * 16
```

```
2164: AE 62 BE            LDX   VDRIV      ;Get drive # specified
2167: E0 02               CPX   #2         ;Drive 2?
2169: D0 02               BNE   SAVEUN     ;No, so branch
216B: 09 80               ORA   #$80       ;Set "drive 2" bit
216D: 8D EB 21   SAVEUN   STA   UNITNUM    ;Store slot, drive as unit num

2170: 20 00 BF   DOCALL   JSR   MLI
2173: C5                  DFB   $C5        ;ON_LINE call
2174: EA 21               DA    OLPARM     ;Address of parm table

2176: 20 8E FD            JSR   CROUT
2179: A0 00               LDY   #0
217B: 98         SCAN     TYA
217C: 48                  PHA
217D: B1 73               LDA   (HIMEM),Y  ;Get slot, drive + length
217F: F0 61               BEQ   SCAN2      ;If $00, then all done
2181: 29 0F               AND   #$0F       ;Isolate length bits
2183: F0 4E               BEQ   NEXTNAME   ;If 0, then must be error
2185: 48                  PHA

2186: A2 00               LDX   #0
2188: BD F4 21   PRTMSG1  LDA   SLOTMSG,X  ;Print slot #
218B: F0 06               BEQ   PRTNUM1
218D: 20 ED FD            JSR   COUT
2190: E8                  INX
2191: D0 F5               BNE   PRTMSG1

2193: B1 73      PRTNUM1  LDA   (HIMEM),Y  ;Get slot, drive + length
2195: 29 70               AND   #$70       ;Isolate slot bits
2197: 4A                  LSR
2198: 4A                  LSR
2199: 4A                  LSR
219A: 4A                  LSR              ;We now have slot #
219B: 09 B0               ORA   #$B0       ;Convert to ASCII digit
219D: 20 ED FD            JSR   COUT

21A0: A2 00               LDX   #0
21A2: BD FA 21   PRTMSG2  LDA   DRIVEMSG,X ;Print drive #
21A5: F0 06               BEQ   PRTNUM2
21A7: 20 ED FD            JSR   COUT
21AA: E8                  INX
21AB: D0 F5               BNE   PRTMSG2

21AD: A2 B1      PRTNUM2  LDX   #$B1       ;Assume drive 1
21AF: B1 73               LDA   (HIMEM),Y
21B1: 10 02               BPL   PSKIP      ;Branch if drive 1
21B3: A2 B2               LDX   #$B2       ;Must be drive 2
21B5: 8A         PSKIP    TXA
21B6: 20 ED FD            JSR   COUT
```

Table 5-8 Continued

```
21B9: A9 BA               LDA  #":
21BB: 20 ED FD            JSR  COUT
21BE: A9 A0               LDA  #$A0
21C0: 20 ED FD            JSR  COUT

21C3: 68                  PLA
21C4: AA                  TAX
21C5: C8         PRTNAME  INY
21C6: B1 73               LDA  (HIMEM),Y  ;Get next character in name
21C8: 09 80               ORA  #$80       ;Set high bit
21CA: 20 ED FD            JSR  COUT       ; and display it
21CD: CA                  DEX
21CE: D0 F5               BNE  PRTNAME    ;Branch until done
21D0: 20 8E FD            JSR  CROUT

21D3: AD EB 21   NEXTNAME LDA  UNITNUM    ;Was only one volume specified?
21D6: D0 0A               BNE  SCAN2      ;Yes, so branch

21D8: 68                  PLA
21D9: 18                  CLC
21DA: 69 10               ADC  #16        ;Move to next name
21DC: A8                  TAY
21DD: C0 E0               CPY  #224       ;At end of table?
21DF: D0 9A               BNE  SCAN       ;No, so branch
21E1: 48                  PHA

21E2: 68         SCAN2    PLA
21E3: 20 8E FD            JSR  CROUT
21E6: 18                  CLC             ;CLC ==> no error
21E7: A9 00               LDA  #0         ;Error code = 0
21E9: 60                  RTS

21EA: 02         OLPARM   DFB  2          ;Two parameters
21EB: 00         UNITNUM  DFB  0          ;Unit number (DSSS0000)
21EC: 00 00      BUFFER   DA   $0000      ;Device buffer

21EE: CF CE CC   CMDNAME  ASC  "ONLINE"   ;External command name
21F1: C9 CE C5
                 CMDLEN   EQU  *

21F4: D3 CC CF   SLOTMSG  ASC  "SLOT ",00
21F7: D4 A0 00
21FA: AC A0 C4   DRIVEMSG ASC  ", DRIVE ",00
21FD: D2 C9 D6 C5 A0 00

                 END      EQU  *
```

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 passed to it. This means control always daisy-chains down to a previously installed external command handler so that it will have a chance to claim the command.

The GETBUFR ($BEF5) subroutine is used to locate a "safe" buffer large enough to store the command handler. It is called with the number of pages required in the accumulator (1). If we run out of room, the carry flag will be set, and a "program too large" error message will be printed by calling ERROUT ($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 its 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 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 automatically patch the code, we chose to "manually" patch it 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 recalculate which addresses must be patched and make the necessary changes to the installation code.

The code is moved into place by using 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, 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 postparsing 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, slot and drive 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 ("no error"), and an RTS returns control to BASIC.SYSTEM.

BASIC.SYSTEM then parses the command line according to the instructions in PBITS, updates FBITS ($BE56) and FBITS+1 ($BE57) to indicate 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 and drive were specified. If so, the slot and drive specified are retrieved from VSLOT ($BE61) and VDRIV ($BE62) and used to form the unit number required by the 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 ON_LINE command has been executed, the names of the active volumes are stored in the buffer beginning at HIMEM. (See the discussion of ON_LINE in Chapter 4 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 see how GS/OS and ProDOS 8 react to and handle interrupt signals generated by I/O devices. GS/OS and ProDOS 8 both let you install assembly-language subroutines to service sources of interrupts. They also define rules these subroutines must follow to ensure they will function smoothly together. In particular, the rules dictate the method an interrupt-handling subroutine must use to indicate whether it serviced the interrupt.

Before we begin, we should review the concept of an interrupt. An interrupt is an electrical signal an I/O device sends to the microprocessor in an attempt to get its immediate and undivided attention. The signal is sent down a special line connected between a specific pin on the expansion slot connector used by the interrupting device and the IRQ (*interrupt request*) pin on the microprocessor. (On the Apple IIc and IIGS, equivalent connections are made between the microprocessor and each built-in I/O device capable of interrupting the system.)

An I/O device typically generates an interrupt signal when it has new data to be read or when it is ready to receive more data. When the microprocessor detects an active IRQ signal, it completes the current instruction, stops executing the main program, and then passes control to an interrupt-handling subroutine. This subroutine (installed by the operating system or the application) 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 it had never been disturbed.

The advantage of using an interrupt scheme like this to control I/O devices is that it is the most efficient one for handling asynchronous I/O operations (that is, operations that can occur at any time). If interrupts were not available, a program would have to waste a lot of time frequently polling each I/O device in the system to ensure that incoming data was not lost or that outgoing data was being pushed out as quickly as possible. This is comparable to picking up a telephone without a ringer every few seconds to see if anyone is calling in. By adding the ringer (the interrupt signal), you can go about your normal duties until the phone 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 usually available on three of the most common I/O devices: the clock, the asynchronous serial interface, and the mouse.

Clock—A clock device is able to keep track of the time and date without the assistance of the 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 computer is off. Most clock cards generate interrupts at regular intervals: every second, minute, or hour.

Asynchronous serial interface—An asynchronous serial interface is most commonly used to link the computer to printers and modems. It can be told to generate interrupts whenever it is ready to send out a character or whenever it receives a character.

Mouse—A mouse is an input device that is normally capable of generating interrupts when it is moved or its button is pressed.

REACTING TO INTERRUPTS

It is important to realize that the IRQ interrupt signal is *maskable*. In other words, it is possible for a program to instruct the microprocessor to ignore an active IRQ interrupt signal. It can do this by executing an SEI (set interrupt disable flag) instruction. (The interrupt disable flag is a bit in the microprocessor status register.) If interrupts are disabled like this, the main program running in the system won't be disturbed. Time-critical operations, like disk reads and writes, cannot be interrupted without loss of data, so interrupts are always disabled first.

The instruction that causes the microprocessor to respond to IRQ interrupts is CLI (clear interrupt disable flag). An application should clear the interrupt disable flag whenever possible so that it will perform smoothly in an environment in which interrupting devices may be active.

When the microprocessor receives an IRQ signal, it immediately pushes the contents of the program counter register and the status register on the stack. If the processor is a 6502 (or, on the IIGS, a 65816 in 6502 emulation mode), it passes control to a low-level interrupt handler whose address is stored at $FFFE–$FFFF (low-order byte first). If the processor is in 65816 native mode, the handler's address is stored at $FFEE–$FFEF in bank $00.

The low-level interrupt handler is in the firmware ROM on any Apple II. On models prior to the Apple IIGS, its main duty is to pass control to a high-level interrupt handler whose address is stored in the user-definable interrupt vector at $03FE and $03FF (low-order byte first). On the Apple IIGS, the low-level handler actually tries to process interrupts from built-in devices and passes control to the user-definable interrupt vector only if it is unable to do so.

A properly designed high-level interrupt handler should perform the following chores in the following order:

- Save the current values in the A, X, and Y registers and all information about the current machine state.
- Clear the source of the interrupt. (It usually does this by reading the status registers 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, and restore the same machine state.
- End with an RTI (return from interrupt) instruction.

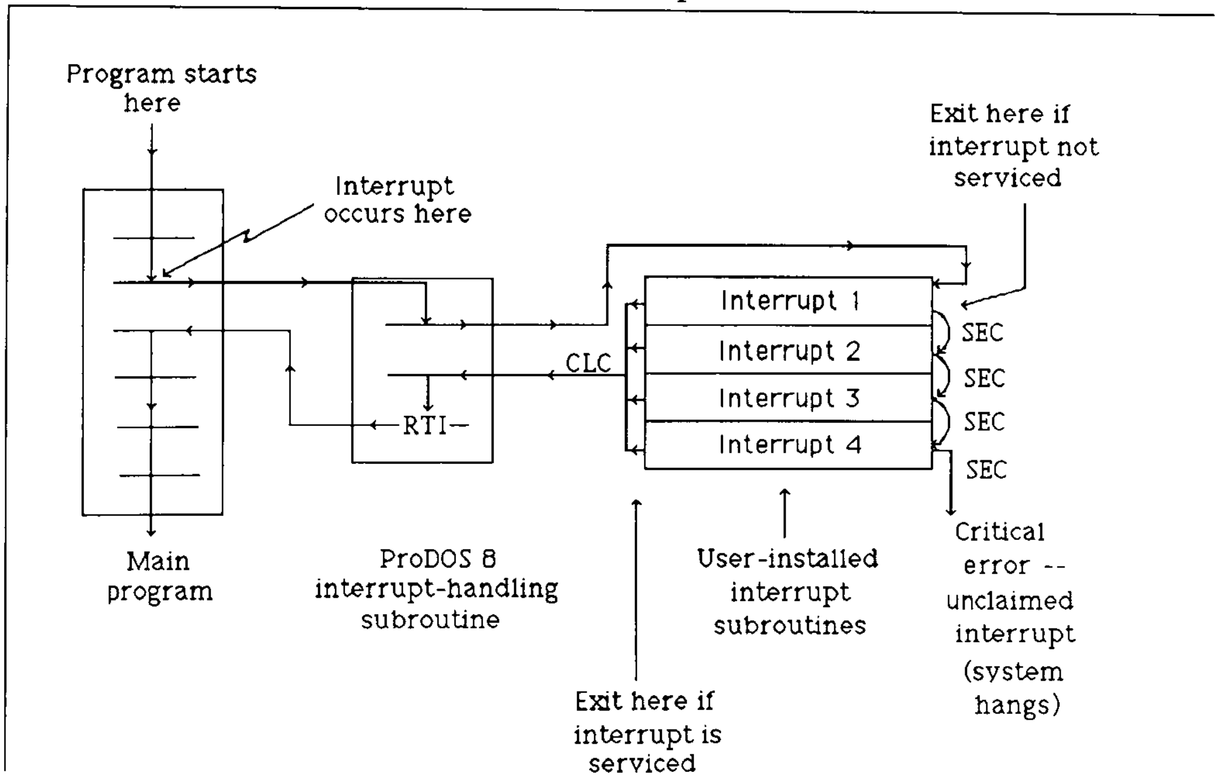
When ProDOS 8 is active, the user-definable interrupt vector points to a general-purpose interrupt handler within the main body of the operating system called the *interrupt dispatcher*. When GS/OS is active, the vector points to a similar dispatcher which manages ProDOS 16-style interrupt handlers. GS/OS-style interrupt handlers actually bind to the system at the low-level firmware level; control never passes to the user-definable interrupt vector unless the interrupt is unclaimed. GS/OS-style interrupt handlers are added to the system with the BindInt command.

The ProDOS 8 interrupt dispatcher contains no specific code for identifying and servicing 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 polls each member in a group of user-installed interrupt subroutines, the addresses of which are stored in an internal interrupt vector table. These subroutines are integrated into the system with the ProDOS 8 ALLOC_INTERRUPT command.

Figure 6-1 shows the events that take place when an interrupt occurs under ProDOS 8. The interrupt dispatcher takes over and calls the first subroutine whose address it finds in the interrupt vector table. This subroutine will either recognize and claim the interrupt or not. If it does, the operating system restores all registers and returns to the interrupted program. If it doesn't, the operating system tries again by calling the next subroutine whose address is in the interrupt vector table. (The operating system examines the state of the carry flag to determine if the interrupt was claimed; if it was claimed, the carry flag comes back cleared.) This process repeats until the interrupt is claimed, at which point the interrupt dispatcher returns control to the interrupted application by executing an RTI instruction. If none of the installed subroutines claim the interrupt, a critical error occurs and the system hangs.

The advantage of using a dispatching scheme like this to handle interrupts is that it allows for the development of interrupt-handling subroutines that are specific to only one device. That is, a subroutine need not concern itself with handling mouse, clock, serial, and "you-name-it" interrupts all at once. If the operating system rules are followed, you can easily install a mouse interrupt subroutine from one manufacturer

Figure 6-1 How ProDOS 8 handles interrupts


Program starts here
Interrupt occurs here
Exit here if interrupt not serviced
Interrupt 1
Interrupt 2
Interrupt 3
Interrupt 4
CLC
RTI
SEC
SEC
SEC
SEC
Main program
ProDOS 8 interrupt-handling subroutine
User-installed interrupt subroutines
Critical error -- unclaimed interrupt (system hangs)
Exit here if interrupt is serviced

and a clock interrupt subroutine from another and they should work properly together. (See Eyes and Lichty's *Programming the 65816* for detailed information on how the 6502 and 65816 microprocessors react to interrupt signals.)

INTERRUPTS AND PRODOS 8

The ProDOS 8 general-purpose interrupt-handling subroutine (stored in the user IRQ vector at $03FE–$03FF) did not work flawlessly in the first versions of ProDOS 8; the one used in the newest versions of ProDOS 8 do. The moral is to always use the most current version of ProDOS 8 if you want the system to work smoothly with interrupts.

You use ALLOC_INTERRUPT to store the address of an interrupt-handling subroutine at the next available location in an 8-byte interrupt vector table in the ProDOS 8 global page beginning at $BF80. (A dummy $0000 address is stored in the table if a vector is unused.) Table 6-1 lists all the global page locations used by the ProDOS 8 interrupt-handling subroutine.

Table 6-1 Global page data areas used by the ProDOS 8 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

The user-installed interrupt subroutine must adhere to the following rules:

- Its first instruction must be CLD.
- If the interrupt was not generated by its device, it must set the carry flag (with an SEC instruction) and exit.
- If its device is the source of the interrupt, it must claim the interrupt by performing the necessary I/O operation, clear the interrupt condition (usually by reading the device status), clear the carry flag with CLC, and exit.
- It must exit with all soft switches in the states they were in on entry. Most of these switches are used for memory bank switching or for controlling video display modes. (See Appendix III of *Inside the Apple IIe.*)
- The subroutine must end with an RTS instruction (*not* an RTI instruction). The ProDOS 8 interrupt handler executes the necessary RTI instruction.

There is no need for such a subroutine to save and restore the microprocessor's registers. The main ProDOS 8 interrupt-handling subroutine automatically does this for you. Two other nice features of the ProDOS 8 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 convenient 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 enough for even the most complex subroutines.

The program in Table 6-2 (MOUSE.MOVE) shows how to properly install an interrupt-handling subroutine in a ProDOS 8 environment. To be able to run this specific example, you must be using an Apple IIc with the Apple Mouse option, an Apple IIe (or II Plus) with an Apple Mouse card installed in slot 4, or an Apple IIGS with its built-in mouse. The program assumes the mouse firmware is in slot 4; if it's not, change the SLOT EQU 4 directive to reflect the actual slot. (The mouse firmware is in slot 7 of the IIc Plus and the memory expandable version of the IIc, but it is in slot 4 of earlier models.)

MOUSE.MOVE directs 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 just slows down by the time it takes to service the interrupt.

The first thing MOUSE.MOVE does is install the address of the interrupt handler (IRQHNDL) in the ProDOS 8 interrupt vector table using the ALLOC_INTERRUPT command. If an error occurs, the program branches to ERROR and enters the system Monitor. (An error occurs only 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 used are indicated at the beginning of the program.

MOUSER is the standard subroutine the programs calls to execute a mouse subroutine. It is responsible for setting up the correct subroutine address and placing the correct numbers in the microprocessor registers before passing control to the mouse firmware.

When the mouse is moved, an interrupt occurs, and ProDOS 8 quickly calls 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

Table 6-2 MOUSE.MOVE, a program that handles mouse movement interrupts

```
                  *************************************
                  *              MOUSE.MOVE           *
                  *  Mouse Movement Interrupt Handler  *
                  *                                   *
                  * Copyright 1985-1988 Gary B. Little *
                  *                                   *
                  * Last modified: August 26, 1988    *
                  *                                   *
                  *************************************
                  SLOT     EQU   4          ;Slot number of mouse card

                  MLI      EQU   $BF00      ;Entry point to ProDOS MLI

                  MTABLE   EQU   SLOT*$100+$C000+$12 ;Start of ROM table

                  * Mouse subroutine numbers:
                  SETM     EQU   0          ;Set mouse mode
                  SERVEM   EQU   1          ;Service mouse interrupt
                  READM    EQU   2          ;Read mouse
                  INITM    EQU   7          ;Initialize the mouse

                  COUT     EQU   $FDED      ;Standard output

                           ORG   $300

0300: 4C 06 03             JMP   ENABLE     ;CALL 768 to enable
0303: 4C 22 03             JMP   DISABLE    ;CALL 771 to disable

                  * Install the interrupt handler:

0306: 78          ENABLE   SEI              ;Interrupts off for this

0307: A9 02                LDA   #2
0309: 8D 39 03             STA   AIPARMS    ;Stuff correct parm count
030C: 20 00 BF             JSR   MLI
030F: 40                   DFB   $40        ;ALLOC_INTERRUPT
0310: 39 03                DA    AIPARMS
0312: B0 29                BCS   ERROR

                  * Prepare the mouse:

0314: A2 07                LDX   #INITM
0316: 20 5B 03             JSR   MOUSER     ;Initialize the mouse

0319: A2 00                LDX   #SETM
031B: A9 03                LDA   #$03       ;(Movement interrupt mode)
031D: 20 5B 03             JSR   MOUSER     ;Set the mouse mode

0320: 58                   CLI              ;Enable 6502 interrupts
```

Table 6-2 Continued

```
0321: 60                 RTS

                * Here's the code to "remove" the interrupt:
0322: 78        DISABLE  SEI                ;Interrupts off for this

0323: A2 00              LDX   #SETM
0325: A9 00              LDA   #0           ;(Turn mouse off)
0327: 20 5B 03           JSR   MOUSER

032A: A9 01              LDA   #1
032C: 8D 39 03           STA   AIPARMS      ;Stuff correct parm count
032F: 20 00 BF           JSR   MLI          ;(Remove interrupt handler)
0332: 41                 DFB   $41          ;DEALLOC_INTERRUPT
0333: 39 03              DA    AIPARMS
0335: B0 06              BCS   ERROR

0337: 58                 CLI
0338: 60                 RTS

0339: 00        AIPARMS  DS    1           ;# of parms
033A: 00                 DS    1           ;Interrupt code # put here
033B: 3E 03              DA    IRQHNDL     ;Address of handler

033D: 00        ERROR    BRK               ;(inelegant error handler!)

