Apple® II

Apple IIGS™ ProDOS® 16 Reference

Includes System Loader

Apple® II

Apple IIGS ProDOS® 16 Reference

Includes System Loader

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California Don Mills, Ontario
Wokingham, England Amsterdam Bonn Sydney Singapore Tokyo
Madrid Bogotá Santiago San Juan

Simultaneously published in the United States and Canada.

ISBN 0-201-17754-4
ABCDEFGHIJ-DO-89877
First printing, May 1987

LICENSING REQUIREMENTS

Apple has a licensing program that allows software developers to incorporate Apple-developed object code files into their products. A license is required for both in-house and external distribution. Before distributing any products that incorporate Apple software, please contact Software Licensing for licensing information.

Contents

Figures and tables

Preface

The *Apple IIGS ProDOS 16 Reference* is a manual for software developers, advanced programmers, and others who wish to understand the technical aspects of the Apple IIGS™ operating system. In particular, this manual will be useful to you if you want to write

- a stand-alone program that automatically runs when the computer is started up
- a routine that catalogs disks, manipulates sparse files, or otherwise interacts with the Apple IIGS file system at a basic level
- an interrupt handler
- a program that loads and runs other programs
- any program using segmented, dynamic code

The functions and calls in this manual are in assembly language format. If you are programming in assembly language, you may use the same format to access operating system features. If you are programming in a higher-level language (or if your assembler includes a ProDOS® 16 macro library), you will use library interface routines specific to your language. Those library routines are not described here; consult your language manual.

Road map to the Apple IIGS technical manuals

The Apple IIGS personal computer has many advanced features, making it more complex than earlier models of the Apple® II. To describe it fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Apple IIGS, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

The technical manuals are listed in Table P-1. Figure P-1 is a diagram showing the relationships among the different manuals.

Table P-1
The Apple IIGS technical manuals

Title	Subject
Technical Introduction to the Apple IIGS	What the Apple IIGS is
Apple IIGS Hardware Reference	Machine internals—hardware
Apple IIGS Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Apple IIGS	Concepts and a sample program
Apple IIGS Toolbox Reference: Volumes 1 and 2	How to use the Apple IIGS tools
Apple IIGS Programmer's Workshop Reference	The development environment
Apple IIGS Programmer's Workshop Assembler Reference	using the APW assembler
Apple IIGS Programmer's Workshop C Reference	Using C on the Apple IIGS
ProDOS 8 Technical Reference Manual	Standard Apple II operating system
Apple IIGS ProDOS 16 Reference	Apple IIGS operating system and loader
Human Interface Guidelines	Guidelines for the desktop interface
Apple Numerics Manual	Numerics for all Apple computers

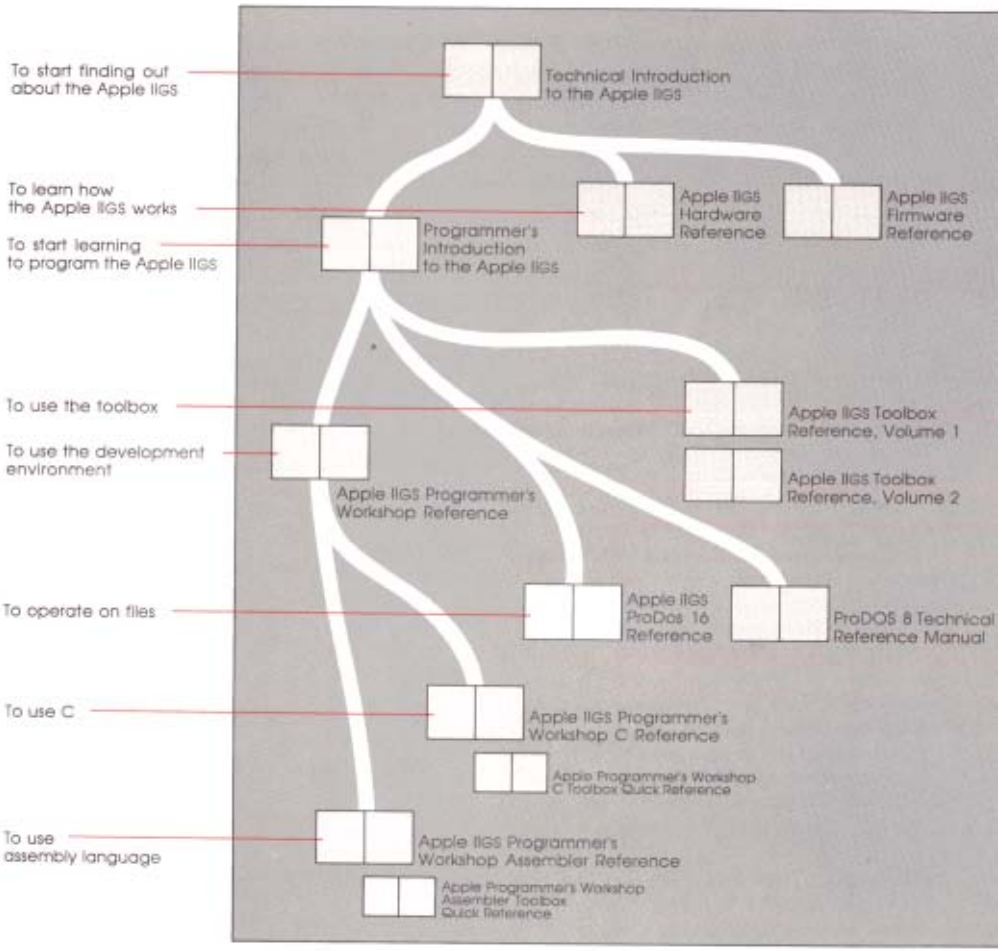


Figure P-1
Roadmap to the technical manuals

How to use this manual

The *Apple IIGS ProDOS 16 Reference* is both a reference manual and a learning tool. It is divided into several parts, to help you quickly find what you need.

- Part I describes ProDOS 16, the central part of the Apple IIGS operating system
- Part II lists and explains the ProDOS 16 operating system calls
- Part III describes the System Loader and lists all loader calls
- The final part consists of appendixes, a glossary, and an index

The first chapter in each part is introductory; read it first if you are not already familiar with the subject. The remaining chapters are primarily for reference, and need not be read in any particular order. The *ProDOS 16 Exerciser*, on a diskette included with the manual, provides a way to practice making ProDOS 16 calls before actually coding them.

This manual does not explain 65C816 assembly language. Refer to *Apple IIGS Programmer's Workshop Assembler Reference* for information on Apple IIGS assembly language programming.

This manual does not give a detailed description of ProDOS 8, the Apple II operating system from which ProDOS 16 was derived. For a synopsis of the differences between ProDOS 8 and ProDOS 16, see Chapter 1 of this manual. For more detailed information on ProDOS 8, see *ProDOS 8 Technical Reference Manual*.

Other materials you'll need

Hardware and software

To use the products described in this manual, you will need an Apple IIGS with at least one external disk drive (Apple recommends two drives). ProDOS 16 and the System Loader require only the minimum memory configuration (256K RAM), although Apple IIGS Programmer's Workshop and many application programs may require more memory.

You will also need an Apple IIGS system disk. A system disk contains ProDOS 16, ProDOS 8, the System Loader, and other system software necessary for proper functioning of the computer. A system disk may also contain application programs.

If you wish to practice making ProDOS 16 operating system calls you will need the ProDOS 16 *Exerciser*, a program on the diskette included with this manual.

Publications

This manual is the only reference for ProDOS 16 and the System Loader. You may find useful related information in any of the publications listed under "Roadmap to Apple IIGS Technical Manuals" in this preface; in particular, you may wish to refer to the following:

- **The technical introduction:** The *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals about the Apple IIGS. It describes all aspects of the Apple IIGS, including its features and general design, the program environments, the toolbox, and the development environment.
- **The programmer's introduction:** When you start writing programs for the Apple IIGS , the *Programmer's Introduction to the Apple IIGS* provides the concepts and guidelines you need. It is a starting point for programmers writing **event-driven** and segmented applications that use routines in the Apple IIGS Toolbox.
- **The firmware reference manual:** The *Apple IIGS Firmware Reference* describes the routines that are stored in the machine's read-only memory (ROM); it includes information about interrupt routines and low-level I/O subroutines for the serial ports and disk port. The *Firmware Reference* also describes the Monitor, a low-level programming and debugging aid for assembly-language programs.
- **The toolbox manuals:** Like the Macintosh™, the Apple IIGS has a built-in toolbox. The two volumes of the *Apple IIGS Toolbox Reference* introduce concepts and terminology, show how to use the tools, and tell how to write and install your own tool set. They also describe the workings of some of the system-level tool sets, such as the Memory manager, that interact closely with proDOS 16 and the System Loader.