                ******************************
                * Here's the interrupt handler *
                ******************************
033E: D8        IRQHNDL  CLD
033F: A2 01              LDX   #SERVEM
0341: 20 5B 03           JSR   MOUSER      ;Check for mouse interrupt
0344: B0 14              BCS   IRQEXIT     ;Branch if it isn't

0346: A2 02              LDX   #READM      ;Clear IRQ condition
0348: 20 5B 03           JSR   MOUSER

034B: AD 82 C0           LDA   $C082       ;Enable monitor ROMs
034E: A9 CD              LDA   #$CD
0350: 20 ED FD           JSR   COUT        ;Display "M"
0353: AD 8B C0           LDA   $C08B
0356: AD 8B C0           LDA   $C08B       ;R/W-enable bank1 of BSR

0359: 18                 CLC
035A: 60        IRQEXIT  RTS

                ****************************************
                * MOUSER executes the mouse subroutine *
                * specified by the code in the X       *
                * register.                            *
```

Table 6-2 Continued

```
                ****************************************
035B: 48        MOUSER   PHA
035C: BD 12 C4           LDA   MTABLE,X    ;Get subroutine addr and
035F: 8D 7C 03           STA   MOUSE       ; set up an indirect JMP
0362: 68                 PLA
0363: 8E 73 03           STX   XSAVE
0366: 8C 74 03           STY   YSAVE
0369: 20 75 03           JSR   DOMOUSE     ;Execute subroutine
036C: AC 74 03           LDY   YSAVE
036F: AE 73 03           LDX   XSAVE
0372: 60                 RTS

0373: 00        XSAVE    DS    1
0374: 00        YSAVE    DS    1

0375: A2 C4     DOMOUSE  LDX   #$C0+SLOT   ;(Mouse in slot 4)
0377: A0 40              LDY   #SLOT*16
0379: 6C 7C 03           JMP   (MOUSE)

037C: 00        MOUSE    DS    1           ;Subroutine address (low)
037D: C4                 DFB   $C0+SLOT    ;(High part is always $Cn)
```

movement). With the Apple Mouse, this determination is made by calling the SERVEMOUSE subroutine. If the carry flag is set, 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, the interrupt is immediately serviced by displaying the letter M on the screen by calling COUT ($FDED), the standard character output subroutine. Before the subroutine ends, the carry flag is cleared so that ProDOS 8 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): Read-enable the ROM area from $D000 to $FFFF. Do this by reading $C082, the soft switch that disables bank-switched RAM. This step is necessary because bank-switched RAM (which is where ProDOS 8 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 8 bank-switched RAM (which includes bank1 of the $Dx bank) for reading and writing by reading from $C08B twice in succession.

You can remove interrupt subroutines from ProDOS 8 with the DEALLOC_INTERRUPT command. But before doing this, 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 command. (The interrupt code number is already in the parameter table from the previous ALLOC_INTERRUPT call.)

Interrupts During MLI Commands

The ProDOS 8 interrupt scheme just described works perfectly well in most situations. Adjustments must be made, however, if an interrupt handler has to call a ProDOS 8 MLI command. (Because of bugs in earlier versions of ProDOS 8, these adjustments work reliably only when using the most recent versions of ProDOS 8.)

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. Typically, 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 permitted to be executed at this time, this data area might be overwritten, causing unpredictable behavior when the first MLI command regained control. You 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.

This means if bit 7 of MLIACTV is 0, the interrupt can be processed normally.

If bit 7 is 1, however, 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 to achieve this result:

- Clear the hardware interrupt condition.
- Take the address stored at CMDADR ($BF9C–$BF9D), and put it in a safe 2-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.)
- 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 to the portion of the interrupt handler that makes the MLI call (that is, the new address stored in CMDADR). Once the MLI call has been made, the interrupt handler passes control to the address originally stored in CMDADR, thus completing the interrupt cycle.

For this procedure to work properly, the reentrant 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 it must end with a JMP to the old CMDADR. Here is what such a subroutine looks like:

```
PHP
PHA
TYA
PHA
TXA
PHA
.
[make the MLI call]
.
PLA
TAX
PLA
TAY
PLA
PLP
JMP (OLDADR)
```

OLDADR is simply the address at which the original address in CMDADR is stored.

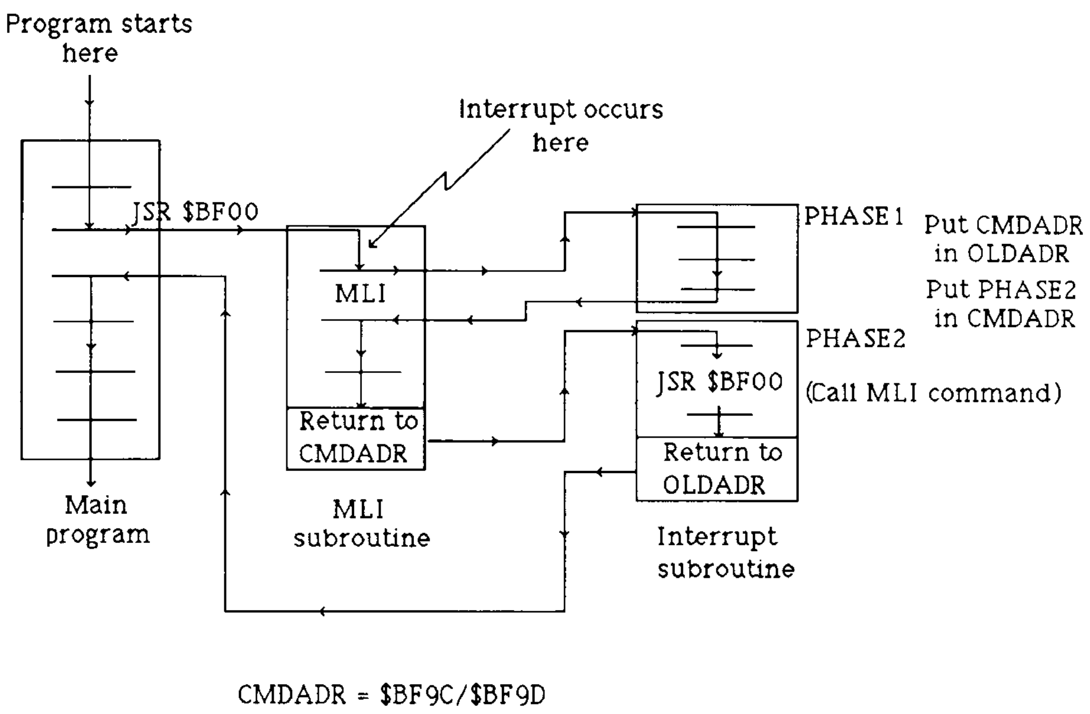
This procedure may seem a little confusing at first. Figure 6-2 should help clarify the flow of control.

The BUTTON.TIME program in Table 6-3 should also help clarify how to deal with the MLI problem. This program enables button interrupts on a mouse and handles such interrupts by reading the current time (using the GET_TIME command) and displaying it on the screen. Once BUTTON.TIME has been installed, the current time will always be at your fingertips. The program assumes a mouse card in slot 4; if that is not the case, change the SLOT EQU 4 directive to reflect the actual slot number.

As usual, the first thing the interrupt handler does is verify that the source of the interrupt is as expected. If it is, the state of bit 7 of MLIACTV is tested using a BIT instruction. If no MLI command is active, bit 7 will be 0, and the interrupt can be serviced right away by calling the GET_TIME command and then displaying the date.

If an MLI command is active, 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 CMDADR before clearing the carry flag and exiting. This means when the current MLI command ends, PHASE2 will take over, and the GET_TIME command will be executed. The time data 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.

Figure 6-2 Handling interrupts during ProDOS 8 MLI commands



Note: CMDADR initially contains the address in the main program to which control is to pass after the JSR $BF00 instruction is executed.

INTERRUPTS AND GS/OS

Generally, the Apple IIGS handles interrupts at the low-level firmware level when GS/OS is active. The firmware maintains an interrupt vector table, each element of which is a JML instruction to the handler for a particular built-in interrupt source, and passes control through the appropriate vector when an interrupt occurs. (See Chapter 8 of *Apple IIGS Firmware Reference* for a detailed description of how the firmware processes interrupts.)

Table 6-3 BUTTON.TIME, a program to illustrate how to handle interrupts during MLI calls

```
                *************************************
                *            BUTTON.TIME            *
                * This program displays the time    *
                * when you click the mouse button.  *
                *                                   *
                * Copyright 1985-1988 Gary B. Little *
                *                                   *
                * Last modified: August 26, 1988    *
                *                                   *
                *************************************
                SLOT     EQU   4          ;Mouse slot number

                MLI      EQU   $BF00      ;Entry point to ProDOS MLI

                MINUTES  EQU   $BF92      ;ProDOS minutes
                HOURS    EQU   $BF93      ;ProDOS hours

                MLIACTV  EQU   $BF9B      ;>=$80 if MLI busy
                CMDADR   EQU   $BF9C      ;Return addr for MLI caller

                HEXDEC   EQU   $ED24      ;Print X/A as decimal number

                MTABLE   EQU   SLOT*$100+$C000+$12 ;Start of ROM table

                * Mouse subroutine numbers:
                SETM     EQU   0          ;Set mouse mode
                SERVEM   EQU   1          ;Service mouse interrupt
                READM    EQU   2          ;Read mouse
                INITM    EQU   7          ;Initialize the mouse

                COUT     EQU   $FDED      ;Standard output

                         ORG   $300

0300: A9 00              LDA   #0         ;Fix ProDOS 1.4 bug by
0302: 8D 9B BF           STA   MLIACTV    ; clearing busy flag

                * Install the interrupt handler:

0305: 78                 SEI              ;Disable interrupts for this

0306: 20 00 BF           JSR   MLI
0309: 40                 DFB   $40        ;ALLOC_INTERRUPT
030A: 1C 03              DA    AIPARMS
030C: B0 12              BCS   ERROR

                * Prepare the mouse:

```

Table 6-3 Continued

```
030E: A2 07              LDX   #INITM
0310: 20 A0 03           JSR   MOUSER     ;Initialize the mouse

0313: A2 00              LDX   #SETM
0315: A9 05              LDA   #$05       ;(Button interrupt mode)
0317: 20 A0 03           JSR   MOUSER     ;Set the mouse mode

031A: 58                 CLI              ;Enable 6502 interrupts
031B: 60                 RTS

031C: 02        AIPARMS  DFB   2          ;# of parms
031D: 00                 DS    1          ;Interrupt code # put here
031E: 21 03              DA   IRQHNDL     ;Address of handler

0320: 00        ERROR    BRK              ;(Inelegant error handler!)

                ********************************
                * Here's the interrupt handler *
                ********************************
0321: D8        IRQHNDL  CLD
0322: A2 01              LDX   #SERVEM
0324: 20 A0 03           JSR   MOUSER     ;Check for mouse interrupt
0327: 90 01              BCC   :1         ;Branch if it is
0329: 60                 RTS

032A: A2 02     :1       LDX   #READM     ;Clear interrupt condition
032C: 20 A0 03           JSR   MOUSER

032F: 2C 9B BF           BIT   MLIACTV    ;In middle of MLI call?
0332: 30 52              BMI   MLIWAIT    ;Yes, so branch

0334: AD 82 C0           LDA   $C082      ;Enable monitor ROMs
0337: 20 54 03           JSR   SHOWTIME
033A: AD 8B C0           LDA   $C08B      ;R/W-enable bank1 of BSR
033D: AD 8B C0           LDA   $C08B      ; (it's active for IRQ)

0340: 18                 CLC              ;(IRQ was serviced)
0341: 60                 RTS

                ********************************************
                * This is the 'deferred' interrupt handler *
                ********************************************
0342: 08        PHASE2   PHP              ;Save all registers first
0343: 48                 PHA
0344: 8A                 TXA
0345: 48                 PHA
0346: 98                 TYA
0347: 48                 PHA

```

Table 6-3 Continued

```
0348: 20 54 03           JSR  SHOWTIME

034B: 68                 PLA               ;Restore all registers
034C: A8                 TAY
034D: 68                 PLA
034E: AA                 TAX
034F: 68                 PLA
0350: 28                 PLP

0351: 6C 9E 03           JMP  (OLDADR)

                ******************************
                * Read the time and print it *
                * as HH:MM.                  *
                ******************************
0354: 20 00 BF  SHOWTIME JSR  MLI
0357: 82                 DFB  $82          ;GET_TIME
0358: 00 00              DA   $0000

035A: AE 93 BF           LDX  HOURS
035D: E0 0A              CPX  #10          ;10 or greater?
035F: B0 05              BCS  ST1          ;Yes, so branch
0361: A9 B0              LDA  #$B0
0363: 20 ED FD           JSR  COUT         ;Print leading zero
0366: A9 00     ST1      LDA  #0
0368: 20 24 ED           JSR  HEXDEC       ;Print HOURS
036B: A9 BA              LDA  #$BA
036D: 20 ED FD           JSR  COUT         ;Print a colon

0370: AE 92 BF           LDX  MINUTES
0373: E0 0A              CPX  #10          ;10 or greater?
0375: B0 05              BCS  ST2          ;Yes, so branch
0377: A9 B0              LDA  #$B0
0379: 20 ED FD           JSR  COUT         ;Print leading zero
037C: A9 00     ST2      LDA  #0
037E: 20 24 ED           JSR  HEXDEC       ;Print MINUTES
0381: A9 8D              LDA  #$8D
0383: 4C ED FD           JMP  COUT

                ***********************************************
                * We now handle the case where an interrupt   *
                * occurs during an MLI call. The address      *
                * stored at CMDADR is saved and replaced by   *
                * the address of PHASE2.                      *
                ***********************************************
0386: AD 9C BF  MLIWAIT  LDA  CMDADR
0389: 8D 9E 03           STA  OLDADR
038C: AD 9D BF           LDA  CMDADR+1
038F: 8D 9F 03           STA  OLDADR+1
```

Table 6-3 Continued

```
0392: A9 42          LDA   #<PHASE2
0394: 8D 9C BF       STA   CMDADR
0397: A9 03          LDA   #>PHASE2
0399: 8D 9D BF       STA   CMDADR+1
039C: 18             CLC               ;("Interrupt handled")
039D: 60             RTS

039E: 00 00   OLDADR DS    2

              ***************************************
              * MOUSER executes the mouse subroutine *
              * specified by the code in the X       *
              * register.                            *
              ***************************************
03A0: 48      MOUSER PHA
03A1: BD 12 C4       LDA   MTABLE,X    ;Get subroutine addr and
03A4: 8D C1 03       STA   MOUSE       ; set up an indirect JMP
03A7: 68             PLA
03A8: 8E B8 03       STX   XSAVE
03AB: 8C B9 03       STY   YSAVE
03AE: 20 BA 03       JSR   DOMOUSE     ;Execute subroutine
03B1: AC B9 03       LDY   YSAVE
03B4: AE B8 03       LDX   XSAVE
03B7: 60             RTS

03B8: 00      XSAVE  DS    1
03B9: 00      YSAVE  DS    1

03BA: A2 C4   DOMOUSE LDX  #$C0+SLOT
03BC: A0 40          LDY   #SLOT*16
03BE: 6C C1 03       JMP   (MOUSE)

03C1: 00      MOUSE  DS    1           ;Subroutine address (low)
03C2: C4             DFB   $C0+SLOT    ;(High part is always $Cn)
```

You use the GS/OS BindInt command to assign an interrupt-handling subroutine to a particular interrupt source. One parameter which BindInt requires is a vector reference number (vrn), a code describing the source of the interrupt to which the handler is to be assigned:

$0008	AppleTalk (SCC)
$0009	Serial ports (SCC)
$000A	Scan-line retrace
$000B	Ensoniq waveform completion
$000C	Vertical blanking signal (VBL)
$000D	Mouse (movement or button)
$000E	1/4-second timer

$000F	Keyboard
$0010	ADB response byte ready
$0011	ADB service request (SRQ)
$0012	Desk Accessory request keystroke
$0013	Flush keyboard buffer request keystroke
$0014	Keyboard micro abort
$0015	1-second timer
$0016	Video graphics controller (external)
$0017	Other interrupt source

Standard system interrupt handlers for many of these interrupt sources are already in place when an application starts up. The vrn for interrupts emanating from devices on peripheral cards is $0017.

When you use BindInt to install a new interrupt handler, remember that the new handler does not replace the old handler. GS/OS chains together all handlers associated with the same vrn, and each handler is called in turn (in reverse order of installation). If one of the handlers claims the interrupt, GS/OS sets the carry flag before calling the next handler in the chain; otherwise, it clears the carry flag.

It may be possible to process certain types of interrupts without installing an interrupt handler. For example, the built-in handler for the vertical blanking interrupt source (vrn = $000C) performs any "heartbeat" tasks which an application, or the operating system, may have placed in a queue with the SetHeartBeat function in the Miscellaneous Tool Set. (See *Apple IIGS Toolbox Reference, Volume 1* for a description of the SetHeartBeat function.)

GS/OS Interrupt Handling

When an interrupt occurs on the Apple IIGS, the firmware interrupt dispatcher identifies the source and then calls each handler for that interrupt source until one of them claims it. Unclaimed interrupts are usually ignored, but if 65,536 of them occur consecutively, a fatal system error occurs.

An interrupt handler is called at fast speed in 65816 native mode with 8-bit A and X registers (m = 1 and x = 1) and with the direct page and data bank registers zeroed. (*Exception*: For Serial Communications Controller interrupts with vrn = $0008 or vrn = $0009, the direct page and data bank registers take on no particular value.) Moreover, the interrupt disable flag in the processor status register is set to 1. The state of the carry flag indicates whether another handler for the same vrn has already dealt with the interrupt (carry set) or not (carry set). All other flags in the processor status register are undefined.

The characteristics of a GS/OS interrupt-handling subroutine are as follows:

- It must *not* enable interrupts with a CLI instruction.
- It must be capable of determining if the source of the interrupt is the one it is designed for. (If the device corresponding to the vrn can generate only one type of interrupt, the handler can assume that its type of interrupt has occurred, of course.)

- If the source of the interrupt is not the one the handler is designed for, the handler must set the carry flag with SEC and exit.
- If the source of the interrupt is the one the handler is designed for, the handler must claim the interrupt by performing the necessary I/O operation, clear the interrupt condition (usually by reading the device status), clear the carry flag with CLC, and exit. (See Table 6-4 for instructions on how to clear certain common Apple IIGS interrupt conditions.) Note, however, that if GS/OS has set the carry flag prior to calling the handler (because a handler with the same vrn has dealt with the interrupt), the handler should not clear the interrupt condition; that will have been done by the first handler to deal with the interrupt and must not be repeated.
- It must exit with an RTL (not an RTI) instruction.

The interrupt handler need not preserve the status of the A, X, and Y registers since GS/OS takes care of that. However, the handler must end at fast speed, in 65816 native mode with 8-bit A and X registers, and with the data bank and direct page registers zeroed. (These are the entry conditions.)

You must install an interrupt handler with the BindInt command (see Chapter 4). Once you've installed an interrupt handler, you can enable the source of the interrupts. For built-in devices, you can do this by passing the appropriate interrupt source reference number to the Apple IIGS Miscellaneous Tool Set's IntSource function, as follows:

```
PushWord #SrcRefNum ;Push interrupt source reference #
LDX  #$2303         ;IntSource
JSL  $E10000
```

Table 6-5 lists the interrupt source reference numbers for the interrupts you can enable with IntSource. Notice that these numbers come in pairs: One is for enabling the source, and the other is for disabling the source.

Use UnbindInt to remove an interrupt handler, but only after you have told the external device to stop generating interrupts (using IntSource if you're dealing with built-in IIGS devices). The int_num parameter you pass to UnbindInt is the number returned by BindInt when you installed the handler.

Handling Interrupts When the System Is Busy

GS/OS and tool set command handlers are generally not reentrant, so a standard interrupt handler should never try to directly call a GS/OS command or a tool set function. If an interrupt happens to occur in the middle of the execution of a GS/OS command, for example, and the handler tries to call a GS/OS command, GS/OS returns error code $07 ("GS/OS is busy"), and the operation fails.

An interrupt handler that needs to use a GS/OS command or a tool set function to process an interrupt request must defer execution until the system is not busy. It can do this by installing a *signal handler* in the GS/OS signal queue; GS/OS processes the

Table 6-4 Clearing Apple IIGS interrupt conditions

Interrupt Condition	*How to Clear the Condition*
1/4-second interrupt	Write anything to $E0C047
1-second interrupt	Clear bit 6 of $E0C032
Scan-line interrupt	Clear bit 5 of $E0C032

Table 6-5 Interrupt source reference numbers for IntSource

Reference Number	*Meaning*
$0000	Enable keyboard interrupts
$0001	Disable keyboard interrupts
$0002	Enable vertical blanking interrupts
$0003	Disable vertical blanking interrupts
$0004	Enable 1/4-second interrupts
$0005	Disable 1/4-second interrupts
$0006	Enable 1-second interrupts
$0007	Disable 1-second interrupts
$000A	Enable Apple Desktop Bus data interrupts
$000B	Disable Apple Desktop Bus data interrupts
$000C	Enable scan-line interrupts
$000D	Disable scan-line interrupts
$000E	Enable external VGC interrupts
$000F	Disable external VGC interrupts

elements in this queue when system resources are guaranteed not to be busy, meaning the signal handler can use GS/OS commands and tool set functions as it pleases.

It is still the duty of the interrupt handler to verify the source of the interrupt, clear the source of the interrupt, and return with the carry flag clear. Moreover, the interrupt handler must install the signal handler by passing its address and priority number to the GS/OS signal installer, as follows:

```
LDA #0000          ;Signal priority number
LDX #DoSignal      ;Handler address (low)
LDY #^DoSignal     ;Handler address (high)
JSL $01FC88        ;Call signal installer routine
```

These instructions must be performed in full native mode. Since the interrupt handler is in 8-bit native mode when it gains control, and must exit in 8-bit native mode, you must precede the above instructions with a REP $30 instruction (and LONGA ON and LONGI ON directives) and follow them with a SEP $30 instruction (and LONGA OFF and LONGI OFF directives).

GS/OS calls the signal handler (with a JSL instruction) in full native mode with interrupts disabled. The A, X, Y, and data bank registers are undefined, and the direct page register takes on the value currently set by the application. The signal handler must end with an RTL instruction.

The program fragment in Table 6-6 shows how to install (and remove) an interrupt handler for the Apple IIGS 1-second interrupt source. To install it, call On_1Sec; to remove it, call Off_1Sec. Both these subroutines use the IntSource function to enable and disable, respectively, the source of the interrupt.

The main interrupt handler begins at OneSec. The first main chore it performs is to clear the source of the interrupt by clearing bit 6 of location $E0C032 (see Table 6-4). Then, since we're assuming the handler must call a GS/OS command, it installs the signal handler at DoSignal before clearing the carry flag and ending with RTL. GS/OS later calls DoSignal when the system is not busy, thereby completing the handling of the interrupt.

Table 6-6 GS/OS subroutines for dealing with 1-second interrupts

```
On_1Sec  _BindInt BI_Parms     ;Install handler
         LDA  int_num
         STA  int_num1
         PushWord #6
         _IntSource            ;Enable 1-second interrupts
         RTL

Off_1Sec  PushWord #7
          _IntSource           ;Disable 1-second interrupts
          _UnbindInt UI_Parms  ;Remove handler
          RTL

BI_Parms  DC   I2'3'
int_num   DS   2
          DC   I2'$0015'       ;vrn for 1-second interrupt
          DC   I4'OneSec'

UI_Parms  DC   I2'1'
int_num1  DS   2

; This is the interrupt handler:

OneSec    ANOP
          LONGA OFF      ;8-bit registers on entry
          LONGI OFF

          PHB            ;Must preserve data bank
          PHK
          PLB
          LDA   $E0C032
          AND   #$BF     ;Bit 6 = 0
          STA   $E0C032  ;Clear 1-second interrupt source

          REP   $30      ;Go to full native mode
          LONGA ON
          LONGI ON

          LDA   #0000    ;Priority number (anything will do)
          LDX   #DoSignal
          LDY   #^DoSignal
          JSL   $01FC88  ;Install signal handler

          SEP   $30      ;Back to 8-bit mode
          LONGA OFF
          LONGI OFF

          PLB            ;Restore data bank
          CLC            ;CLC = we handled it
          RTL            ;Always end with RTL
```

Table 6-6 Continued

```
; This is the signal handler:

DoSignal  ANOP
          LONGA ON       ;16-bit registers on entry
          LONGI ON

; Call the GS/OS command here

          RTL            ;Always end with RTL
```

CHAPTER 7

Disk Devices

Low-level communication with a block-structured data storage device like a 3.5-inch disk drive or a hard disk is managed by an assembly-language subroutine called a *disk driver*. (This name is conventional only—a disk driver may actually communicate with a block storage device that is not a disk drive.) We say "low level" because the disk driver is the subroutine every operating system command eventually calls to access the disk, and it is the disk driver that directly manipulates the I/O locations that control the operation of the drive.

The important tasks conventional disk drivers perform are

- Moving the disk's read/write head over any track on the disk
- Identifying data blocks within each track
- Reading and writing data blocks
- Reading the write-protect (or other) status of the disk
- Formatting the disk

The driver for a 5.25-inch drive performs these tasks using several disk I/O locations for controlling the disk stepper motor, storing a byte on a disk, reading a byte from a disk, and sensing the write-protect status of the disk.

Under ProDOS 8, it is relatively easy to add a custom disk driver (such as one for controlling a RAMdisk) to the system—it's just a matter of changing a few bytes in the system global page to tell ProDOS 8 where you've loaded the driver and what slot and drive numbers you want to assign to it. The only difficult part is deciding where to put the driver so that it won't be overwritten by applications.

GS/OS has a more formal mechanism for adding disk drivers, but we do not discuss them here; GS/OS comes with drivers for all the disk drivers you're ever likely to need. If you do need to know about how to write GS/OS disk drivers, refer to *GS/OS Reference, Volume 2.*

In this chapter, we investigate just how GS/OS and ProDOS 8 determine what disk devices are available and how they keep track of the disk drivers associated with each of these devices. We also review the general characteristics of a ProDOS 8 disk driver and learn how to write one from scratch.

HOW GS/OS AND PRODOS 8 KEEP TRACK OF DISK DEVICES

When GS/OS and ProDOS 8 boot up, one of the first things they do is determine how many disk devices are connected to the system and how they may be accessed, for example, through a card in a slot or a RAM-based driver. (GS/OS also checks for character devices.) We see how these operating systems identify disk devices in the next section.

GS/OS Device Scan

When you boot GS/OS, it scans the IIGS system looking for both block-structured disk devices and character devices. When it identifies a device for which a driver exists in the SYSTEM/DRIVERS/ subdirectory on the boot disk, it loads the driver into memory and installs it. Apple currently provides drivers for 3.5- and 5.25-inch disk drives, SCSI drives, and the console (the standard keyboard input and text-screen output system). If no driver for the device exists on disk, GS/OS tries to generate one in memory on the fly; it can generate character drivers for printer and modem interfaces and disk drivers for most SmartPort devices. The disk devices GS/OS cannot generate a driver for are the 5.25-inch disk drive and the HD20SC SCSI hard disk.

GS/OS assigns a unique device number and device name to each device it finds in the system. It assigns device numbers consecutively, beginning with 1, and the device names begin with a period and can be up to 31 characters long (for example, .DEV3 and .APPLEDISK3.5). Unlike ProDOS 8, GS/OS does not use unit numbers (which are derived from slot and drive numbers) to identify disk devices. You can use the GS/OS DInfo command to determine the names of all the devices in the system.

ProDOS 8 Device Scan

Table 7-1 lists all the system global page locations ProDOS 8 uses to manage disk devices.

ProDOS 8 stores the number of active disk devices, less 1, in DEVCNT ($BF31) in the system global page. It stores the physical locations of the disk devices (that is, their slot and drive numbers) in encoded form in a 14-byte table beginning at DEVLST ($BF32). As Figure 7-1 shows, the high-order 4 bits of each entry in this table hold the drive and slot number in packed form, and the low-order 4 bits hold an identification code unique to the type of disk device installed (5.25-inch drive, 3.5-inch drive, HD20SC hard disk, and so on).

You can also use the ProDOS 8 ON_LINE command to determine the slot and drive locations of all the disk drives in the system.

Table 7-1 ProDOS 8 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 minus 1
$BF32	DEVLST	Table of active disk device codes (14 entries in table)

NOTE: The format of the entries in DEVLST and DEVNUM is the same as shown in Figure 7-1, except that the low-order 4 bits of DEVNUM are always 0.

Figure 7-1 The format of DEVLST ($BF32) table entries

Each byte in the 14-byte DEVLST table holds the slot, drive, and disk identification number in a special packed format:

```
 7   6   5   4   3   2   1   0
+---+-----------+---------------+
|DR |   SLOT    |    DISK_ID    |
+---+-----------+---------------+
```

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 5.25-inch disk drive
= $B for a 3.5-inch disk drive
= $F for the /RAM device
= the high-order 4 bits stored at $CnFE if a disk controller adhering to the 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.

Suppose you are using a two-drive Apple IIe with an extended 80-column text card installed in the auxiliary slot and a disk controller card installed in slot 6. ProDOS 8 sets up DEVCNT and DEVLST as follows:

```
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 8 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 and 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 cannot reside in slot 0 (a slot that doesn't even exist on the Apple IIe, IIc, or IIGS), ProDOS 8 uses the two slot 0 entries in the disk driver vector table for a special purpose: to hold the address of the subroutine that generates MLI error $28 if ProDOS 8 calls it. This is the code for the "no device connected" error. If the vector table entry for a given slot and drive combination is this address, ProDOS 8 has not assigned a disk device to that slot and drive.

The six most common entries in the disk driver vector table are as follows:

$D000	disk driver for a standard 5.25-inch disk drive (in bank-switched RAM)
$FF00	disk driver for the /RAM RAMdisk volume (in bank-switched RAM)
$DEAC	address of "no device connected" error subroutine (in bank-switched RAM)
$Cn0A	UniDisk 3.5 and Apple IIGS SmartPort (n = slot number of the controller card)
$Cn4E	Apple II Memory Expansion card (n = slot number of the memory card)

The first three addresses are those used by ProDOS 8 version 1.7 only. (The others are fixed in ROM on firmware or controller cards.) They may change when Apple releases later versions of ProDOS 8.

HOW GS/OS AND PRODOS 8 IDENTIFY DISK DEVICES

To connect a disk device to an Apple II, you generally attach it to a disk controller card located in a peripheral expansion slot. (The IIc and IIGS both have built-in disk controllers, so no card is necessary.) This card is responsible for booting the disk and, in some cases, for transferring data between the Apple and the disk medium.

A controller card holds a program in ROM that occupies the address space from $CnOO to $CnFF (where n is the slot number) and, sometimes, from $C800 to $CFFF. For standard 5.25-inch disk controllers, this program is capable of only transferring a short loader program from the disk medium into RAM and executing it; this loader then reads in the rest of the disk operating system from disk. (This is where the term booting comes from: The operating system picks itself up by its own bootstraps.)

Other controllers may contain code that performs much more sophisticated tasks, such as reading or writing any block on the disk, doing status checks, and formatting a disk. Intelligent controllers with these capabilities are used with 3.5-inch disks and hard disks. Apple currently uses an intelligent controller called a SmartPort for 3.5-inch drives and RAMdisk memory cards. (A SmartPort is built in to the IIGS and newer models of the IIc.)

When ProDOS 8 or GS/OS first starts up, it examines each slot (beginning with 7 and working down to 1) to determine whether a controller card for a disklike device is present. A controller card contains 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 location $Cn07 contains $3C, and if the controller is in a higher-numbered slot than any other disk controller, the original Apple II system Monitor program in ROM (the one in the II Plus or the original IIe) automatically boots the disk in the drive when you turn the system on. Unfortunately, $Cn07 cannot contain $3C in the ROM of a controller for a disk device other than a 5.25-inch disk drive because the Apple Pascal operating system erroneously believes any such device is a 5.25-inch disk drive. As a result, it is not possible to automatically boot from a hard disk or a 3.5-inch disk when using a system with the original Monitor program.

You *can* automatically boot a non-5.25-inch disk device if you have an Apple IIGS or an enhanced Apple IIe. This is because the system Monitor in these computers identifies a bootable disk drive by the presence of the first 3 identification bytes only.

If you want to know if the disk controller is a SmartPort (perhaps so that you can take advantage of the special SmartPort commands described later in this chapter), check location $Cn07. If it contains $00, it is a SmartPort.

When ProDOS 8 or GS/OS finds the 3 identification bytes, it looks at the byte stored at $CnFF to determine the exact type of controller it has found. If $CnFF contains $00, ProDOS 8 and GS/OS consider the card a 5.25-inch disk controller with standard 16-sector-per-track ROMs. In this case, ProDOS 8 places the appropriate device code in the DEVLST table and the address of the internal 5.25-inch disk device driver in the ProDOS 8 disk driver vector table. Note that it actually makes two entries in each table since each 5.25-inch disk controller can have two drives (or volumes) attached to it. (They are referred to as drive 1 and drive 2.) The disk driver itself ultimately determines if there is actually a drive 2 device attached and returns a "device not connected" error code if an attempt is made to access it and it is not there.

If $CnFF contains $FF, GS/OS and ProDOS 8 consider the card a 5.25-inch disk controller with 13-sector-per-track ROMs. (This was the disk formatting scheme used by Apple's original 5.25-inch drive controller.) GS/OS and ProDOS 8 do not support this type of controller card and so ignore it.

If $CnFF contains any other value, GS/OS and ProDOS 8 assume the disk controller has a device driver entry point 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 describe the meaning of these bits in the next section), ProDOS 8 stores this address in the device driver vector table and adds an appropriate device code to DEVLST. (The low-order 4 bits of the DEVLST entry are set equal to the high-order 4 bits of the byte at $CnFE.) If one, or both, of bits 0 and 1 of $CnFE are 0, GS/OS and ProDOS 8 ignore the disk controller.

ProDOS 8 identifies three special "disk" devices in quite a different way. If it is running on an Apple IIe with an extended 80-column card (the one with 64K of auxiliary RAM on it), or on an Apple IIc or IIGS, ProDOS 8 installs a special device, called a RAMdisk, as the slot 3, drive 2 disk device. The medium for this disk is the 64K auxiliary memory space on the IIe, IIc, or IIGS, 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.

GS/OS and ProDOS 8 create another type of RAMdisk using memory on the Apple IIGS Memory Expansion card (or equivalent) if the Control Panel Minimum RAM Disk Size parameter is not set to zero. This RAMdisk is called /RAM5. The third special device, again available on the IIGS only, is a ROMdisk. Although Apple's memory card doesn't support ROMdisk memory, several independent suppliers have cards that do. Despite the name ROMdisk, the memory for the disk could also be in battery backed-up static or dynamic RAM, EEPROM, or EPROM.

EXTENDED PROTOCOL FOR DISK CONTROLLER CARDS

Apple has also defined a special extended controller card ROM protocol that manufacturers of disk devices and disk controller cards must adhere to if their devices are to work properly with GS/OS and ProDOS 8. (The 5.25-inch disk controllers do not actually follow this protocol and are handled as special cases by GS/OS and ProDOS

8.) This protocol defines the use of 4 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 the benefit of formatting programs that also initialize the volume directory and volume bit map on disk. The controller for the old 5-megabyte ProFile hard disk has the number $2600 (9728) stored here. If the number is $0000 (as it is for most controller cards), you must send a status request to the disk driver to determine the volume size; the number of blocks comes back in the X register (low) and Y register (high). We 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 UniDisk 3.5 has the value $BF stored at $CnFE. This means the disk medium is removable (bit 7 = 1); the UniDisk 3.5 is not interruptible (bit 6 = 0); two volumes are supported (bits 5,4 = 11); and the device driver for the UniDisk 3.5, located in ROM on the controller card, supports format (bit 3 = 1), 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 8 disk 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 8 global page when ProDOS 8 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 unless ProDOS 8 is able to determine that a second drive is not actually connected. After the vector table is updated, bits 4–7 of the byte stored at $CnFE are stored in the low-order 4 bits of the DEVLST entry for the device.

  The controller for the UniDisk 3.5 has the value $0A stored at $CnFF, and its DEVLST entry is of the form nB, where n is the controller slot number. This means the address of the disk driver is $Cn0A.

Special Cases

$CnFF contains $00 for a 16-sector 5.25-inch disk controller and $FF for a 13-sector 5.25-inch disk controller. In these situations, GS/OS and ProDOS 8 attribute no special meaning to the values stored at $CnFC, $CnFD, and $CnFE.

If ProDOS 8 finds a 16-sector controller, it assumes the disk medium is a single volume of 280 blocks and uses its own internal disk driver to communicate with it. GS/OS uses a similar driver it loads from the SYSTEM/DRIVERS/ subdirectory. GS/OS and ProDOS 8 ignore the older 13-sector 5.25-inch disk controller.

COMMUNICATING WITH A PRODOS 8 DISK DRIVER

Just before ProDOS 8 calls a disk driver subroutine, it sets up four parameters in the microprocessor's page zero area 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 (one block) data transfer buffer to be used, and the block number.

The four parameters are stored in locations $42 to $47 and have the following meanings:

- COMMAND ($42). This location holds the command code for the disk operation to be performed. Four codes are defined:

0	Check device status. On return, the carry flag is clear and the accumulator is zero if the device is ready to accept read and write commands. Moreover, the number of blocks on the disk is in the X register (low) and Y register (high) but only if the device's controller ROM adheres to the extended ProDOS 8 protocol (remember that 5.25-inch disk controllers do not). If the device is not ready to accept read and write commands, the carry flag is set, and the accumulator contains an MLI error code. The standard drivers for 3.5- and 5.25-inch drives return an error code on a status request if the disk medium is write-protected (error $2B) or no disk is in the drive (error $2F).
1	Read one block from the disk.
2	Write one block to the disk.
3	Format the disk. When you format a disk, special address marks are set up to allow each sector to be identified by the disk driver. Generally, 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 driver for /RAM is an exception.) The format request is actually not supported by the standard

```
5.25-inch device driver because of space
limitations; instead, a separate utility program
(such as Filer on the ProDOS 8 master disk) must
be used to format a diskette or hard disk and to
lay out the boot record, volume directory, and bit
map. The source code for the standard diskette
formatting subroutines (called FORMATTER) can also
be licensed from Apple for use in other formatting
programs. The format request is supported by the
/RAM driver and the 3.5-inch disk driver.
```

- SLOT_DRIVE ($43). These locations hold the drive and slot numbers 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 to be written to (COMMAND = 2) or read from (COMMAND = 1).

The disk driver performs the I/O operation dictated by these parameters and then returns control to the caller. If no error occurred, the carry flag is clear, and the accumulator is zero.

Errors can occur, of course, when ProDOS 8 communicates with a disk device. The disk drivers flag error conditions in the standard MLI way: by setting the carry flag and placing an appropriate MLI error code in the accumulator. Table 7-2 shows the error codes and conditions supported by the ProDOS 8 disk driver for standard 5.25-inch disk drives. Any other properly implemented disk driver will identify and report these error conditions in the same way.

THE SMARTPORT CONTROLLER

A SmartPort is the intelligent device controller Apple now uses to interface to all its high-capacity disk drives, including the UniDisk 3.5, Apple 3.5 Drive, and HD20SC SCSI hard disk. The SmartPort firmware can handle up to 127 devices chained

Table 7-2 ProDOS 8 disk driver error codes

MLI Error Code	*Meaning*
$27	I/O error
$28	No disk device is connected
$2B	The medium is write-protected
$2F	The device is off-line

together to the same SmartPort, but the Apple power supply gives out well before then—for the SmartPort on the IIGS, for example, Apple recommends connecting no more than four 3.5-inch drives.

As we mentioned earlier in this chapter, the SmartPort firmware has the same three basic identification bytes as any other ProDOS-compatible disk controller. A $00 at location $Cn07 serves to uniquely identify the controller as a SmartPort, however. The SmartPort ID type byte at $CnFB gives you a little more information about the SmartPort:

```
bit 0   1 = supports RAMdisk card
bit 1   1 = supports SCSI devices
bit 2   [reserved]
bit 3   [reserved]
bit 4   [reserved]
bit 5   [reserved]
bit 6   [reserved]
bit 7   1 = supports extended commands
```

The SmartPort assigns a unique unit number (from $01 to $7F) to each device connected to it. The numbers it assigns are consecutive, starting with $01. (The SmartPort controller itself is unit number $00.) Programs use the unit number to identify the device a SmartPort command is directed to.

In general, the SmartPort assigns unit numbers to devices in the order they appear in the chain of devices. But on the IIGS, the SmartPort considers any ROMdisk or /RAM5 RAMdisk (the RAMdisk you set up with the Control Panel) in the system to be part of the SmartPort chain and assigns unit numbers to them first. To complicate matters further, if the startup device (set using the Control Panel) is a SmartPort device, the SmartPort rearranges unit numbers to ensure the startup device has a unit number of $01. The only safe way to determine which device corresponds to a given unit number is to use the SmartPort's Status command (see below).

Many ProDOS 8 commands use slot and drive parameters to identify a disk device, so ProDOS 8 automatically assigns slot and drive combinations to SmartPort unit numbers when it first boots up. Assuming the SmartPort is in slot 5, ProDOS 8 assigns the first four SmartPort devices to slot 5, drive 1; slot 5, drive 2; slot 2, drive 1; and slot 2, drive 2. It ignores any other devices that may be connected to the SmartPort. The phantoming of the third and fourth devices to slot 2 is necessary because ProDOS 8 has space for only two drives per slot in its disk driver vector table.

Using SmartPort Commands

The SmartPort firmware provides several commands a program can use to communicate with a disk device. Under ProDOS 8, you won't have to use them for common types of disk operations because you can use the disk driver commands described in the previous section instead. Under GS/OS, you can probably get by with the DInfo, DRead, DWrite, DStatus, and DControl commands. You will have to use SmartPort commands to obtain extended status information and to perform special control operations, however.

To use a SmartPort command, you must first determine the dispatch address of the command interpreter. This address is always 3 bytes past the standard ProDOS 8 device driver entry point, so its offset into page $Cn00 is the value stored at $CnFF plus 3.

You call a *standard* SmartPort command much as you call a ProDOS 8 MLI command:

```
JSR  DISPATCH       ;DISPATCH = $Cn00+($CnFF)+3
DFB  CMDNUM         ;SmartPort command number
DA   PARM_BLK       ;Pointer to SmartPort parameters
BCS  ERROR          ;Carry set if error occurred
```

where DISPATCH is the SmartPort dispatch address, CMDNUM is the SmartPort command number, and PARM_BLK is a command-specific parameter block. (If GS/OS is active on a IIGS, you must call the SmartPort dispatcher in emulation mode with code that resides in bank $00.) If an error occurs, the carry flag is set, and the accumulator contains the error code. If the operation was successful, the carry flag is clear, and the accumulator is zero.

If bit 7 of the SmartPort ID type byte at $CnFB is 1 (and it is for the IIGS SmartPort), the SmartPort also supports *extended* SmartPort commands. The command number for an extended command is the same as the number for the corresponding standard command except that bit 6 is set to 1. That means, for example, if the standard command number is $01, the extended command number is $41.

You call extended commands just like standard commands except that the pointer to the parameter block contains a long address (4 bytes) rather than a short address (2 bytes). This permits access to a parameter block located anywhere in the IIGS's 16Mb memory space. The other difference between a standard and extended command is the structure of the parameter block for the command, as we see below.

Important: The IIGS SmartPort clobbers several locations in the caller's 65816 direct page (IIGS ROM version 01) or true zero page (original IIGS ROM) when you call a SmartPort command. The affected locations are $57 through $5A. If these locations are important to your application, save them before a SmartPort call and restore them afterward.

All SmartPorts support a standard set of commands so that ProDOS 8 or GS/OS can communicate with it properly. The ones you probably will never use in an application are ReadBlock, WriteBlock, Format, and Init (you can use ProDOS 8 disk driver or GS/OS commands instead) as well as Open, Close, Read, and Write (appropriate for character devices only). Let's now take a close look at the two remaining commands, Status and Control.

Status Command

The Status command is for determining the status of any device in the SmartPort chain or the SmartPort controller itself. Its command number is $00 (standard) or $40 (extended), and the standard parameter block looks like this:

parameter count (byte, always $03)

unit number (byte, from $00 to $7E)

status list pointer (low byte)

status list pointer (high byte)

status code (byte, from $00 to $FF)

The extended parameter block uses a 4-byte pointer to the status list instead (low-order bytes first). You must reserve space for the status list before calling the Status command.

There are four possible values for the status code byte:

```
$00    return device status
$01    return device control block
$02    return newline status
$03    return device information block
```

Of these, you probably won't use code $01 or $02 very often. Code $01 returns a device-dependent control block, up to 256 bytes long, preceded by a length byte; a length byte of $00 means the block is 256 bytes long. Code $02 is for character devices only.

Code $00 (return device status) returns 4 or 5 bytes in the status list depending on whether a standard or extended call is made. The first byte is a general device status byte:

bit 0	1 = disk switched (block device only) or
	1 = device is open (character device only)
bit 1	1 = device is interrupting
bit 2	1 = medium is write-protected (block device only)
bit 3	1 = device allows formatting
bit 4	1 = a disk is in the drive
bit 5	1 = device allows reading
bit 6	1 = device allows writing
bit 7	1 = block device
	0 = character device

Note that the *disk-switched* bit is 1 if a disk has been ejected and another disk (perhaps the same one) has been inserted since the last status check. But this bit is significant only if the device supports disk-switched errors; it does if bit 6 of the subtype byte returned by the code $03 status command is 1 (see below). Of Apple's SmartPort devices, only the Apple 3.5 Drive for the IIGS supports these types of errors. (The UniDisk 3.5 does not.)

The next 3 bytes (standard call) or 4 bytes (extended call) hold the size of the device in blocks. These bytes are zero if the device is a character device.

The SmartPort handles a Status call differently if the unit number is $00. In this case, it returns an 8-byte status list describing the status of the SmartPort controller itself:

byte 0	number of devices the SmartPort controls
byte 1	interrupt status (no interrupt if bit 6 is set)
byte 2	manufacturer of driver:
	$00 = unknown
	$01 = Apple
	$02 = third-party driver

Bytes 3 through 7 are reserved.

Code $03 (return device information block) returns more detailed status information in the status list. The form of the list after a standard call is as follows:

device status (byte)

block size (low byte)

block size (medium byte)

block size (high byte)

ID string length (byte)

ID string (16 bytes)

device type (byte)

device subtype (byte)

version (2 bytes)

For an extended call, the block size field occupies 4 bytes instead of 3.

The device status and block size bytes are the same as those returned by a status code $00 call. The ID string is a sequence of up to 16 standard ASCII characters (the high-order bit of each character is 0) representing the name of the device. The 16-character string space is padded with spaces if necessary.

The device type byte tells you the general nature of the device you're dealing with. The currently defined values are as follows:

`$00`	`Memory expansion card RAMdisk`
`$01`	`3.5-inch disk drive`
`$02`	`ProFile-type hard disk`
`$03`	`Generic SCSI hard disk`
`$04`	`ROMdisk`
`$05`	`SCSI CD-ROM`
`$06`	`SCSI tape or other SCSI sequential device`
`$07`	`SCSI hard disk`
`$08`	`[reserved]`
`$09`	`SCSI printer`
`$0A`	`5.25-inch disk drive`
`$0B`	`[reserved]`
`$0C`	`[reserved]`
`$0D`	`Printer`
`$0E`	`Clock`
`$0F`	`Modem`

The subtype byte indicates some of the characteristics of the device:

`bit 7`	`1 = supports extended SmartPort commands`
`bit 6`	`1 = supports disk-switched errors`
`bit 5`	`1 = nonremovable medium`

The other 5 bits are reserved.

Version is a word (low-order byte first) describing the version number of the SmartPort device driver.

Control Command

The Control command sends control information to a device. Its command number is $04 (standard) or $44 (extended), and the standard parameter block looks like this:

parameter count (byte, always $03)

unit number (byte, from $00 to $7E)

control list pointer (low byte)

control list pointer (high byte)

control code (byte, from $00 to $FF)

The extended parameter block uses a 4-byte pointer to the control list instead (low-order bytes first).

The Control command understands five general control codes, only one of which (eject medium) is particularly useful to most applications: $00 (device reset), $01 (set device control block), $02 (set newline status), $03 (service device interrupt), and $04 (eject medium). Device-specific control codes are numbered $05 and above.

The most useful control code for most applications is the one that causes a 3.5-inch disk to eject automatically. For the UniDisk 3.5 SmartPort card and the internal IIGS SmartPort, the eject control code is $04, and the control list contains two $00 bytes. (For a summary of other device-specific control codes, see Chapter 7 of *Apple IIGS Firmware Reference*.)

Remember to use the eject command with 3.5-inch drives only. You can easily check whether you're dealing with a 3.5-inch drive by using the Status command. If you are, the device type byte is $01, the block size is $000640, and the ID string is DISK 3.5.

If you need to be convinced to do a Status check first, keep in mind that revisions A and B of the SCSI card (a SmartPort device) for Apple's HD20SC hard disk use a control code of $04 to format the disk! That's not an operation you want to perform accidentally. (For revision C of the SCSI card, the $04 code *is* the eject code.)

THE PRODOS 8 RAMDISK: THE /RAM VOLUME

We saw earlier that ProDOS 8 automatically installs a special RAMdisk driver if you are using an Apple IIGS, Apple IIc, or Apple IIe with an extended 80-column text card and creates a special volume called /RAM. (Apple II Plus users are out of luck.) All these systems have 64K of *auxiliary* memory that maps to addresses in exactly the same way as the standard 64K of main RAM memory usually used for program and data storage. In this auxiliary memory, the RAMdisk driver stores the volume directory, volume bit map, and file blocks. Figure 7-2 shows a map of the usage of auxiliary memory by /RAM.

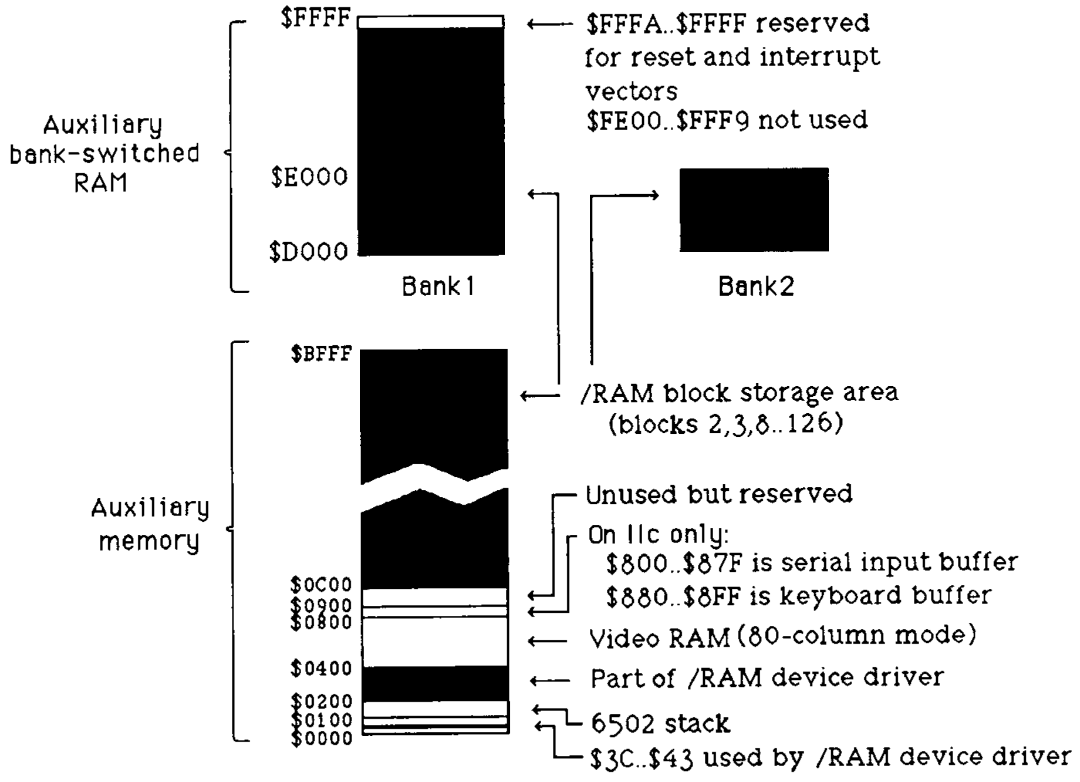
Since no slow-moving mechanical parts are used to perform "disk" operations (all I/O operations simply involve block moves from one part of memory to another), the RAMdisk responds much more quickly than a conventional disk drive. But its contents are temporary, so you must be careful to transfer any files from it to a permanent disk medium before turning off the Apple or rebooting ProDOS 8, or you will lose all of your data.

Characteristics of the /RAM Volume

When ProDOS 8 initializes the /RAM volume, it allocates only one volume directory block (block 2; recall that standard disks use four directory blocks). This means there is room for only 12 entries in the volume directory, not the usual 51. If files are created inside subdirectories, however, you can store as many files as will fit on the volume.

When ProDOS 8 first initializes the /RAM volume, 119 blocks are available for file storage. (They are numbered from 8 to 126.) Since a 64K space is normally capable of holding 128 512-byte blocks, you might be wondering about the "missing" 9 blocks.

Figure 7-2 A map of auxiliary memory usage on the Apple IIe, IIc, and IIGS with ProDOS 8 active



Auxiliary bank-switched RAM
$FFFF
$E000
$D000
Bank1
$FFFA..$FFFF reserved for reset and interrupt vectors
$FE00..$FFF9 not used
Bank2
Auxiliary memory
$BFFF
/RAM block storage area (blocks 2,3,8..126)
Unused but reserved
On IIc only:
$800..$87F is serial input buffer
$880..$8FF is keyboard buffer
$0C00
$0900
$0800
Video RAM (80-column mode)
$0400
Part of /RAM device driver
$0200
$0100
$0000
6502 stack
$3C..$43 used by /RAM device driver

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 IIc ($0800–$08FF), and the auxiliary memory interrupt vectors ($FFFA–$FFFF). Thus these seven blocks are marked as "in use" in the /RAM volume bit map.

The areas of auxiliary memory that the /RAM volume or its driver does not use 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 8 and must not be used by nonsystem software.

The first 8K of memory allocated for use by files stored in /RAM maps to locations $2000–$3FFF in auxiliary memory. This same space is used whenever you activate page 1 of the double-width high-resolution graphics display mode available on the IIGS, IIc, or IIe. If you are going to use this graphics mode while /RAM is active, 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 exactly 8K bytes long. You can do this by entering the following command from Applesoft command 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 used as the second page of double-width high-resolution graphics. You can protect this page by saving another dummy file that is 8K long.

Removing and Reinstalling /RAM

You may want your application to use the auxiliary memory area for purposes other than as a convenient 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. But before you start overwriting the RAM volume with such data, you must remove the /RAM volume from the system in an orderly manner. If you don't, the system could crash when ProDOS 8 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.

1. 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 exist in only a 128K system.
2. 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, though not equivalent to /RAM, will also use the first bank of auxiliary memory. (Cards such as RamWorks III and MultiRAM have several banks of auxiliary memory available.) The actual $BX byte stored in DEVLST must be saved if you later want to reinstall the /RAM device.

3. Remove the $BX entry from the DEVLST table by moving higher-addressed active entries down one position (starting with the lowest-addressed one).
4. 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 generates a "no device connected" error condition.) The original slot 3, drive 2 entry must be saved if you later want to reinstall the /RAM device.
5. Decrement DEVCNT ($BF31).
6. Make an ON_LINE call with unit_num set to $B0. This frees up an internal buffer so that you can have more disk volumes active at once.

After you perform these steps, the /RAM device disappears from ProDOS 8, and auxiliary memory can be safely used for other purposes.

When your application ends, it should reinstall /RAM. Do this by performing the following steps:

1. As a precaution, verify that you have not already reinstalled /RAM by checking for a slot 3, drive 2 device code in DEVLST.
2. Restore the original slot 3, drive 2 device vector that you saved before /RAM was disconnected.
3. 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 $BF32).
4. Increment DEVCNT ($BF31).
5. 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 must enable that bank by reading $C08B twice in succession before making the call. When the call ends, reenable the Applesoft and motherboard ROMs by reading $C082. Here is a subroutine that performs 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       ;Read/write enable bank1
```

```
        LDA $C08B       ; (where the driver is)
        JSR TORAM
        LDA $C082       ;Reenable Applesoft ROMs
        RTS
TORAM   JMP ($BF26)     ;Call the /RAM driver
```

After you reinstall /RAM like this, it is once again available for use as a file-storage device.

WRITING A PRODOS 8 DISK DRIVER

The best way to learn about disk drivers and how ProDOS 8 installs them is to actually write one. In this section, we do just that by creating a driver for an 8K version of /RAM called /RAM8. It is suitable for use in an Applesoft programming environment and can be used by all ProDOS 8 users (unlike /RAM, which is not available to Apple II Plus users). The RAMdisk driver itself resides in page three, and the "disk" storage space it uses is located from $0800 to $27FF. We ensure that Applesoft programs do 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 create the disk driver, let's outline the steps to follow to remedy the Applesoft conflict, bind the driver into ProDOS 8, and then initialize the RAM-disk. This is really a five-step process.

The first step in the procedure is to adjust the Applesoft pointers so that when you enter or load BASIC programs, 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
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.

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 see what it looks like in a moment.)

Finally, we must initialize the volume directory block and the volume bit map. But before we can do this, we must know three things:

- 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 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 to conform to ProDOS conventions.

The volume bit map block will be stored at block 3, leaving a total of 14 blocks (7K) for file storage. To keep the file storage area contiguous, we 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 a placeholder for a sparse file.) This means ProDOS will think 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 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 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 examines 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 is as follows:

```
block 2  --> $800-$9FF
block 3  --> $A00-$BFF
block 4  --> $C00-$DFF
.
.
.

block 17 --> $2600-$27FF
```

(The driver returns an error code if a block number greater than 17 is requested.) With this scheme in place, 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 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 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 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 8)
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 8 checks to see if a valid driver is installed. The first part of the driver saves the contents of two zero page locations we're going to overwrite and then checks whether the requested block number (stored at $46–$47) is within the allowable range. If it isn't, 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 driver can access the block of data 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 performs just the opposite transfer. 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 simply the movement of a 512-byte block of data from one area of memory to another.

Table 7-3 shows the complete source listing for a slightly embellished form of this driver. 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 8 global page to prevent the /RAM8 volume from being overwritten. Any attempt to load a file into these areas (using BLOAD or BRUN) results 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.

Table 7-3 The /RAM8 disk driver program

```
               *************************************
               *                                   *
               *     ProDOS RAMdisk disk driver    *
               *                                   *
               * This driver controls a 8K RAMdisk *
               * volume called /RAM8.              *
               *                                   *
               * Copyright 1985-1988 Gary B. Little *
               *                                   *
               * Last modified: August 26, 1988    *
               *                                   *
               *************************************
               RAMPTR   EQU  $6         ;Pointer to RAMdisk block

               COMMAND  EQU  $42        ;Command code
               BUFFER   EQU  $44        ;Buffer address
               BLOCK    EQU  $46        ;Block number

               TXTTAB   EQU  $67        ;Applesoft program pointer

               INITBLK  EQU  $3000      ;Block buffer

               MLI      EQU  $BF00      ;MLI interface
               DEVADR01 EQU  $BF10      ;Start of disk driver table
               DEVCNT   EQU  $BF31      ;# of disk devices (minus 1)
               DEVLST   EQU  $BF32      ;Table of slot, drive for disks
               BITMAP   EQU  $BF58      ;Start of system bit map

                        ORG  $2000

               * Move device driver code into place:

2000: A0 00             LDY  #0
2002: B9 F4 20 MOVECODE LDA  BEGIN,Y
2005: 99 00 03          STA  RAMDISK,Y
2008: C8                INY
2009: C0 7C             CPY  #END-RAMDISK
200B: D0 F5             BNE  MOVECODE

               **********************************
               * Mark pages 3, 8..27 as "in use" *
               * in the system bit map. This    *
               * prevents /RAM8 or its driver   *
               * from being overwritten by BLOAD.*
               **********************************
200D: AD 58 BF          LDA  BITMAP
2010: 09 10             ORA  #$10       ;Block 3 bit = 1
2012: 8D 58 BF          STA  BITMAP
2015: A9 FF             LDA  #$FF
```

Table 7-3 Continued

```
2017: 8D 59 BF            STA  BITMAP+1     ;Blocks 8..15
201A: 8D 5A BF            STA  BITMAP+2     ;Blocks 16..23
201D: AD 5B BF            LDA  BITMAP+3
2020: 09 F0               ORA  #$F0         ;Block 24..27 bits = 0
2022: 8D 5B BF            STA  BITMAP+3

2025: AD C7 20            LDA  SLFAKE
2028: 0A                  ASL
2029: 0A                  ASL
202A: 0A                  ASL
202B: 0A                  ASL               ;Multiply slot by 16
202C: AC C8 20            LDY  DFAKE
202F: C0 01               CPY  #1           ;Drive 1?
2031: F0 02               BEQ  SETDS        ;Yes, so branch
2033: 09 80               ORA  #$80         ;Set bit 7 ("drive 2" bit)
2035: 8D DF 20  SETDS     STA  NEWDRSL

                * Check for existing device:
2038: AC 31 BF            LDY  DEVCNT
203B: B9 32 BF  DUPCHECK  LDA  DEVLST,Y     ;Get existing slot, drive
203E: CD DF 20            CMP  NEWDRSL      ;Same as RAMdisk slot, drive?
2041: D0 01               BNE  DC1

2043: 00                  BRK               ;Crash if duplicate found

2044: 88        DC1       DEY
2045: 10 F4               BPL  DUPCHECK     ;No, so on to next device

2047: EE 31 BF            INC  DEVCNT       ;Add "disk" drive
204A: AC 31 BF            LDY  DEVCNT
204D: AD DF 20            LDA  NEWDRSL
2050: 99 32 BF            STA  DEVLST,Y     ;Save slot, drive code

2053: AD C7 20            LDA  SLFAKE       ;Get slot #
2056: 0A                  ASL               ;x2 to step into table
2057: AC C8 20            LDY  DFAKE
205A: C0 01               CPY  #1           ;Drive 1?
205C: F0 03               BEQ  FIXTABLE     ;Yes, so branch

205E: 18                  CLC
205F: 69 10               ADC  #16          ;Offset to drive 2 table

2061: A8        FIXTABLE  TAY
2062: A9 00               LDA  #<RAMDISK    ;Save address of driver
2064: 99 10 BF            STA  DEVADR01,Y   ; in vector table
2067: A9 03               LDA  #>RAMDISK
2069: 99 11 BF            STA  DEVADR01+1,Y

                *************************************
```

Table 7-3 Continued

```
                * Change Applesoft program pointer *
                * and initialize program space.    *
                ************************************
206C: A9 01             LDA   #1
206E: 85 67             STA   TXTTAB
2070: A9 28             LDA   #$28
2072: 85 68             STA   TXTTAB+1
2074: A9 00             LDA   #0
2076: 8D 00 28          STA   $2800      ;Must begin with $00 byte
2079: 20 4B D6          JSR   $D64B      ;Applesoft "NEW" command

                **************************
                * Initialize the RAMdisk *
                **************************
207C: 20 E4 20          JSR   ZEROBLK

207F: A0 00             LDY   #0
2081: B9 C9 20  DONAME  LDA   VOLNAME,Y
2084: F0 06             BEQ   SETLEN
2086: 99 05 30          STA   INITBLK+5,Y ;Put volume name in buffer
2089: C8                INY
208A: D0 F5             BNE   DONAME

208C: 98        SETLEN  TYA
208D: 09 F0             ORA   #$F0       ;Set "directory" bits
208F: 8D 04 30          STA   INITBLK+4  ;Save file type + name length

                * Store misc. volume parameters:
2092: A0 22             LDY   #$22
2094: B9 AC 20  DOPARMS LDA   INITPARM-$22,Y
2097: 99 00 30          STA   INITBLK,Y
209A: C8                INY
209B: C0 2B             CPY   #$2B
209D: D0 F5             BNE   DOPARMS

209F: A9 02             LDA   #2
20A1: 8D E2 20          STA   BLKNUM     ;Writing to block 2
20A4: A9 00             LDA   #0
20A6: 8D E3 20          STA   BLKNUM+1
20A9: 20 D7 20          JSR   DOWRITE

                *****************************
                * Fix up the volume bit map *
                *****************************
20AC: 20 E4 20          JSR   ZEROBLK
20AF: A9 0F             LDA   #$0F       ;0..3 in use / 4..7 free
20B1: 8D 00 30          STA   INITBLK
20B4: A9 FF             LDA   #$FF       ;8..15 free
20B6: 8D 01 30          STA   INITBLK+1
```

Table 7-3 Continued

```
20B9: A9 C0              LDA  #$C0        ;16, 17 free
20BB: 8D 02 30           STA  INITBLK+2

20BE: EE E2 20           INC  BLKNUM      ;Change to block 3
20C1: 20 D7 20           JSR  DOWRITE

20C4: 4C D0 03           JMP  $3D0        ;Reconnect ProDOS hooks

20C7: 03        SLFAKE   DFB  3           ;RAMdisk slot #
20C8: 01        DFAKE    DFB  1           ;RAMdisk drive #

20C9: 52 41 4D  VOLNAME  ASC  'RAM8',00   ;Volume name
20CC: 38 00

20CE: C3        INITPARM DFB  $C3         ;Access code
20CF: 27                 DFB  $27         ;Entry length
20D0: 0D                 DFB  13          ;Entries/block
20D1: 00 00              DW   0           ;File count
20D3: 03 00              DW   3           ;Block for bit map
20D5: 12 00              DW   18          ;Total blocks

                ********************************
                * Write a block to the device *
                ********************************
20D7: 20 00 BF  DOWRITE  JSR  MLI
20DA: 81                 DFB  $81         ;WRITE_BLOCK command
20DB: DE 20              DA   CMDLIST
20DD: 60                 RTS

20DE: 03        CMDLIST  DFB  3
20DF: 00        NEWDRSL  DS   1           ;Drive and slot
20E0: 00 30              DA   INITBLK     ;I/O buffer
20E2: 00 00     BLKNUM   DW   0           ;Block # gets filled in here

                ******************
                * Zero the block *
                ******************
20E4: A9 00     ZEROBLK  LDA  #0
20E6: A8                 TAY
20E7: 99 00 30  ZB1      STA  INITBLK,Y
20EA: C8                 INY
20EB: D0 FA              BNE  ZB1
20ED: 99 00 31  ZB2      STA  INITBLK+256,Y
20F0: C8                 INY
20F1: D0 FA              BNE  ZB2
20F3: 60                 RTS

                BEGIN    EQU  *

```

Table 7-3 Continued

```
                *****************************
                * This is the device driver *
                * for the /RAM8 volume.     *
                *****************************

                        ORG   $300

                RAMDISK EQU   *

0300: D8                CLD               ;(Required by ProDOS)

                * Save zero page locations:
0301: A5 06             LDA   RAMPTR
0303: 8D 7A 03          STA   ZPTEMP
0306: A5 07             LDA   RAMPTR+1
0308: 8D 7B 03          STA   ZPTEMP+1

                *******************************
                * Check for block range error *
                *******************************
030B: A5 47             LDA   BLOCK+1     ;Check block number (high)
030D: D0 34             BNE   IOERROR     ;Error if not zero
030F: A5 46             LDA   BLOCK       ;Check block number (low)
0311: C9 12             CMP   #18         ;Is it out of bounds?
0313: B0 2E             BCS   IOERROR     ;It's >=18, so error

                **********************************
                * Convert block # to RAM address *
                **********************************
0315: 0A                ASL               ;Multiply block by 2
0316: 18                CLC
0317: 69 04             ADC   #4          ;... and add 4 to get
0319: 85 07             STA   RAMPTR+1    ;starting page of block
031B: A9 00             LDA   #0
031D: 85 06             STA   RAMPTR

                **********************
                * Check command code *
                **********************
031F: A5 42             LDA   COMMAND     ;Get command code
0321: C9 03             CMP   #3          ;Format?
0323: F0 0C             BEQ   EXIT        ;Yes, so exit normally
0325: C9 00             CMP   #0          ;Check status?
0327: F0 08             BEQ   EXIT        ;Yes, so exit normally
0329: C9 01             CMP   #1          ;Read?
032B: F0 1B             BEQ   READ        ;Yes, so branch
032D: C9 02             CMP   #2          ;Write?
032F: F0 30             BEQ   WRITE       ;Yes, so write

```

Table 7-3 Continued

```
0331: 18        EXIT     CLC                  ;CLC ==> no error
0332: A9 00              LDA   #0
0334: 08        EXIT1    PHP
0335: 48                 PHA
0336: AD 7A 03           LDA   ZPTEMP
0339: 85 06              STA   RAMPTR
033B: AD 7B 03           LDA   ZPTEMP+1
033E: 85 07              STA   RAMPTR+1
0340: 68                 PLA                  ;Restore error code
0341: 28                 PLP                  ;Restore carry status
0342: 60                 RTS

0343: 38        IOERROR  SEC                  ;SEC ==> error occurred
0344: A9 27              LDA   #$27           ;I/O ERROR code
0346: D0 EC              BNE   EXIT1          ;(always taken)

                ******************************
                * Perform READ command by    *
                * transferring data from the *
                * RAM to the data buffer.    *
                ******************************
0348: A0 00     READ     LDY   #0
034A: B1 06     FROMCARD LDA   (RAMPTR),Y
034C: 91 44              STA   (BUFFER),Y
034E: C8                 INY
034F: D0 F9              BNE   FROMCARD
0351: E6 07              INC   RAMPTR+1
0353: E6 45              INC   BUFFER+1
0355: B1 06     FC1      LDA   (RAMPTR),Y
0357: 91 44              STA   (BUFFER),Y
0359: C8                 INY
035A: D0 F9              BNE   FC1
035C: C6 45              DEC   BUFFER+1
035E: 4C 31 03           JMP   EXIT

                ******************************
                * Perform WRITE command by   *
                * transferring data from the *
                * data buffer to the RAMcard.*
                ******************************
0361: A0 00     WRITE    LDY   #0
0363: B1 44     TOCARD   LDA   (BUFFER),Y
0365: 91 06              STA   (RAMPTR),Y
0367: C8                 INY
0368: D0 F9              BNE   TOCARD
036A: E6 45              INC   BUFFER+1
036C: E6 07              INC   RAMPTR+1
036E: B1 44     TC1      LDA   (BUFFER),Y
0370: 91 06              STA   (RAMPTR),Y
```

Table 7-3 Continued

```
0372: C8                 INY
0373: D0 F9              BNE   TC1
0375: C6 45              DEC   BUFFER+1
0377: 4C 31 03           JMP   EXIT

037A: 00 00     ZPTEMP   DS    2

                END      EQU   *
```

When 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 this screen uses ($2000–$3FFF) overlaps the /RAM8 block storage area. Moreover, the Applesoft program must not overwrite the device driver in page 3, or the storage space itself, with POKE statements. If you want to avoid these memory conflicts, you can relocate the disk driver (and its corresponding storage 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 using the technique described above for the removal of the /RAM volume. You will also 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.

CHAPTER 8

Clocks

In Chapter 2, we saw that the directory entry for each file on a disk formatted for the ProDOS file system contains 4 bytes for the time and date the file was created and 4 more bytes for the time and date it was last modified. Most other file systems save similar time and date information.

The ProDOS file system's date-stamping feature is very useful, especially for those who routinely save several versions of the same file on different disks. Three months later you won't have to guess which one is the latest version; all you have to do is compare modification dates. The BASIC.SYSTEM CATALOG command displays these dates when it lists the names of the files on disk.

GS/OS and ProDOS 8 determine the current time and date by accessing a real-time clock/calendar chip interfaced to the microprocessor. On the IIGS, this chip is an integral part of the system and does not occupy a slot or port; on the IIe and II Plus, you must add an optional clock card. There are also clocks available for the slotless IIc.

A computer clock contains special integrated circuits that allow it to keep track of the current time and date independently of the microprocessor. It is the Apple's digital watch, if you like. Clocks keep the correct time even when the Apple is turned off because they are powered by batteries.

ProDOS 8 uses a special assembly-language program, called a *clock driver*, to transfer the time and date from the card to the Apple in an understandable form. ProDOS 8 comes with internal clock drivers for the built-in IIGS clock and for any clock card that understands a standard set of time-related commands originally used in Thunderware's Thunderclock. ProDOS 8 automatically installs the correct driver into the system when it first boots up. If there is no recognizable clock, ProDOS 8 installs a null driver, and application programs should ask the user to enter the correct time and date if that information is needed. GS/OS always installs a driver for the built-in clock on the Apple IIGS.

In this chapter, we examine how ProDOS 8 deals with time issues in general. In particular, we see how it detects the presence of a clock card, how it installs the clock driver, and how to design and install your own ProDOS 8 clock driver for a nonstandard clock. (Since GS/OS has a built-in driver for the IIGS clock, you will never have to install your own driver; therefore GS/OS has no mechanism for installing custom clock drivers.) We also go through some useful examples of how to make the most of the time and date capabilities of GS/OS and ProDOS 8.

HOW GS/OS AND PRODOS 8 READ THE TIME AND DATE

Whenever ProDOS 8 needs to know the time and date it always makes the same call: JSR DATETIME. The code starting at DATETIME ($BF06) is either a 1-byte RTS instruction (if no ProDOS-compatible clock is in the system) or a 3-byte JMP instruction that passes control to a ProDOS 8 clock driver (if a compatible clock is present). In either case, the 2 bytes at $BF07–$BF08 always hold the address of the start of the ProDOS 8 clock driver space.

The clock driver reads the time and date from the clock and stores the data in a special format at TIME ($BF92–$BF93) and DATE ($BF90–$BF91) in the ProDOS 8 global page. Figure 8-1 describes the format used. If no clock driver is present, TIME and DATE are not modified because the RTS instruction stored at DATETIME ($BF06) immediately bounces control back to the caller. The only way to set the time and date in this situation is to write directly to the TIME and DATE locations.

The approved method of determining the date and time in a ProDOS 8 application is to use the GET_TIME command. Recall from Chapter 4 that you can do this by executing a subroutine like this one:

```
JSR $BF00        ;Make a call to the MLI
DFB $82          ;GET_TIME
DA  $0000        ;Dummy parameter table
RTS
```

When this subroutine finishes, the TIME and DATE locations contain the current time and date in the format described above.

GS/OS has no equivalent operating system command for returning the current time and date. If you want the time and date, you must use two commands in the Apple IIGS Miscellaneous Tool Set: ReadTimeHex and ReadAsciiTime.

ReadTimeHex (toolbox command $0D03) returns the current time and date parameters as binary numbers. Here's how to call it from 65816 full native mode:

```
PHA              ;Space for results
PHA              ; (eight bytes)
PHA
PHA
LDX  #$0D03      ;ReadTimeHex
JSL  $E10000
PLA              ;WeekDay (high)
PLA              ;Month (high), Day (low)
PLA              ;CurYear (high), Hour (low)
PLA              ;Minute (high), Second (low)
```

Figure 8-1 The formats of the ProDOS 8 DATE and TIME bytes

(a) DATE ($BF90–$BF91)

7	6	5	4	3	2	1	0	
Y6	Y5	Y4	Y3	Y2	Y1	Y0	M3	$BF91

7	6	5	4	3	2	1	0	
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, 89 for 1989).

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, November 30, 1989, would be stored as follows:

```
High-order byte     Low-order byte
---------------     --------------
1 0 1 1 0 0 1 1     0 1 1 1 1 1 1 0
YYYYYYYYYYYYY M     MMMMM DDDDDDDDD

  Year ($59)  Month ($0B)  Day ($1E)
    1989       November       30
```

(b) TIME ($BF92–$BF93)

7	6	5	4	3	2	1	0	
0	0	0	H4	H3	H2	H1	H0	$BF93

7	6	5	4	3	2	1	0	
0	0	M5	M4	M3	M2	M1	M0	$BF92

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, 9:20 p.m. (21:20) would be stored as follows:

```
High-order byte     Low-order byte
---------------     --------------
0 0 0 1 0 1 0 1     0 0 0 1 0 1 0 0
      HHHHHHHHH         MMMMMMMMMMM

    Hours ($15)       Minutes ($14)
      21                 20
```

The values ReadTimeHex returns are as follows:

WeekDay	1..7	1 = Sunday, 2 = Monday, and so on
Month	0..11	0 = January, 1 = February, and so on
Day	0..30	day of month minus 1
CurYear	0..99	current year minus 1900
Hour	0..23	hour in military format
Minute	0..59	
Second	0..59	

ReadAsciiTime (toolbox command $0F03) returns a 20-byte ASCII-encoded character string describing the current time and date. Here is how to call it:

```
            PushPtr TimeString ;Pointer to string area
            LDX #$0F03         ;ReadAsciiTime
            JSL $E10000
            RTS

TimeString  DS 20              ;Space for time string
```

Note that the time string returned is *not* preceded by a length byte. The string is always exactly 20 bytes long, and the high-order bit of each byte is set to 1.

The format of the time string depends on the settings of the date and time formats in the Control Panel. There are six possibilities:

```
mm/dd/yy HH:MM:SS XM     XM = AM or PM
dd/mm/yy HH:MM:SS XM
yy/mm/dd HH:MM:SS XM
mm/dd/yy HH:MM:SS        24-hour military format
dd/mm/yy HH:MM:SS
yy/mm/dd HH:MM:SS
```

The first format listed here is the Control Panel's default.

HOW PRODOS 8 IDENTIFIES A CLOCK CARD

When you first boot ProDOS 8 on a system other than the IIGS, ProDOS 8 examines each peripheral expansion slot in the system for a standard clock card. ProDOS 8 identifies such a card by the following unique pattern of bytes in the card's dedicated $Cn00–$CnFF ROM space (n is the slot number):

$Cn00	$08
$Cn02	$28
$Cn04	$58
$Cn06	$70

If it finds a clock card, ProDOS 8 installs its standard clock driver and changes the RTS opcode ($60) at $BF06 to a JMP opcode ($4C). Since the 2 bytes following this opcode contain the address of the clock driver space (low-order byte first), the driver takes control whenever a program executes a JSR $BF06 instruction. Actually, a program should always use the GET_TIME command to read the time and date; the GET_TIME command handler is what calls the clock driver directly.

The built-in IIGS clock does not occupy a slot or port, so ProDOS 8 can't identify it by checking bytes in ROM. Instead, it simply checks to see what Apple II model it is running on; if it's a IIGS, it installs the IIGS clock driver.

ProDOS 8 also sets the clock bit (bit 0) of the machine identification byte, MACHID ($BF98), to 1 if it finds a clock.

WRITING AND INSTALLING A PRODOS 8 CLOCK DRIVER

If you are using a nonstandard clock, you must write and install your own ProDOS 8 clock driver. Two examples of nonstandard clocks are a clock interfaced through the serial port of a IIc and a clock on a multifunction peripheral card that does not occupy a phantom slot.

Writing a clock driver is no easy feat since it requires detailed information concerning how the clock circuitry is interfaced to the Apple and the procedure a programmer must follow to extract time and date information from the card. If you're lucky, the manufacturer of the card will have a detailed technical reference manual that contains this information. But more commonly you will have to beg, borrow, or steal this information before you can get started. Happily, manufacturers of nonstandard clock cards have already written their own ProDOS 8 clock drivers and include them on disk 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 you write a driver, you must move it to an area of memory that other programs will not use. The best available area is the one the very clock driver you are replacing uses; you can always find the starting address of this area at $BF07–$BF08 (low-order byte first).

If you choose to use the standard driver area (and we do recommend this selection), keep several important considerations in mind:

- Never assume the standard clock driver will reside at the same position in every version of ProDOS 8. To ensure 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 need to patch it to resolve all internal absolute address references after you move it to its new position.

- Make sure your clock driver is no longer than 125 bytes. ProDOS 8 reserves this amount of space for its standard drivers, 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 standard clock 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. Do this 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 standard clock 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 has been installed in the system. Do this by 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, which automatically runs when ProDOS 8 executes the BASIC.-SYSTEM Applesoft interpreter.

TIME/DATE UTILITY PROGRAMS

An Applesoft Time and Date 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 a program needs to display the current time, automatically time-stamp printed reports, calculate elapsed times, perform benchmarking studies, and so on.

You can use the READ.TIME subroutine in Table 8-1 to return the time and date in the form *DD-MM-19YY HH:MM* in any Applesoft string variable you specify. After loading the subroutine, use it by executing the following statement from within an Applesoft program:

```
CALL 768,TM$
```

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

```
                *************************************
                *            READ.TIME              *
                *                                   *
                * This program reads the time and   *
                * date and stores it in an Applesoft *
                * string variable. The syntax is    *
                *                                   *
                *          CALL 768,TM$             *
                *                                   *
                * The TM$ string has the form       *
                *         DD-MM-19YY HH:MM          *
                *                                   *
                * Copyright 1985-1988 Gary B. Little *
                *                                   *
                * Last modified: August 28, 1988    *
                *                                   *
                *************************************
                FRETOP   EQU   $6F         ;Bottom of string space
                VARPNT   EQU   $83         ;Pointer to string data

                IN       EQU   $200        ;Input buffer

                MLI      EQU   $BF00       ;Entry point to MLI
                DATE     EQU   $BF90       ;Year + Month + Day
                TIME     EQU   $BF92       ;Minutes + Hours

                CHKCOM   EQU   $DEBE       ;Skip comma
                PTRGET   EQU   $DFE3       ;Locate a variable
                GETSPACE EQU   $E452       ;Get string space for "A" chars
                MOVSTR   EQU   $E5E2       ;Move string to free space

                         ORG   $300

0300: 20 00 BF           JSR   MLI         ;Call the MLI and
0303: 82                 DFB   $82         ; select GET_TIME command
0304: 00 00              DA    $0000       ;(no parameter table)

                * "Unpack" the time:

0306: AD 92 BF           LDA   TIME        ;Get minutes
0309: 8D B8 03           STA   MINUTES     ; and save them
030C: AD 93 BF           LDA   TIME+1      ;Get hours
030F: 8D B9 03           STA   HOURS       ; and save them
0312: AD 90 BF           LDA   DATE        ;Get "day" bits (0...4),
0315: 29 1F              AND   #$1F        ; strip "month" bits,
0317: 8D BA 03           STA   DAY         ; and store correct number
031A: AD 91 BF           LDA   DATE+1      ;Get "year" bits (1...7)
031D: 8D BC 03           STA   YEAR        ; and month bit (0).
```

Table 8-1 Continued

```
0320: AD 90 BF           LDA   DATE        ;Get month bits (5...7)
0323: 4E BC 03           LSR   YEAR        ;Put "year" bits into 0...6
0326: 6A                 ROR               ;Get "month" bits in one byte
0327: 4A                 LSR               ; and move them into
0328: 4A                 LSR               ; the lower 5 bits
0329: 4A                 LSR
032A: 4A                 LSR
032B: 8D BB 03           STA   MONTH       ;Save month bits (0...4)

               * Assemble the Applesoft time/date string:

032E: A2 00              LDX   #0
0330: 8E B7 03           STX   TIMEPOS     ;Clear ptr to time string
0333: 8A       FORMTIME TXA
0334: 48                 PHA
0335: BD BD 03           LDA   FORMAT,X    ;Get formatting byte
0338: 08                 PHP
0339: AE B7 03           LDX   TIMEPOS
033C: 28                 PLP
033D: 30 1E              BMI   NOTNUM      ;Branch if not number
033F: A8                 TAY               ;Get time code in Y

0340: B9 B8 03           LDA   TIMEDATA,Y ;Get binary time/date data
0343: 20 92 03           JSR   CONVERT     ;Convert to BCD
0346: 48                 PHA               ;Save number
0347: 4A                 LSR               ;Move "tens" digit to
0348: 4A                 LSR               ; lower 4 bits by
0349: 4A                 LSR               ; shifting right four
034A: 4A                 LSR               ; times
034B: 09 30              ORA   #$30        ;Convert to ASCII digit
034D: 9D 00 02           STA   IN,X
0350: E8                 INX
0351: 68                 PLA               ;Get original number back
0352: 29 0F              AND   #$0F        ;Isolate units digit
0354: 09 30              ORA   #$30        ;Convert to ASCII digit
0356: 9D 00 02           STA   IN,X
0359: E8                 INX
035A: 4C 63 03           JMP   TONEXT

035D: 29 7F    NOTNUM    AND   #$7F        ;Strip high bit for Applesoft
035F: 9D 00 02           STA   IN,X        ;Insert punctuation
0362: E8                 INX
0363: 8E B7 03 TONEXT    STX   TIMEPOS
0366: 68                 PLA
0367: AA                 TAX
0368: E8                 INX               ;Go to next position
0369: E0 0C              CPX   #12         ;At end of template?
036B: D0 C6              BNE   FORMTIME    ;No, so keep going

```

Table 8-1 Continued

```
                 * Move string to bottom of string space:

0360: AD B7 03            LDA  TIMEPOS    ;Get length of string
0370: 20 52 E4            JSR  GETSPACE   ;Make room for it
0373: A2 00               LDX  #0
0375: A0 02               LDY  #2         ;Y/X point to string
0377: 20 E2 E5            JSR  MOVSTR     ;Move the string (length in A)

                 * Point Applesoft variable to time/date string.
                 * The string is now positioned at the bottom
                 * of string space and is pointed to by FRETOP.

037A: 20 BE DE            JSR  CHKCOM     ;Skip over comma
037D: 20 E3 DF            JSR  PTRGET
0380: AD B7 03            LDA  TIMEPOS    ;Get length of string
0383: A0 00               LDY  #0
0385: 91 83               STA  (VARPNT),Y ;... and save it
0387: C8                  INY
0388: A5 6F               LDA  FRETOP
038A: 91 83               STA  (VARPNT),Y ;Save address (low)
038C: C8                  INY
038D: A5 70               LDA  FRETOP+1
038F: 91 83               STA  (VARPNT),Y ;Save address (high)
0391: 60                  RTS

                 ****************************
                 * Binary to BCD Conversion *
                 * Number must be 0...99    *
                 ****************************
0392: 8D B6 03   CONVERT  STA  TEMP       ;Put # into work area
0395: 8E B5 03            STX  XSAVE
0398: A9 00               LDA  #0         ;Start with a 0 result
039A: F8                  SED             ;Use decimal arithmetic
039B: A2 06               LDX  #6         ;Examine bits 0...6
039D: 4E B6 03   NEXTBIT  LSR  TEMP       ;Move low bit into carry
03A0: 90 04               BCC  NOWEIGHT   ;Branch if it was zero
03A2: 18                  CLC             ; else add it
03A3: 7D AE 03            ADC  BINDEC,X   ; to result
03A6: CA         NOWEIGHT DEX             ;Count down to -1
03A7: 10 F4               BPL  NEXTBIT    ;Branch if more to go
03A9: D8                  CLD             ;Return to binary arithmetic
03AA: AE B5 03            LDX  XSAVE
03AD: 60                  RTS

03AE: 64 32 16   BINDEC   DFB  $64,$32,$16 ;These are the weights of
03B1: 08 04 02            DFB  $08,$04,$02 ;the low 7 bits in
03B4: 01                  DFB  $01        ;a byte (in BCD)

03B5: 00         XSAVE    DS   1          ;Temporary X location
```

Table 8-1 Continued

```
03B6: 00         TEMP     DS    1         ;Temporary work area
03B7: 00         TIMEPOS  DS    1

                 TIMEDATA EQU   *

03B8: 00         MINUTES  DS    1         ;Minutes (0...59)
03B9: 00         HOURS    DS    1         ;Hours (0...23)
03BA: 00         DAY      DS    1         ;Day of month (1...31)
03BB: 00         MONTH    DS    1         ;Month of year (1...12)
03BC: 00         YEAR     DS    1         ;Year (0...99)

                 * Formatting template for "DD-MM-19YY HH:MM"
                 * (digits refer to entries in TIMEDATA table)

                 FORMAT   EQU   *

03BD: 02                  DFB   2
03BE: AD 03 AD            DFB   "-",3,"-"
03C1: B1 B9 04            DFB   "1","9",4
03C4: A0 A0 01            DFB   $A0,$A0,1
03C7: BA 00               DFB   ":",0
```

When you call READ.TIME, it first uses the ProDOS 8 GET_TIME command to read the current time and date into the ProDOS 8 global page locations. It then unpacks the year, month, and day data from the DATE locations and stores each of them in its own temporary location. The hours and minutes are already unpacked, but they are also transferred to temporary locations.

After unpacking, 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.

Next, READ.TIME moves the string from the input buffer to the main Applesoft string space in the high end of memory to ensure the string will not be overwritten the next time your program executes an Applesoft INPUT statement. This is done using two Applesoft ROM subroutines called GETSPACE ($E4B2) and MOVSTR ($E5E2). When you call GETSPACE 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, 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 1 byte, effectively skipping over the comma separating the CALL address from the variable. The second instruction stores the address of the 3-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 2 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 2 bytes.

Setting the Time and Date on a Clockless Apple

Even if you do not have a clock in your Apple II, you can still date- and time-stamp a file by explicitly storing the current date and time in the ProDOS 8 global page locations just before saving the file to disk. This is somewhat inconvenient, but it's better than nothing. If you can survive with just the correct date, life becomes much easier because you have to set the date only once when you first turn the computer on (assuming, perhaps naively, that you don't work past midnight).

The TIMEDATE program in Table 8-2 lets you enter a time and date in English. After you do so, the program converts the information into the encoded format used by ProDOS 8 and then stores it in the ProDOS 8 global page locations.

Table 8-2 TIMEDATE, a program to manually set the time and date.

```
1   REM "TIMEDATE"
2   REM COPYRIGHT 1985-1987 GARY B. LITTLE
3   REM DECEMBER 21, 1987
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 1985-1987 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 $BF91 (YYYYYYYM)
400  VTAB 6: CALL - 958: INPUT "ENTER YEAR (1900-1999): 19";A$:
     YR = VAL (A$): IF YR < 0 OR YR > 99 OR A$ = "" THEN 400
500  VTAB 7: CALL - 958: INPUT "ENTER MONTH (JAN...DEC): ";A$:M$ = ""
501  IF A$ = "" THEN 500
505  FOR I = 1 TO LEN (A$): IF ASC ( MID$ (A$,I,1)) > = 96 THEN
     B$ = B$ + CHR$ ( ASC ( MID$ (A$,I,1)) - 32): GOTO 507
506 B$ = B$ + MID$ (A$,I,1)
507  NEXT :A$ = B$
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
1050  HOME : PRINT "THE TIME AND DATE HAVE NOW BEEN SET."
```

CHAPTER 9

GS/OS Character Devices

An important feature of GS/OS is that you can use its commands to communicate with character devices, not just block-structured disk devices. For example, to get keyboard input, you open the keyboard, read data from it, then close it, just as if it were a file on a disk drive. Under ProDOS 8, you must use completely different techniques to access character devices, such as accessing memory-mapped hardware addresses or calling firmware subroutines.

The character FST is responsible for translating standard GS/OS commands into commands that the driver for a character device understands. It resides in a file called CHAR.FST in the SYSTEM/FSTS/ subdirectory of the boot disk.

In this chapter, we see how to use GS/OS commands to communicate with two particularly important character devices: the keyboard and the video display screen. The device driver that controls these devices is called the Console Driver; we also investigate the commands this driver understands.

Note: The Apple IIGS has a tool set, called the Text Tool Set, that you can also use to access character devices. But you should use the GS/OS commands since they are more powerful and easier to use.

GS/OS COMMANDS FOR CHARACTER DEVICES

The character FST works with a small subset of GS/OS commands: Open, NewLine, Read, Write, Close, and Flush. (You shouldn't use NewLine, however, because the Console Driver supports a more powerful way of terminating input prematurely; see the discussion of terminator characters below.) You can also use the GS/OS device commands, DInfo, DControl, DRead, DStatus, and DWrite, to communicate directly with any character-based device driver, including the Console Driver.

The name of the Console Driver is usually .CONSOLE, but the user may be able change it when a GS/OS driver configuration program becomes available. To determine the actual name, call the DInfo command with successively higher device numbers (starting with 1) until DInfo returns a device_ID_num of $000A. The name that DInfo returns for the device with this device_ID_num is the actual name of the Console Driver.

DControl and DStatus are important for setting and returning various parameters and operating mode flags the Console Driver uses. We summarize the DControl and DStatus commands near the end of this chapter.

You won't need to use DRead and DWrite to communicate with the Console Driver (you can use Read and Write instead), so they are not described here.

KEYBOARD INPUT

The Console Driver deals with character input from the Apple IIGS keyboard. It reads data directly from the keyboard hardware or, if the IIGS Event Manager is active, from the operating system event queue. The Console Driver returns standard ASCII character codes (bit 7 of each code is zero).

The Console Driver supports two main input modes: *raw mode* and *user input mode*.

In raw mode, the driver continuously polls for keyboard data until it has read in the number of characters requested in the Read command parameter table or until the user enters a terminator character. (More on terminator characters below.) It then returns these characters, including any terminator character, in the Read command's data buffer. During a raw mode input operation, no cursor appears on the screen, and characters are not echoed on the screen. Raw mode is useful for programs that wish to implement their own user input and editing routines.

In user input mode, the driver uses an intelligent User Input Routine (UIR) to return keyboard input. The UIR displays an input field and a cursor, echoes input, and permits editing according to Apple's human-interface guidelines. An input operation ends when the user enters a terminator character.

To begin a keyboard input operation, you must first open the "file" called .CONSOLE using the GS/OS Open command. After doing this, set up various input parameters and the appropriate input mode, as follows:

1. Select wait or no-wait mode. When wait mode is active, GS/OS keeps processing a Read command until the user has typed in the specified number of characters from the keyboard (in raw mode) or until the user enters a terminator character (in raw or UIR mode). When no-wait raw mode is active, GS/OS returns control to the application as soon as it determines there is no keyboard input available. (The UIR always operates in wait mode, so control never returns until the user enters a terminator character.) This gives the application a chance to perform other tasks during a keyboard input operation, but the application must keep making Read calls until the user enters a terminator character. The default mode is wait mode; to switch to no-wait mode, use the GS/OS DControl command.

2. Set up the input port. The input port is a 17-byte record that keeps track of the status of a UIR input operation. When you open the Console Driver, GS/OS sets up a default input port suitable for most input operations. If you want to change some of the entries in the port, for example, to set the initial cursor position and mode, now is the time to do it. The procedure to follow is to read in a copy of the current input port (with DStatus), change the desired fields, and then set the new input port (with DControl). A description of the fields in the input port appears below.
3. Set up the terminator characters. A terminator character is one that, when entered, causes a Read operation to end. The Console Driver lets you specify the terminator character and the combination of modifier keys that must be held down when the user enters it. When using the UIR, the application *must* set up a terminator character, typically the Return key, or the user won't be able to end an input operation. You can set up a list of terminator characters with the DControl command.
4. Set up the default string. The UIR displays a default string in the input field when you call the Read command for the first time after an Open. Use the DControl command to set up the default string.

Once these preliminary steps are out of the way, use the Read command (with the reference number set to the one returned by Open) to return the number of characters specified in the request_count field of its parameter table.

On return from the Read command, use DStatus to get a copy of the input port. The exit_type field of this port (see below) indicates the reason for the return of control. In normal raw mode, a $00 value indicates that the specified number of characters has been returned, so input processing can end. In no-wait raw mode, a $00 indicates a no-wait return, and the application must inspect the transfer_count field to determine if any more characters have to be processed; if so, it must process them, then call the Read command again (after reducing request_count) until the desired number of characters have been returned.

Any other value for exit_type, in raw mode or UIR mode, indicates that a terminator character was pressed. If the value corresponds to an application-defined interrupt key (see below), you should process it without disturbing the current UIR environment, and then call the Read command again. (When you call Read again in UIR mode, you don't have to make any adjustments to the parameter table because the Console Driver keeps track of the state of the input operation when it was last exited.) If you wish to abort the input operation instead, use DControl's Abort Input subcommand. This subcommand zeroes the entry_type field of the input port (see below) so that the next Read command will not be interpreted as a continuation of the previous one.

If a non-zero exit_type value does not correspond to an interrupt key, the input operation is complete. The Console Driver handles the next Read command as an initial entry to UIR mode.

When you're through reading keyboard input, call the Close command. This is not necessary, however, if you still need the Console Driver to process video output or more input.

The Input Port

As we mentioned, the Console Driver maintains an input port to keep track of the input environment. The fields in the 17-byte input port are arranged in the following order:

fill_char

def_cursor

cursor_mode

beep_flag

entry_type

exit_type

last_char

last_mod

last_term_ch

last_term_mod

cursor_pos

input_length

input_field

origin_h

origin_x1

origin_x2

origin_v

The values in these fields completely describe the input environment. Here is what each field means:

fill_char. This is the character code that UIR sends to the Console Driver when it wants to display an empty position in the input field. The default value is $20 (a space).

def_cursor. Three bits in this byte indicate the default cursor mode at the beginning of a UIR session:

```
bit 7    0 = put cursor at end of default string
         1 = put cursor at beginning of default string
bit 6    0 = don't allow the entry of control characters
         1 = allow the entry of control characters
bit 0    0 = use an insert cursor
         1 = use an overstrike cursor
```

The default value is $00.

cursor_mode. One bit in this byte indicates the current cursor status in UIR mode:

```
bit 0    0 = an insert cursor is active
         1 = an overstrike cursor is active
```

beep_flag. If this byte is nonzero (the default value), the UIR beeps if the user attempts an illegal operation. If this byte is zero, there is no beep.

entry_type. When the application calls the Read command, the Console Driver inspects this byte to determine the current input status. The possible values are

```
$00    this is the initial entry
$01    this is an interrupt key reentry
$02    this is a no-wait mode reentry (raw mode only)
```

The Console Driver adjusts this byte whenever it relinquishes control to the application, setting it to $00 if a noninterrupt terminator character was entered. This enables the Console Driver to properly restart a Read operation that is already in progress.

exit_type. This byte indicates the reason for the exit from the Read request:

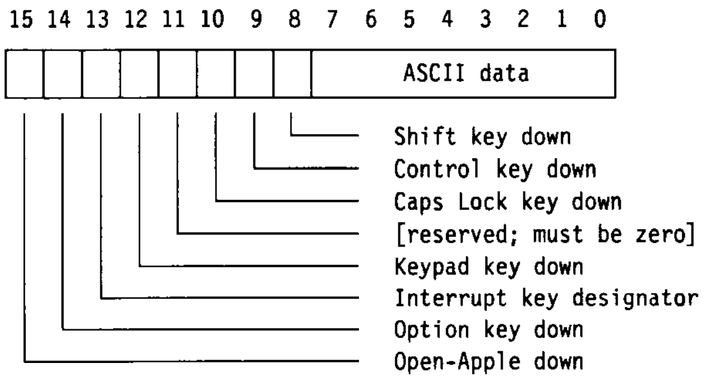
```
$00    a raw-mode exit, because the maximum number of
       characters have been read, or a no-wait raw mode
       exit
```

A nonzero value indicates a terminator key was pressed. The value is the entry number of the terminator character in the terminator table. If the terminator is not an interrupt key, the Console Driver zeroes the entry_type field so that the next Read operation will begin from scratch; otherwise, it puts a $01 there so that the Console Driver will continue the same input operation the next time the application calls Read.

last_char. The ASCII code of the most recently typed key. The high-order bit is always 0.

last_mod. The modifier byte of the most recently typed key. The meanings of the bits in the modifier byte are the same as those for the bits in the high-order byte of a terminator modifier (see Figure 9-1).

Figure 9-1 The format of the terminator mask and the terminator modifier word



last_term_ch. The ASCII code of the most recently typed terminator key. The high-order bit is always 0.

last_term_mod. The modifier byte of the most recently typed terminator key. The meanings of the bits in the modifier byte are the same as those for the bits in the high-order byte of a terminator modifier (see Figure 9-1).

cursor_pos. The position of the cursor relative to the start of the UIR input field. A $00 value means the cursor is over the first character in the field. The maximum value is the length of the field, meaning the cursor can move to the first character past the end of the field.

input_length. The current length of the string being edited. This is the same as the number returned in the transfer_count field of the Read command.

input_field. This value is for the Console Driver's private use.

origin_h. The horizontal position of the cursor in UIR mode.

origin_x1. This value is for the Console Driver's private use.

origin_x2. This value is for the Console Driver's private use.

origin_v. The vertical position of the cursor in UIR mode.

UIR Editing

The UIR supports several standard commands for editing the characters in the input field:

left-arrow	Move the cursor one position to the left.
-left-arrow	Move the cursor to the start of the previous word (if it's currently over a space) or to the start of the current word (if it's not).
right-arrow	Move the cursor one position to the right.
-right-arrow	Move the cursor to the end of the next word (if it's currently over a space) or to the end of the current word (if it's not).
-> or -.	Move the cursor to the end of the input field.
-< or -,	Move the cursor to the beginning of the input field.
-E or Control-E	Toggle the cursor between insert mode (blinking underscore) and overstrike mode (blinking box).
Delete or Control-D or Control-Delete or -Delete or -D	Erase the character to the left of the cursor and move the characters beneath and to the right of the cursor one position to the left. The cursor also moves one position to the left.
-F or Control-F	Erase the character underneath the cursor and move the characters to the right of the cursor one position to the left. The cursor stays put.
-X or Control-X or Clear	Erase the entire input field.
-Y or Control-Y	Erase the characters from the current cursor position to the end of the input field.
-Z or Control-Z	Restore the default input string.
-Control-<char>	Enter a control character. You can do this only if control character entry is enabled by setting bit 6 of the def_cursor field in the input port record.

Terminator Characters

A terminator character is one that when entered, causes a raw mode Read operation to end even if the user has not yet entered the number of characters specified in the request_count field of the Read command. Entering a terminator character also forces a UIR operation to end right away. (In fact, the user must end a UIR operation by entering a terminator character, so the application must define at least one such character.) The transfer_count field in the Read parameter table contains the actual number of characters that Read has returned in the data_buffer field.

When the user enters a terminator character, the exit_type field in the input port is set to the position number of the terminator character in the terminator list. The position number of the first item in the list is $01.

The Console Driver lets you specify the terminator character itself, as well as the modifier keys (Open Apple, Shift, Caps Lock, and so on) that the user must hold down while entering the character. It uses a data structure called a terminator list to hold the definitions of up to 254 terminator characters and their modifiers. The list begins with a terminator mask and a terminator count and is followed by the terminator characters and their modifiers.

Here is the meaning of each entry in a terminator list:

Terminator Mask (word). When the user enters a keystroke, the Console Driver logically ANDs the keystroke data with the terminator mask before checking the list of terminator modifiers for a match. By setting bits of the mask to zero, you can force matches even if the associated modifier keys are being pressed. (Figure 9-1 shows the meaning of the bits in a terminator mask.) If the state of the Caps Lock key is unimportant to your application, for example, you would specify a mask of $FBFF (bit 10 = 0).

Terminator Count (word). This word contains the number of entries in the list of terminator modifiers. If there are no terminators, this word should be set to zero.

Terminator Modifiers (words). A terminator modifier is a 2-byte value describing the ASCII code of the terminator (low byte) and the modifiers themselves (high byte). Figure 9-1 shows the meaning of each of the bits in a terminator modifier.

If bit 13, the interrupt bit, of a terminator modifier is set to 1, the terminator character is considered an interrupt key. When the user enters an interrupt key, the Read command ends, but the entry_type byte in the input port is set to $01. The next time the same Read command is called, input processing continues from where the interruption took place.

One reason to define an interrupt key is to implement a help command. To include a standard -? help key, for example, set bits 15 and 13 in the modifiers byte and put the ASCII code for a question mark in the low-order byte. You should also assign -/ as an interrupt key so that the user can get help without having to press a Shift key (? and / share the same keycap).

VIDEO OUTPUT

The Console Driver also manages all activities related to the display of characters on the Apple IIGS text screen. There are actually two text screens: an 80-column, 24-line screen and a 40-column, 24-line screen; you can switch between them by sending control codes to the Console Driver with the GS/OS Write command.

The Console Driver stores video data directly to the video RAM buffers located at $0400–$07FF in banks $E0 and $E1 of memory. As a result, applications that want to access the screen bytes directly should not look at the "traditional" video RAM buffers in banks $00 and $01 even if these areas are set up to shadow to banks $E0 and $E1. See *Exploring the Apple IIGS* for a discussion of text screen shadowing.

The Console Driver lets you confine video output operations to any rectangular window within the full hardware screen; this window is called a *text port*. When you first

Open the .CONSOLE device, the Console Driver sets the boundaries of the text port to the full 80-column text screen; you can change the boundaries with a control code.

The text port keeps track of all important screen-related parameters, including the dimensions of the text port, the cursor position, and the settings of various cursor movement parameters. The cursor position marks where the Console Driver will display the next outputted character. It is set to the top left-hand corner of the text port when you first open the .CONSOLE device.

To display a character on the screen, use the GS/OS Write command to send the character code to the Console Driver. Table 9-1 indicates these character codes in both normal and inverse modes. (In normal mode, the characters are white, and the background is black; in inverse mode, the characters are black, and the background is white. If MouseText mapping is enabled, MouseText symbols appear instead of inverse uppercase characters.)

Notice that you can display inverse characters and MouseText symbols without explicitly enabling inverse mode or MouseText mapping. Just send character codes with the high-order bit set to 1.

Control Commands

As Table 9-1 indicates, the codes from $00 to $1F do not correspond to visible screen characters. Instead, they represent commands to the Console Driver to perform special screen-related tasks such as clearing portions of the text port, positioning the cursor, scrolling the text port, and enabling MouseText mapping. Table 9-2 gives a complete list of these commands.

Many of the commands in Table 9-2 simply involve sending the corresponding code through the output stream. But some require you to follow the code with one or more data bytes. In general, the values of these data bytes are 32 higher than the value you are trying to set.

Some of the commands in Table 9-2 refer to global flags in the text port record called cons_DLE, cons_scroll, cons_wrap, cons_LF, and cons_advance. The settings of these variables govern how some cursor movement and scrolling operations are to be performed. To set these flags, use the Set Cursor Movement command (control code $15).

Multiple Windows

The Console Driver facilitates the development of multiwindow text-screen applications because it has commands for saving and restoring a text port. To create a second text port, for example, use the Push and Reset Text Port ($01) command; then set the dimensions and characteristics of the new text port. To switch back to the original text port, use the Pop Text Port ($04) command.

If the text ports overlap, you must also save and restore the screen data for the text port that is about to be inactivated. You can do this with the Return Text Port Data subcommand of the GS/OS DStatus command. To put the data back in the text port,

Table 9-1 Character codes used by the Console Driver[a]

Character Code	*Normal Mode*	*Inverse Mode*
\$00–\$1F	Control commands	Control commands
\$20–\$3F	Normal symbols, digits (\$A0–\$BF)[b]	Inverse symbols, digits (\$20–\$3F)
\$40–\$5F	Normal uppercase (\$80–\$9F)	Inverse uppercase (\$00–\$1F)[c]
\$60–\$7F	Normal lowercase (\$E0–\$FF)	Inverse lowercase (\$60–\$7F)
\$80–\$9F	Inverse uppercase (\$00–\$1F)	Normal uppercase (\$80–\$9F)
\$A0–\$BF	Inverse symbols, digits (\$20–\$3F)	Normal symbols, digits (\$A0–\$BF)
\$C0–\$DF	MouseText symbols (\$40–\$5F)	Normal uppercase (\$C0–\$DF)
\$E0–\$FF	Inverse lowercase (\$60–7F)	Normal lowercase (\$E0–\$FF)

NOTES:
[a]The exact sequence of characters from \$20 to \$7F is the same as the sequence defined by the ASCII standard.
[b]The numbers in parentheses indicate the values the Console Driver actually stores in the video RAM buffer.
[c]When MouseText mapping is on, MouseText symbols appear for character codes \$40–\$5F in inverse mode instead of inverse uppercase characters.

use the Restore Text Port Data subcommand of the GS/OS DControl command. These subcommands are described in the next section.

DEVICE COMMANDS

In this section, we summarize the DControl and DStatus subcommands you use to communicate with the Console Driver. As we mentioned, these subcommands are for setting up the character input/output environment and returning status information.

DControl and DStatus may return two possible errors:

```
$22    bad driver parameter
$23    the Console Driver is not open
```

DControl returns error \$22 if the amount of data in the control list (request_count bytes) is not enough for the requested operation. DStatus returns error \$22 if the status list buffer isn't large enough to hold all the data the operation needs to return.

DControl Subcommands

Recall from Chapter 4 that one parameter in the DControl parameter list is control_code, a numeric code describing the type of control operation a device driver is to perform. Other important parameters are control_list, a pointer to a control list buffer

Table 9-2 The Console Driver's video output commands

Command Code	*Command Description*
$00	Null. This command does nothing.
$01	Push and Reset Text Port. This command saves the current text port and then sets the text port to its default state.
$02	Set Text Port Size. This command sets the boundaries of the current text port. It must be followed by four parameters: window left + 32, window top + 32, window right + 32, and window bottom + 32 (in that order). The values of the parameters are relative to the full hardware screen and numbering begins with $00.
$03	Clear from Beginning of Line. This command erases all characters from the left edge of the current line in the text port up to and including the character beneath the cursor.
$04	Pop Text Port. This command restores the text port saved by Push and Reset Text Port (command $01) and makes it the current text port.
$05	Horizontal Scroll. This command shifts the contents of a text port left or right and erases the vacated space. It must be followed by a signed byte describing the direction and extent of the shift: if negative, the shift is to the left; if positive, the shift is to the right. The absolute value of the byte gives the number of columns to shift.
$06	Set Vertical Position. This command sets the vertical position of the cursor within the text port. It must be followed by a byte describing the vertical position + 32.
$07	Ring Bell. This command beeps the speaker.
$08	Backspace. This command moves the cursor one position to the left. If the cursor is already at the left edge of the text port, and cons_wrap is true, it moves to the end of the previous line. If it is at the top left-hand corner of the text port, and cons_scroll is also true, the text port scrolls backward one line.
$09	Null. This command does nothing.
$0A	Line Feed. This command moves the cursor one line down in the text port without affecting the column position. If the cursor is on the last line of the text port, and cons_scroll is true, the text port scrolls up one line.
$0B	Clear to End of Text Port. This command erases the characters from the current cursor position to the end of the text port.

Table 9-2 Continued

Command Code	*Command Description*
$0C	Clear Text Port and Home Cursor. This command erases the entire text port and puts the cursor in the top left-hand corner.
$0D	Carriage Return. This command moves the cursor to the left edge of the current line in the text port. If cons_LF is true, a line feed operation ($0A) automatically follows.
$0E	Set Normal Display Mode. This command forces subsequently outputted characters to be displayed in normal mode.
$0F	Set Inverse Display Mode. This command forces subsequently outputted characters to be displayed in inverse mode.
$10	DLE Space Expansion. This command is for outputting a sequence of space characters very quickly. If cons_DLE is true, this command must be followed by a byte containing the number of space characters + 32 to be displayed. If cons_DLE is false, the next byte is ignored, and no space characters are displayed.
$11	Set 40-Column Mode. This command turns on the 40-column display mode hardware.
$12	Set 80-Column Mode. This command turns on the 80-column display mode hardware.
$13	Clear From Beginning of Text Port. This command erases all characters from the beginning of the text port up to and including the character beneath the cursor.
$14	Set Horizontal Position. This command sets the horizontal position of the cursor within the text port. It must be followed by a byte describing the horizontal position + 32.
$15	Set Cursor Movement. This command sets the cursor movement flags, which are arranged as follows in the byte:

7	6	5	4	3	2	1	0
0	0	0	DLE	Scrl	Wrap	LF	Adv

bit 4	1 =	cons_DLE is true (DLE space expansion can occur)
bit 3	1 =	cons_scroll is true (text port scrolls if the cursor moves up when on the first line or down when on the last line)

Table 9-2 Continued

Command Code	*Command Description*
	bit 2 1 = cons_wrap is true (cursor moves past the end of a line to the next line) bit 1 1 = cons_LF is true (carriage return is followed by a line feed) bit 0 1 = cons_advance is true (cursor moves one space to the right after printing a character) Many video output commands check these flags to determine how to behave when certain cursor movement operations are requested. The default settings for all these flags is true.
$16	Scroll Down One Line. This command scrolls the text port down one line. The cursor stays put.
$17	Scroll Up One Line. This command scrolls the text port up one line. The cursor stays put.
$18	Disable MouseText Mapping. This command turns off the conversion of inverse uppercase characters to MouseText icons.
$19	Home Cursor. This command moves the cursor to the top left-hand corner of the current text port.
$1A	Clear Line. This command erases the line in the text port that the cursor is on.
$1B	Enable MouseText Mapping. This command enables the conversion of inverse uppercase characters to MouseText icons.
$1C	Move Cursor Right. This command moves the cursor one position to the right. If the cursor is on the right edge of the text port, and cons_wrap is true, the cursor moves to the beginning of the next line. If cons_scroll is also true, and the cursor is on the right edge of the last line, the text port scrolls up one line.
$1D	Clear to End of Line. This command erases all characters from the current cursor position to the end of the current line.
$1E	GotoXY. This command positions the cursor within the current text port. It must be followed by bytes describing the horizontal position + 32 and the vertical position + 32.
$1F	Move Cursor Up. This command moves the cursor up one line without affecting the horizontal position. If the cursor is on the top line of the text port, and cons_scroll is true, the text port scrolls up one line.

containing the data the control operation needs, and request_count, the size of the control list buffer.

We describe each of the important control_code operations in the following paragraphs:

Set Wait/No-Wait Mode (code $0004). Use this command to set up wait mode or no-wait mode before commencing a Read operation. Put $0000 in the control list for wait mode or $8000 for no-wait mode. The setting of the wait/no-wait flag is irrelevant as far as UIR operations are concerned because the UIR always operates in wait mode. The request_count is always 2.

Set Input Port (code $8000). Use this command to set the input port to a given state. A copy of the input port record must be in the control list, and the request_count is always 17.

Set Terminator List (code $8001). Use this command to set up the terminator list for the Read command to use. The terminator list must be in the control list; it begins with a mask word and a terminator count word, followed by the terminator words (if any). The request_count must be equal to 4 + 2 × (terminator count).

Restore Text Port Data (code $8002). Use this command to copy the video data in the control list to the current text port. The data in the control list is in the same format used by the Save Text Port Data DStatus command: port width byte, port length byte, followed by the video bytes for each line in the text port. The request_count for a full 80 by 24 screen is 1922 (2 + 80 × 24).

Set Read Mode (code $8003). Use this command to select between raw mode and UIR mode. Put $0000 in the control list for UIR mode or $8000 for raw mode. The request_count is always 2.

Set Default String (code $8004). Use this command to set up the default string to be used by the UIR. Put the string in the control list and the length in the request_count field. If you don't want a default string, set the length to zero. The default string cannot exceed 254 characters.

Abort Input (code $8005). Use this command to cancel an input operation that is in progress. If you don't, the next Read command will pick up from where the last one left off. The request_count is always 0.

DStatus Subcommands

The GS/OS DStatus command uses a status_code parameter describing the nature of the status operation to be performed. A DStatus command returns data in a status_list buffer specified by the application; the number of bytes returned comes back in the transfer_count field. Before using the DStatus command, make sure the size of the status list buffer (the value in the request_count field) is large enough for the expected transfer_count.

We describe each of the important status_code operations in the following paragraphs:

Return Wait/No-Wait Mode (code $0002). This command returns the current Read mode in the status list. Wait mode is active if the result is $0000, and no-wait mode is active if the result is $8000. UIR operations always use wait mode, notwithstanding the status of the wait/no-wait flag, however. The transfer_count is 2.

Return Text Port (code $8000). This command returns a copy of the current text port record in the status list. The transfer_count is 16.

Return Input Port (code $8001). This command returns a copy of the current input port record in the status list. The transfer_count is 17.

Return Terminator List (code $8002). This command returns a copy of the current terminator list in the status list. The terminator list begins with a terminator mask (word) and a terminator count word, followed by the terminator words (if any). The transfer_count is 4 + 2 × (terminator count).

Return Text Port Data (code $8003). This command returns in the status list a copy of the characters that appear in the active text port. The returned data begins with a port width byte and a port length byte and is followed by the screen bytes for each line of the text port. The transfer_count for the full 80 by 24 text screen is 1922 (2 + 80 × 24).

Return Screen Character (code $8004). This command returns in the status list the screen byte for the character beneath the current cursor position. The screen byte is the value actually stored in video RAM to display the character, not the character code (see Table 9-1). The transfer_count is 1.

Return Read Mode (code $8005). This command returns the read mode flag in the status list. The result is $0000 if UIR is active and $8000 if raw mode is active. The transfer_count is 2.

Return Default String (code $8006). This command returns the current default input string in the status list. The maximum transfer_count is 254.

CONSOLE DRIVER PROGRAMMING EXAMPLE

The program in Table 9-3 illustrates many of the programming techniques you will use when working with the Console Driver. It prompts the user to enter a name and uses the UIR to handle the response. The terminator list includes two interrupt keys (-? and -/) that the program responds to by displaying a dummy help screen, asking the user to press Return to continue, and then returning to the initial Read command to get the rest of the name input. The program ends when the user presses Return or Esc (two other terminator characters) while entering a name.

After you assemble the program with the APW assembler, change its file type to S16 (GS/OS system file) before running it. You can do this with the FILETYPE command.

The program first calls GetDevNumber to determine the device number for the .CONSOLE device. It stores this number in the parameter tables for all the DControl and DStatus commands the program uses. The program then calls Open to enable access to the Console Driver and copies the reference number Open returns to the necessary parameter tables.

Next, the program calls DStatus to return two copies of the default input port. The first copy is used in the DoHelp subroutine. The second copy is the one the main Read command uses, but before the program calls Read, it changes the fill_char field of the input port to $C9 (the code for the MouseText underscore symbol). A call to DControl tells the Console Driver about the change.

The last three preliminary steps are to use DControl to set up the default input string (to John Q. Public), set up the terminator list, and enable UIR mode. (Remember, the default is raw mode.)

Four terminator characters are placed in the terminator list: Return, Esc, -?, and -/. Return and Esc are ordinary terminators, whereas -? and -/ are interrupt terminators. (Bit 13 of the terminator modifier word is set to 1.) Notice that the terminator mask (in the DC_Parms1 parameter table) is set to $A0FF so that only the Open-Apple and interrupt modifier bits (and the ASCII character code) will be significant.

The Write command clears the screen, positions the cursor on the middle line, and displays the "Enter your name: " prompt.

The program calls the Read command to begin processing user input. On exit, it calls DStatus to retrieve a copy of the current input port so that the exit_type field can be inspected. If exit_type is 3 or 4, a Help key (-? or -/) was pressed, and control branches to DoHelp. Any other exit must have been caused by the user pressing Return or Esc, so the program calls the Close command and ends.

Since the DoHelp subroutine uses video output and keyboard input commands, it must be sure to preserve the Console Driver's status quo. It uses DStatus to save the data in the text port (the characters on the screen) and uses Write to save the text port record by sending the $01 command just before the help message.

The program saves the state of the input operation in progress by saving a copy of the current input port. It then sets up a default input port before calling the Read command to wait for a Return keypress. (In a more general application, the read mode, terminator list, and default string would be saved too.)

On return, the input and text ports are restored, as are the data in the text port, and the default string. Control then returns to the main Read command so that the user can finishing entering the name.

Table 9-3 This program shows how to use the GS/OS Console Driver

```
*************************************
*     Console Driver Exerciser      *
*                                   *
* Copyright 1988 Gary B. Little     *
*                                   *
* Last modified: September 4, 1988  *
*                                   *
*************************************

           KEEP    CONSOLE          ;Object code file
           MCOPY   CONSOLE.MAC      ;Macro file

Console    START

           PHK
           PLB

           _GetDevNumber GDN_Parms
           LDA     dev_num
           STA     dev_num1
           STA     dev_num2
           STA     dev_num3
           STA     dev_num4
           STA     dev_num5
           STA     dev_num6
           STA     dev_num7

           _Open   Open_Parms
           LDA     ref_num
           STA     ref_num1
           STA     ref_num2
           STA     ref_num3
           STA     ref_num4
           STA     ref_num5
           STA     ref_num6

           LDA     #$8001
           STA     IP_Cmd
           _DStatus IP_Parms        ;Get copy of std input port

           LDA     #$8001
           STA     IP_Cmd1
           _DStatus IP_Parms1       ;Get input port

* Make changes to the default port:

           SEP     #$20
```

Table 9-3 Continued

```
           LONGA   OFF

           LDA     #$C9             ;New Fill_Char
           STA     Input_Rec1+0

           REP     #$20
           LONGA   ON

           LDA     #$8000
           STA     IP_Cmd1
           _DControl IP_Parms1      ;Set new input port

           _DControl DS_Parms       ;Set default string
           _DControl DC_Parms1      ;Set terminator list
           _DControl DC_Parms2      ;Set UIR mode

           _Write   Wr_Parms1       ;Set up prompt

GetInput   _Read    Read_Parms

           LDA     #$8001
           STA     IP_Cmd1
           _DStatus IP_Parms1       ;Get input port

           LDA     Input_Rec1+5     ;Exit_type
           AND     #$00FF

           CMP     #3               ;Terminator #3?
           BEQ     DoHelp
           CMP     #4               ;Terminator #4?
           BEQ     DoHelp

           _Close  Close_Parms

           _Quit   Quit_Parms
           BRK     $00

* Here is where we display a help screen and wait for
* any key to continue. We must preserve the text port,
* the text port data, and the input record.

DoHelp     ANOP

           LDA     #$8003
           STA     TP_Cmd
           _DStatus TP_Parms        ;Save text port data

           _Write  Wr_Parms2        ;Push port, display help screen
```

Table 9-3 Continued

```
            LDA     #$8001
            STA     IP_Cmd1
            _DStatus IP_Parms1      ;Get current input port

            LDA     #$8000
            STA     IP_Cmd
            _DControl IP_Parms      ;Set up a virgin input port

            _DControl DS_Parms1     ;Set default string to null

            _Read   OneByte

            LDA     #$8000
            STA     IP_Cmd1
            _DControl IP_Parms1     ;Restore input port

            _DControl DS_Parms      ;Restore default string

            _Write  Wr_Parms3       ;Pop text port

            LDA     #$8002
            STA     TP_Cmd
            _DControl TP_Parms      ;Restore text port data

            JMP     GetInput        ;Get rest of input

Quit_Parms  ANOP
            DC      I2'2'
            DC      I4'0'
            DC      I2'0'

Open_Parms  ANOP
            DC      I2'2'
ref_num     DS      2               ;Reference number returned here
            DC      I4'Cons_Name'

Cons_Name   DC      I2'8'           ;Length word
            DC      C'.CONSOLE'

Read_Parms  ANOP
            DC      I2'4'
ref_num2    DS      2
            DC      I4'Buffer'
            DC      I4'30'          ;Request count
rd_count    DS      4               ;Actual count

Buffer      DS      30
```

Table 9-3 Continued

```
* Parameter table for reading one character:

OneByte    ANOP
           DC      I2'4'
ref_num5   DS      2
           DC      I4'TheChar'
           DC      I4'1'            ;Request count
           DS      4

TheChar    DS      1

Wr_Parms1  ANOP
           DC      I2'4'
ref_num3   DS      2
           DC      I4'Scr_Init'
           DC      I4'Msg_Len-Scr_Init'
           DS      4

Scr_Init   ANOP
           DC      I1'$0C'          ;Clear screen
           DC      I1'$1E,$20,$2C'  ;Move to row 12, column 0
           DC      C'Enter your name: '
Msg_Len    ANOP

Wr_Parms2  ANOP
           DC      I2'4'
ref_num4   DS      2
           DC      I4'Scr_Help'
           DC      I4'Msg_Len1-Scr_Help'
           DS      4

Scr_Help   ANOP
           DC      I1'$01'          ;Push and reset text port
           DC      I1'$0C'          ;Clear screen
           DC      I1'$1E,$2A,$2C'  ;Move to row 12, column 10
           DC      C'This is a help screen!'
           DC      I1'$1E,$20,$37'  ;Move to row 23, column 0
           DC      C'Press Return to continue: '
Msg_Len1   ANOP


Wr_Parms3  ANOP
           DC      I2'4'
ref_num6   DS      2
           DC      I4'Pop_TP'
           DC      I4'1'
           DS      4
Pop_TP     DC      I1'$04'          ;Pop text port
```

Table 9-3 Continued

```
Close_Parms ANOP
           DC     I2'1'
ref_num1   DS      2

GDN_Parms  ANOP
           DC      I2'2'
           DC      I4'Cons_Name'
dev_num    DS      2

* Parameter table for setting the
* default input string:

DS_Parms   ANOP
           DC      I2'5'
dev_num1   DS      2
           DC      I2'$8004'
           DC      I4'Def_Name'
           DC      I4'Size-Def_Name' ;Length of list
           DS      4

Def_Name   DC      C'John Q. Public'
Size       ANOP

DS_Parms1  ANOP
           DC      I2'5'
dev_num7   DS      2
           DC      I2'$8004'
           DC      I4'Def_Name'
           DC      I4'0'             ;No default
           DS      4

DC_Parms1  ANOP
           DC      I2'5'
dev_num2   DS      2
           DC      I2'$8001'         ;Set terminator list
           DC      I4'TermList'
           DC      I4'Size1-TermList' ;Length of list
           DS      4

TermList   DC      I2'$A0FF'         ;Terminator mask
           DC      I2'4'             ;Count
           DC      I2'$000D'         ;Return
           DC      I2'$001B'         ;Esc
           DC      I2'$A03F'         ;OA-? (interrupt)
           DC      I2'$A02F'         ;OA-/ (interrupt)
Size1      ANOP

DC_Parms2  ANOP
```

Table 9-3 Continued

```
          DC      I2'5'
dev_num3  DS      2
          DC      I2'$8003'        ;Set read mode
          DC      I4'RM_List'
          DC      I4'2'            ;Length of list
          DS      4

RM_List   DC      I2'$0000'        ;UIR


IP_Parms  ANOP
          DC      I2'5'
dev_num5  DS      2

* IP_Cmd = $8001 (return input port) for DStatus
* IP_Cmd = $8000 (set input port) for DControl

IP_Cmd    DS      2                ;Return/set input port
          DC      I4'Input_Rec'
          DC      I4'IPR_Size-Input_Rec'
          DS      4

Input_Rec DS      17               ;Space for input port record
IPR_Size  ANOP


IP_Parms1 ANOP
          DC      I2'5'
dev_num4  DS      2

* IP_Cmd = $8001 (return input port) for DStatus
* IP_Cmd = $8000 (set input port) for DControl

IP_Cmd1   DS      2                ;Return/set input port
          DC      I4'Input_Rec1'
          DC      I4'IPR_Size1-Input_Rec1'
          DS      4

Input_Rec1 DS     17               ;Space for input port record
IPR_Size1 ANOP

*******************************************
* Parameter table for saving and restoring *
* the data in the text port.               *
*******************************************
TP_Parms  ANOP
          DC      I2'5'
dev_num6  DS      2
```

Table 9-3 Continued

```
TP_Cmd      DS      2
            DC      I4'TextPort'
            DC      I4'TP_Len-TextPort'
            DS      4

TextPort    DS      80*24+2
TP_Len      ANOP

            END
```

APPENDIX I

Using Assemblers

Two assemblers were used to create the assembly-language example programs in this book. Merlin 8/16 (Roger Wagner Publishing, 1050 Pioneer Way, Suite P, El Cajon, CA 92020, 619/442-0522) was used for the ProDOS 8 programs, and the Apple Programmer's Workshop (APW) assembler (APDA, Mail Stop 33-G, 20525 Mariani Avenue, Cupertino, CA 95014, 800/282-2732) was used for the GS/OS programs.

The reason for using two different assemblers is primarily historical. Merlin 8/16 (previously called Merlin Pro) is probably the most popular assembler available for creating ProDOS 8 applications primarily because it was introduced soon after Apple first released ProDOS 8. Similarly, the APW assembler is the most popular assembler for creating GS/OS applications because it was the only 65816 assembler available when the Apple IIGS came out, and its linker can create GS/OS load files. Even though the current version of Merlin 8/16 now has a linker for creating GS/OS load files, most programmers are more familiar with the APW assembler, so that's the one used for the GS/OS examples.

If you want to modify and reassemble the example programs, and you are not using the same assembler, you may have to make changes to the source code to resolve any differences in syntax and command structure. Differences usually arise in the area of *pseudo-instructions*; these are commands to the assembler that appear in the instruction field of a line of source code. They can be used to place data bytes at specific locations within the program, to define symbolic labels, to indicate the starting address of the program, and for several other purposes.

MERLIN 8/16

Here are the meanings of some of Merlin 8/16'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).
ADRL $E100A8	Stores the 65816 long address $E100A8 in the object code as $A8 $00 $E1 $00 (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.
STR 'string'	Stores the ASCII codes for the string, preceded by a length byte.

The operand formats for most ProDOS 8 assemblers like Merlin 8/16 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- or low-order byte of a 2-byte address is identified as an immediate quantity. With Merlin 8/16, 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. One assembler, Apple's 6502 Editor/Assembler, uses the same general method, but it reverses the meaning: # > identifies the low-order byte, and # < identifies the high-order byte! Be careful.

APW ASSEMBLER

Here are the meanings of some of the APW assembler's most important pseudo-opcodes:

DC I1'$03'	Stores the byte $03 in the object code.
DS 16	Reserves a data space of 16 bytes (to no particular value).
DC I2'$FDED'	Stores the address $FDED in the object code as $ED $FD (that is, low-order byte first).
DC I4'$E100A8'	Stores the 65816 long address $E100A8 in the object code as $A8 $00 $E1 $00 (that is, low-order byte first).

DC C'ABCD'	Stores the ASCII codes for ABCD in the object code. By default, the codes are stored with the high-order bit off; you can use the MSB ON directive if you want them stored with the high-order bit on.
COUT GEQU $FDED	Equates the symbolic label COUT with the address $FDED.

The APW assembler permits you to create macros—assembler directives that expand into a series of 65816 instructions. APW comes with a standard set of macros for all GS/OS commands and IIGS tool set functions. The macro name is the same as the command or tool set name except that it begins with an underscore character (_). The GS/OS macros require one parameter, the address of the parameter table for the GS/OS command.

By using these standard macros, you don't have to memorize GS/OS command numbers or tool set function numbers. It also makes your source code easier to understand.

Here are five other macros some of the examples use:

STR	Stores an ASCII string preceded by a length byte.
STR1	Stores an ASCII string preceded by a length word.
PushPtr	Pushes the address of a data area on the stack.
PushWord	Pushes a word on the stack.
PushLong	Pushes a long word on the stack.

To use a macro, put its name in the assembler's instruction field. If the macro has a parameter, put it in the operand field. When the source code file is assembled, the 65816 instructions that the macro defines are placed in the object code.

APPENDIX II

ProDOS Blocks and DOS 3.3 Sectors

The ProDOS 8 READ_BLOCK and WRITE_BLOCK commands discussed in Chapter 4 can be used to access directly any sector on any track of a DOS 3.3-formatted disk. This makes it easier 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. (See Chapter 5 of *Inside the Apple IIe* for this information.)

To use READ_BLOCK and WRITE_BLOCK with DOS 3.3 disks, first 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 and 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	0 and 14
1	13 and 12
2	11 and 10
3	9 and 8
4	7 and 6
5	5 and 4
6	3 and 2
7	1 and 15

For example, track 17, sector 15 on a DOS 3.3 diskette corresponds to block number 143 ($8 \times 17 + 7$).

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 corresponds 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, you must first 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 are not disturbed.

APPENDIX III

Bibliography

GS/OS AND PRODOS 8 REFERENCE BOOKS

Apple Computer, Inc., *GS/OS Reference, Volume 1* (Apple Programmer's and Developer's Association, 1988). This manual gives a programmer's overview of GS/OS, describes the GS/OS commands, and discusses specific file system translators.

Apple Computer, Inc., *GS/OS Reference, Volume 2* (Apple Programmer's and Developer's Association, 1988). This manual describes low-level GS/OS entities, like device drivers and interrupt handlers.

Apple Computer, Inc., *Apple IIGS ProDOS 16 Reference* (Addison-Wesley, 1987). This is the official reference manual for ProDOS 16.

Apple Computer, Inc., *ProDOS 8 Technical Reference Manual* (Addison-Wesley, 1987). This is the official reference manual for ProDOS 8.

Apple Computer, Inc., *BASIC Programming with ProDOS* (Addison-Wesley, 1987). This book describes how to use the BASIC.SYSTEM commands.

APPLE II REFERENCE BOOKS

Apple II Reference Manual (Apple Computer, Inc., 1979). The official reference manual for the Apple II and Apple II Plus.

Apple Computer, Inc., *Apple IIe Technical Reference Manual* (Addison-Wesley, 1987). The official reference manual for the Apple IIe.

Apple Computer, Inc., *Apple IIc Technical Reference Manual* (Addison-Wesley, 1987). The official reference manual for the Apple IIc.

Apple Computer, Inc., *Apple IIGS Toolbox Reference: Volume 1* (Addison-Wesley, 1988). This book describes the Apple IIGS tool set functions.

Apple Computer, Inc., *Apple IIGS Toolbox Reference: Volume 2* (Addison-Wesley, 1988). This book describes the Apple IIGS tool set functions.

Apple Computer, Inc., *Apple II SCSI Card Technical Reference* (Apple Programmer's and Developer's Association, 1988). This book describes the SmartPort calls for the Apple II SCSI interface card.

Gary B. Little, *Inside the Apple IIc* (Brady/Prentice Hall Press, 1985). This book is a programmer's guide to the Apple IIc.

Gary B. Little, *Inside the Apple IIe* (Brady/Prentice Hall Press, 1985). This book is a programmer's guide to the Apple IIe.

Gary B. Little, *Exploring the Apple IIGS* (Addison-Wesley, 1987). This book is a programmer's guide to the Apple IIGS.

65816 ASSEMBLY-LANGUAGE BOOKS

David Eyes and Ron Lichty, *Programming the 65816* (Brady/Prentice Hall Press, 1986). This book is a programmer's guide to the 65816 microprocessor.

APPENDIX IV

The Program Disk

A disk containing the source code for each of the programs described in this book, as well as four bonus programs, can be ordered directly from Gary Little. See the last page of this book for ordering information.

The files on the disk are one of five types:

- TXT (text) files having names of the form xxxxxxxxxxx.S. These files contain assembly-language source code in the format expected by the Merlin 8/16 assembler.
- $B0 (source) files. These files contain assembly-language source code in the format expected by the APW assembler.
- BAS (BASIC) files. These files contain Applesoft programs that you can run using the BASIC.SYSTEM RUN or - command.
- BIN (binary) files. These files contain assembly-language programs you can run using the BASIC.SYSTEM BRUN or - command. A BIN file is created from its corresponding source code file by assembling the source with Merlin 8/16 and saving the object code to disk.
- SYS (system) files. These files contain assembly-language programs you can run by using the BASIC.SYSTEM - command or by specifying the file's pathname in a program selector utility.

The program disk 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 8 master disk using the ProDOS 8 Filer or System Utilities program.

The names of the programs on the disk are the same as those used in this book.

Here are descriptions of the four bonus programs (source code is included on the program disk):

THE DISK.MAP PROGRAM

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 5.25-inch disk. To run the program, enter the command

```
-DISK.MAP
```

from Applesoft command mode. After you do this, you will be asked for the slot number of the drive in which the disk has been placed. (If you have two drives for a slot, put the disk in the drive 1.) DISK.MAP maps each block on the disk to a unique position in an 8 by 35 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 graphic 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 graphic block is gray, the disk block is not being used.

DISK.MAP also displays the amount of free space on the disk and the name of the volume directory.

THE PROTIME PROGRAM

When you execute PROTIME (with the - command), the TIME command is added to the BASIC.SYSTEM command set. When you enter the TIME command from Applesoft command 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 November 30, 1988 and the time is 9:20 p.m., you will see

```
30-NOV-1988  21:20
```

As you see, 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 you execute the program line

```
100  PRINT  CHR$(4);"TIME": INPUT TM$
```

the time string is assigned to the TM$ variable. The Applesoft string parsing commands can then be used to isolate elements of the string your program may need to examine.

THE PROTYPE PROGRAM

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

```
-PROTYPE
```

from Applesoft command 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 =	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 (nonzero means page pause)
,T# =	title code (nonzero means 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). Moreover, a page number will appear on each page (,T1), and there will be a left margin of five spaces (,E5).

You can temporarily halt all output generated by the TYPE command by entering [Control-S] from the keyboard. To resume, press [Control-S] once again. You can press [Control-C] at any time to cancel the command.

THE SMARTPORT PROGRAM

SMARTPORT is for determining which slots in the Apple II have SmartPort controllers connected to them. It displays status information for the devices connected to each SmartPort it finds. In particular, it displays the device name, the slot number and unit number, the device type and subtype, the version number, the device status, and the total number of blocks the device supports. This last number is either a 4-byte quantity or a 3-byte quantity depending on whether the SmartPort supports extended commands. (See Chapter 7 for a thorough discussion of the characteristics of a SmartPort.)

To run SMARTPORT, enter the command

```
-SMARTPORT
```

from Applesoft command mode. (SMARTPORT is a system program, so you could also run it from any program selector.) When it starts up, you can specify whether or not you want to send the results of the scan to a printer in slot 1.

INDEX

PROGRAM DISK FOR EXPLORING APPLE GS/OS AND PRODOS 8 BY GARY B. LITTLE

All the programs listed in this book are available on disk, in source code form, directly from the author. The disk also contains several other useful programs, all described in Appendix IV of this book.

To order the disk, simply clip or photocopy this entire page and complete the coupon below. Enclose a check or money order for $15.00 in U.S. funds. (California residents add applicable state sales tax.)

Mail to: Gary B. Little
3304 Plateau Drive
Belmont, CA
94002

– –

Please send me a copy of the *Exploring Apple GS/OS and ProDOS 8* disk.

Specify disk format: 3½" _______ or 5¼" _______

I am enclosing a check or money order in the amount of $15 in U.S. funds, plus applicable California state tax.

Name: ____

Address: ____

City: ______________________ State/Province: ________ Zip: ________

Country: ______________________

EXPLORING APPLE GS/OS AND ProDOS 8

REFERENCE CARD

GARY B. LITTLE

Command Calling Sequence

GS/OS:

```
JSL $E100A8        ;call command interpreter
DC I2'CmdNum'      ;command number
DC I4'ParmTbl'     ;address of parm table
BCS Error          ;carry set if error occurs
```

ProDOS 8:

```
JSR $BF00          ;call command interpreter
DFB CmdNum         ;command number
DA ParmTbl         ;address of parm table
BCS Error          ;carry set if error occurs
```

If an error occurs, GS/OS or ProDOS 8 sets the carry flag, clears the zero flag, and puts an error code in the accumulator.

GS/OS and ProDOS 8 Error Codes

$00 No error occurred
$01 Invalid command
$04 Invalid parameter count
$07 GS/OS is busy
$10 Device not found
$11 Invalid device number
$22 Bad GS/OS driver parameter.
$23 GS/OS Console Driver is not open.
$25 Interrupt vector table full
$27 Disk I/O error
$28 No disk device connected
$2B Disk is write-protected
$2E Disk volume was switched
$2F No disk in drive
$40 Invalid pathname syntax
$42 No more file buffers allowed
$43 File not open (invalid ref_num)
$44 Directory does not exist
$45 Volume directory does not exist
$46 File does not exist
$47 Duplicate pathname
$48 Disk is full
$49 Volume directory is full
$4A Incompatible version
$4B Unsupported storage type
$4C End of data
$4D Range error
$4E Access code forbids operation
$4F Class 1 output space too small
$50 File is open
$51 Damaged directory
$52 Unsupported file system
$53 Parameter out of range
$54 Out of memory
$55 VCB table is full
$56 Buffer area is in use
$57 Volumes have same name
$58 Not a block device
$59 Invalid file level
$5A Volume bit map is damaged
$5B Illegal pathname change
$5C File is not executable
$5D Operating system not supported
$5E /RAM cannot be removed
$5F Quit Return Stack overflow
$60 Can't report last device
$61 End of directory
$62 Invalid class number
$64 Invalid file system ID code
$65 Invalid FST operation

ProDOS 8 MLI Commands

(R = result, I = input)

ALLOC_INTERRUPT ($40)

+0	num_parms (2)	I
+1	int_num	R
+2 to +3	int_code	I

CLOSE ($CC)

+0	num_parms (1)	I
+1	ref_num	I

CREATE ($C0)

+0	num_parms (7)	I
+1 to +2	pathname	I
+3	access	I
+4	file_type	I
+5 to +6	aux_type	I
+7	storage_type	I
+8 to +9	create_date	I
+10 to +11	create_time	I

DEALLOC_INTERRUPT ($41)

+0	num_parms (1)	I
+1	int_num	I

DESTROY ($C1)

+0	num_parms (1)	I
+1 to +2	pathname	I

FLUSH ($CD)

+0	num_parms (1)	I
+1	ref_num	I

GET_BUF ($D3)

+0	num_parms (2)	I
+1	ref_num	I
+2 to +3	io_buffer	R

GET_EOF ($D1)

+0	num_parms (2)	I
+1	ref_num	I
+2 to +4	eof	R

GET_FILE_INFO ($C4)

+0	num_parms (10)	I
+1 to +2	pathname	I
+3	access	R
+4	file_type	R
+5 to +6	aux_type[a]	R
+7	storage_type	R
+8 to +9	blocks_used[a]	R
+10 to +11	mod_date	R
+12 to +13	mod_time	R
+14 to +15	create_date	R
+16 to +17	create_time	R

[a] For a volume directory file, aux_type contains the capacity of the volume in blocks and blocks_used returns the number of blocks in use by all files on the volume.

GET_MARK ($CF)

+0	num_parms (2)	I
+1	ref_num	I
+2 to +4	position	R

GET_PREFIX ($C7)

+0	num_parms (1)	I
+1 to +2	prefix	R

GET_TIME ($82)

– no parameter table –

NEWLINE ($C9)

+0	num_parms (3)	I
+1	ref_num	I
+2	enable_mask	I
+3	newline_char	I

ON_LINE ($C5)

+0	num_parms (2)	I
+1	unit_num	I
+2 to +3	data_buffer	R

OPEN ($C8)

+0	num_parms (3)	I
+1 to +2	pathname	I
+3 to +4	io_buffer	I
+5	ref_num	R

QUIT ($65)

+0	num_parms (4)	I
+1	quit_type	I
+2 to +3	pathname	I
+4	[reserved]	I
+5 to +6	[reserved]	I

READ ($CA)

+0	num_parms (4)	I
+1	ref_num	I
+2 to +3	data_buffer	I
+4 to +5	request_count	I
+6 to +7	transfer_count	R

READ_BLOCK ($80)

+0	num_parms (3)	I
+1	unit_num	I
+2 to +3	data_buffer	R
+4 to +5	block_num	I

RENAME ($C2)

+0	num_parms (2)	I
+1 to +2	curr_name	I
+3 to +4	new_name	I

SET_BUF ($D2)

+0	num_parms (2)	I
+1	ref_num	I
+2 to +3	io_buffer	I

SET_EOF ($D0)

+0	num_parms (2)	I
+1	ref_num	I
+2 to +4	eof	I

SET_FILE_INFO ($C3)

+0	num_parms (7)	I
+1 to +2	pathname	I
+3	access	I
+4	file_type	I
+5 to +6	aux_type	I
+7	[not used]	I
+8 to +9	[not used]	I
+10 to +11	mod_date	I
+12 to +13	mod_time	I

SET_MARK ($CE)

+0	num_parms (2)	I
+1	ref_num	I
+2 to +4	position	I

SET_PREFIX ($C6)

+0	num_parms (1)	I
+1 to +2	prefix	I

WRITE ($CB)

+0	num_parms (4)	I
+1	ref_num	I
+2 to +3	data_buffer	I
+4 to +5	request_count	I
+5 to +6	transfer_count	R

WRITE_BLOCK ($81)

+0	num_parms (3)	I
+1	unit_num	I
+2 to +3	data_buffer	I
+4 to +5	block_num	I

GS/OS MLI Commands

(R = result, I = input)

BeginSession ($201D)

+0 to +1	pcount (0)	I

BindInt ($2031)

+0 to +1	pcount (3)	I
+2 to +3	int_num	R
+4 to +5	vrn	I
+6 to +9	int_code	I

ChangePath ($2004)

+0 to +1	pcount (2)	I
+2 to +5	pathname	I
+6 to +9	new_pathname	I

ClearBackup ($200B)

+0 to +1	pcount (1)	I
+2 to +5	pathname	I

Close ($2014)

+0 to +1	pcount (1)	I
+2 to +3	ref_num	I

Create ($2001)

+0 to +1	pcount (7)	I
+2 to +5	pathname	I
+6 to +7	access	I
+8 to +9	file_type	I
+10 to +13	aux_type	I
+14 to +15	storage_type	I
+16 to +19	eof	I
+20 to +23	resource_eof	I

DControl ($202E)

+0 to +1	pcount (5)	I
+2 to +3	dev_num	I
+4 to +5	control_code	I
+6 to +9	control_list	I
+10 to +13	request_count	I
+14 to +17	transfer_count	R

Destroy ($2002)

+0 to +1	pcount (1)	I
+2 to +5	pathname	I

DInfo ($202C)

+0 to +1	pcount (10)	I
+2 to +3	dev_num	I
+4 to +7	dev_name	R
+8 to +9	characteristics	R
+10 to +13	total_blocks	R
+14 to +15	slot_num	R
+16 to +17	unit_num	R
+18 to +19	version	R
+20 to +21	device_ID_num	R
+22 to +23	head_link	R
+24 to +25	forward_link	R

DRead ($202F)

+0 to +1	pcount (6)	I
+2 to +3	dev_num	I
+4 to +7	buffer	R
+8 to +11	request_count	I
+12 to +15	starting_block	I
+16 to +17	block_size	I
+18 to +21	transfer_count	R

DStatus ($202D)

+0 to +1	pcount (5)	I
+2 to +3	dev_num	I
+4 to +5	status_code	I
+6 to +9	status_list	R
+10 to +13	request_count	I
+14 to +17	transfer_count	R

DWrite ($2030)

+0 to +1	pcount (6)	I
+2 to +3	dev_num	I
+4 to +7	buffer	I
+8 to +11	request_count	I
+12 to +15	starting_block	I
+16 to +17	block_size	I
+18 to +21	transfer_count	R

EndSession ($201E)

+0 to +1	pcount (0)	I

EraseDisk ($2025)

+0 to +1	pcount (4)	I
+2 to +5	dev_name	I
+6 to +9	vol_name	I
+10 to +11	file_sys_id	R
+12 to +13	requested_fsys	I

ExpandPath ($200E)

+0 to +1	pcount (3)	I
+2 to +5	input_path	I
+6 to +9	output_path	R
+10 to +11	flags	I

Flush ($2015)

+0 to +1	pcount (1)	I
+2 to +3	ref_num	I

Format ($2024)

+0 to +1	pcount (4)	I
+2 to +5	dev_name	I
+6 to +9	vol_name	I
+10 to +11	file_sys_id	R
+12 to +13	requested_fsys	I

FSTSpecific ($2033)

+0 to +1	pcount (3)	I
+2 to +3	file_sys_id	I
+4 to +5	command_num	I
+6 to +7/9	command_parm	I/R

GetBootVol ($2028)

+0 to +1	pcount (1)	I
+2 to +5	data_buffer	R

GetDevNumber ($2020)

+0 to +1	pcount (2)	I
+2 to +5	dev_name	I
+6 to +7	dev_num	R

GetDirEntry ($201C)

+0 to +1	pcount (17)	I
+2 to +3	ref_num	I
+4 to +5	flags	I
+6 to +7	base	I
+8 to +9	displacement	I
+10 to +13	name	R
+14 to +15	entry_num	R
+16 to +17	file_type	R
+18 to +21	eof	R
+22 to +25	block_count	R
+26 to +33	create_td	R
+34 to +41	modify_td	R
+42 to +43	access	R
+44 to +47	aux_type	R
+48 to +49	file_sys_id	R
+50 to +53	option_list	R
+54 to +57	res_eof	R
+58 to +61	res_block_count	R

GetEOF ($2019)

+0 to +1	pcount (2)	I
+2 to +3	ref_num	I
+4 to +7	eof	R

GetFileInfo ($2006)

+0 to +1	pcount (12)	I
+2 to +5	pathname	I
+6 to +7	access	R
+8 to +9	file_type	R
+10 to +13	aux_type	R
+14 to +15	storage_type	R
+16 to +23	create_td	R
+24 to +31	modify_td	R
+32 to +35	option_list	R
+36 to +39	eof	R
+40 to +43	blocks_used	R
+44 to +47	resource_eof	R
+48 to +51	resource_blocks	R

GetFSTInfo ($202B)

+0 to +1	pcount (8)	I
+2 to +3	FST_num	I
+4 to +5	file_sys_id	R
+6 to +9	FST_name	R
+10 to +11	version	R
+12 to +13	attributes	R
+14 to +15	block_size	R
+16 to +19	max_vol_size	R
+20 to +23	max_file_size	R

GetLevel ($201B)

+0 to +1	pcount (1)	I
+2 to +3	level	R

GetMark ($2017)

+0 to +1	pcount (2)	I
+2 to +3	ref_num	I
+4 to +7	position	R

GetName ($2027)

+0 to +1	pcount (1)	I
+2 to +5	data_buffer	R

GetPrefix ($200A)

+0 to +1	pcount (2)	I
+2 to +3	prefix_num	I
+4 to +7	prefix	R

GetSysPrefs ($200F)

+0 to +1	pcount (1)	I
+2 to +3	preferences	R

GetVersion ($202A)

+0 to +1	pcount (1)	I
+2 to +3	version	R

NewLine ($2011)

+0 to +1	pcount (4)	I
+2 to +3	ref_num	I
+4 to +5	enable_mask	I
+6 to +7	num_chars	I
+8 to +11	newline_table	I

Null ($200D)

+0 to +1	pcount (0)	I

Open ($2010)

+0 to +1	pcount (15)	I
+2 to +3	ref_num	R
+4 to +7	pathname	I
+8 to +9	request_access	I
+10 to +11	resource_num	I
+12 to +13	access	R
+14 to +15	file_type	R
+16 to +19	aux_type	R
+20 to +21	storage_type	R
+22 to +29	create_td	R
+30 to +37	modify_td	R
+38 to +41	option_list	R
+42 to +45	eof	R
+46 to +49	blocks_used	R
+50 to +53	resource_eof	R
+54 to +57	resource_blocks	R

OSShutdown ($2003)

+0 to +1	pcount (1)	I
+2 to +3	shutdown_flag	I

Quit ($2029)

+0 to +1	pcount (2)	I
+2 to +5	pathname	I
+6 to +7	flags	I

Read ($2012)

+0 to +1	pcount (5)	I
+2 to +3	ref_num	I
+4 to +7	data_buffer	I
+8 to +11	request_count	I
+12 to +15	transfer_count	R
+16 to +17	cache_priority	I

ResetCache ($2026)

+0 to +1	pcount (0)	I

SessionStatus ($201F)

+0 to +1	pcount (1)	I
+2 to +3	status	R

SetEOF ($2018)

+0 to +1	pcount (3)	I
+2 to +3	ref_num	I
+4 to +5	base	I
+6 to +9	displacement	I

SetFileInfo ($2005)

+0 to +1	pcount (12)	I
+2 to +5	pathname	I
+6 to +7	access	I
+8 to +9	file_type	I
+10 to +13	aux_type	I
+14 to +15	[not used]	I
+16 to +23	create_td	I
+24 to +31	modify_td	I
+32 to +35	option_list	I
+36 to +39	[not used]	I
+40 to +43	[not used]	I
+44 to +47	[not used]	I
+48 to +51	[not used]	I

SetLevel ($201A)

+0 to +1	pcount (1)	I
+2 to +3	level	I

SetMark ($2016)

+0 to +1	pcount (3)	I
+2 to +3	ref_num	I
+4 to +5	base	I
+6 to +9	displacement	I

SetPrefix ($2009)

+0 to +1	pcount (2)	I
+2 to +3	prefix_num	I
+4 to +7	prefix	I

SetSysPrefs ($200C)

+0 to +1	pcount (1)	I
+2 to +3	preferences	I

UnbindInt ($2032)

+0 to +1	pcount (1)	I
+2 to +3	int_num	I

Volume ($2008)

+0 to +1	pcount (6)	I
+2 to +5	dev_name	I
+6 to +9	vol_name	R
+10 to +13	total_blocks	R
+14 to +17	free_blocks	R
+18 to +19	file_sys_id	R
+20 to +21	block_size	R

Write ($2013)

+0 to +1	pcount (5)	I
+2 to +3	ref_num	I
+4 to +7	data_buffer	I
+8 to +11	request_count	I
+12 to +15	transfer_count	R
+16 to +17	cache_priority	I

Unit Number Code

DSSS0000
where D = 0 (drive 1)
= 1 (drive 2)
SSS = 001 to 111 (slot)

Storage Type Code

$00	inactive (deleted) file
$01	seedling file
$02	sapling file
$03	tree file
$04	Pascal region on partitioned d
$05	extended file
$0D	directory file
$0E	subdirectory header
$0F	volume directory header

File Type Codes

$00	UNK
$01	BAD
$04	TXT
$06	BIN
$08	FOT
$0B	WPF
$0F	DIR
$19	ADB
$1A	AWP
$1B	ASP
$AB	GSB
$AC	TDF
$AD	BDF
$B0	SRC
$B1	OBJ
$B2	LIB
$B3	S16
$B4	RTL
$B5	EXE
$B6	STR
$B7	PTI
$B8	NDA
$B9	CDA
$BA	TOL
$BB	DVR
$BC	IDF
$C0	PNT
$C1	PIC
$C8	FON
$C9	FND
$CA	ICN
$CB	AIF
$EF	PAS
$F0	CMD
$F1–$F8	[user-definable]
$F9	O.S
$FA	INT
$FB	IVR
$FC	BAS
$FD	VAR
$FE	REL
$FF	SYS

Auxiliary Type Codes

SRC	APW language ID number
TXT	record length
BAS BIN SYS VAR	address from which the file was saved

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	1 = invisible file
bit 1	1 = write-enabled
bit 0	1 = read-enabled

ProDOS 8 Date Format

MMMDDDDD (low byte)
YYYYYYYM (high byte)

M = month (1..12)
D = day (1..31)
Y = year (0..99)

ProDOS 8 Time Format

00MMMMMM (low byte)
000HHHHH (high byte)

M = minutes (0..59)
H = hours (0..23)

Important GS/OS Locations

$E100A8–$E100AB	Inline command interpreter entry point
$E100B0–$E100B3	Stack-based command interpreter entry point
$E100BC	OS_KIND byte. ProDOS 8 = $00 GS/OS = $01
$E100BD	OS_BOOT byte. ProDOS 8 = $00 GS/OS = $01
$E100BE–$E100BF	GS/OS status flag word. bit 15 = 1 if GS/OS is busy.

Important ProDOS 8 Global Page Locations

$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 GET_TIME)
$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 (IIe, IIc, IIGS 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 (minus 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	System file level
$BF98	MACHID	Machine identification byte:

```
00xx0xxx = Apple II
01xx0xxx = Apple II Plus
10xx0xxx = Apple IIe or IIGS
11xx0xxx = Apple III emulation mode
10xx1xxx = Apple IIc
xx01xxxx = 48K
xx10xxxx = 64K
xx11xxxx = 128K (IIe, IIc, IIGS 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 require0d (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 command 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[b]
$BEB4	SET_FILE_INFO parameter list[b]
$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

[b]The proper parameter count must be set before using this parameter list.

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

Volume Directory and Subdirectory Header

+0	previous directory block		
+2	next directory block		
+4	storage type	name length	
+5	directory name (15 bytes)		
+20	[reserved (8 bytes)]		
+28	date of creation		
+30	time of creation		
+32	version		
+33	minimum version		
+34	access code		
+35	entry length		
+36	entries per block		
+37	number of files		
+39	location of volume bit map		volume directory header only
+41	total blocks on volume		
+39	location of parent directory		subdirectory header only
+41	entry number	entry length	

General File Entry

+0	storage type	name length
+1	filename (15 bytes)	
+16	file type code	
+17	location of key block	
+19	number of blocks used by file	
+21	size of file (EOF)	
+24	date of creation	
+26	time of creation	
+28	version	
+29	minimum version	
+30	access code	
+31	auxiliary type code	
+33	date of modification	
+35	time of modification	
+37	location of start of directory	

Apple Programming

>$21.95 FPT USA

Addison-Wesley Publishing Company

Exploring Apple® GS/OS™ and ProDOS® 8

GARY B. LITTLE

GS/OS™ is the new versatile and powerful operating system for the Apple® IIGS. Built for speed, this true 16-bit operating system takes full advantage of the machine's 65816 microprocessor. The fast and efficient GS/OS replaces ProDOS® 16 as the standard operating system for the Apple IIGS. ProDOS 8 is the original 8-bit operating system for the Apple IIe, IIc, and Apple IIGS (running in emulation mode).

Gary B. Little, acclaimed author of Addison-Wesley's *Exploring the Apple IIGS*, now presents an in-depth analysis of the inner workings of both the GS/OS and ProDOS 8 operating systems. Written for intermediate-to-advanced Apple II programmers, ***Exploring Apple GS/OS and ProDOS 8*** is a complete reference for writing software for Apple II computers.

Little provides a thorough and detailed discussion of:

- the ProDOS file system
- GS/OS and ProDOS 8 commands for performing disk operations
- the `BASIC.SYSTEM` interpreter in the ProDOS 8 environment
- writing and installing `BASIC.SYSTEM` disk commands
- writing GS/OS and ProDOS 8 system programs
- communicating with a SmartPort disk controller
- managing interrupts from I/O devices in GS/OS and ProDOS 8
- writing and installing ProDOS 8 disk and clock drivers
- communicating with character devices using the GS/OS Console Driver

In an easy-to-read style, Little explains the intricacies of programming in GS/OS and ProDOS 8. Having written major ProDOS 8 and GS/OS applications and utilities, he is uniquely qualified to discuss the major features and functions of these operating systems. Numerous programming examples highlight and clarify vital concepts throughout the book and make ***Exploring Apple GS/OS and ProDOS 8*** the ideal guide to learning to use and maximize the power of the Apple II family of computers.

Gary B. Little is a well-known author, columnist, and software developer and is a leading authority on the Apple II family of computers. He was formerly Editor of *A+ Magazine* and currently works in the Developer Tools Group at Apple Computer, Inc. His previous books include *Inside the Apple IIe* and *Inside the Apple IIc*.

Cover design by Doliber Skeffington

Addison-Wesley
Publishing Company, Inc.

52195

9 780201 150087

ISBN 0-201-15008-5