- **The Programmer's Workshop manuals:** The development environment on the Apple IIGS is the Apple IIGS Programmer's Workshop (APW). APW is a set of programs that enable you to create and debug application programs on the Apple IIGS. The *Apple IIGS Programmer's Workshop Reference* includes information about the parts of the workshop that are independent of programming language: the shell, the editor, the linker, the debugger, and the utilities. In addition, there is a separate reference manual for each programming language. The manuals for the languages Apple provides are the *Apple IIGS Programmer's Workshop Assembler Reference* and the *Apple IIGS Programmer's Workshop C Reference*.
- **The ProDOS 8 manual:** ProDOS 8 (previously called just *ProDOS*) is compatible with all Apple II computers, including the Apple IIGS. As a developer of Apple IIGS programs, you may need to refer to the *ProDOS 8 Technical Reference Manual* if you are developing programs to run on standard Apple II's as well as on the Apple IIGS, or if you are converting a ProDOS 8-based program to run under ProDOS 16.

Notations and conventions

To help make the manual more understandable, the following conventions and definitions apply throughout.

Terminology

This manual may define certain terms, such as Apple II and ProDOS, slightly differently than what you are used to. Please note:

- **Apple II:** A general reference to the Apple II family of computers, especially those that may use ProDOS 8 or ProDOS 16 as an operating system. It includes the 64k Apple II Plus, the Apple IIc, the Apple IIe, and the Apple IIGS.
- **standard Apple II:** Any Apple II computer that is *not* an Apple IIGS. Since previous members of the Apple II family share many characteristics, it is useful to distinguish them as a group from the Apple IIGS. A standard Apple II may also be called an *8-bit Apple II*, because of the 8-bit registers in its 6502 or 65C02 microprocessor.

- **ProDOS:** A general term describing the family of operating systems developed for Apple II computers. It includes both ProDOS 8 and ProDOS 16; it does not include DOS 3.3 or SOS.
- **ProDOS 8:** The 8-bit ProDOS operating system, through version 1.2, originally developed for standard Apple II computers but compatible with the Apple IIGS. In previous Apple II documentation, ProDOS 8 is called simply *ProDOS*.
- **ProDOS 16:** A 16-bit operating system developed for the Apple IIGS computer. It is the system described in this manual.

Typographic conventions

Each new term introduced in this manual is printed first in **bold** type. That lets you know that the term has not been defined earlier, and also indicates that there is an entry for it in the glossary.

Assembly language labels, entry points, routine names, and file names that appear in text passages are printed in a special typeface (for example, `name_length` and `GET_ENTRY`). Function names that are English language terms are printed with initial caps (for example, Load Segment By Number). When the name of a label or variable is used to mean the *value of* that variable rather than its name, the word is printed in italics (for example, "the first *name_length* bytes of this field contain the volume name...").

Watch for these

The following words mark special messages to you:

❖ *Note:* Text set off in this manner—with a word or phrase such as *Note* or *By the way*—presents sidelights or interesting points of information.

Important Text set off in this manner—with the word **Important**—presents important information or instructions.

Warning Text set off in this manner—with the word **Warning**—indicates potential serious problems.

Part I

How ProDOS 16 Works

This part of the manual gives a general description of ProDOS 16. ProDOS 16 is the **disk operating system** for the Apple IIGS; it provides file management and input/output capabilities, and controls certain other aspects of the Apple IIGS **operating environment.**

Chapter 1

About ProDOS 16

This chapter introduces ProDOS 16. It gives background information on the development of ProDOS 16, followed by an overview of ProDOS 16 in relation to the Apple IIGS. A brief comparison of ProDOS 16 with ProDOS 8, its closest relative in the Apple II world, is followed by a reference list of the most pertinent ProDOS 16 features.

The chapter's organization roughly parallels that of Part I as a whole. Each section refers you to the appropriate chapter for more information on each aspect of ProDOS 16.

Background

The Apple IIGS is the latest Apple II computer. Its microprocessor, the 65C816, is a successor to the standard Apple IIs' 6502 and functions in both 8-bit (6502 **emulation**) mode and 16-bit (**native**) mode (see *Technical Introduction to the Apple IIGS*). In accordance with the design philosophy governing all Apple II family products, the Apple IIGS is compatible with standard Apple II software—most presently available Apple II, Apple IIc, and Apple IIe applications will run without modification on the Apple IIGS.

To retain this compatibility while adding new features, the Apple IIGS requires two separate operating systems, ProDOS 8 and ProDOS 16:

- ProDOS 8 is the operating system for standard Apple II computers. The Apple IIGS uses ProDOS 8 and puts the processor into emulation mode in order to run standard–Apple II applications.
- ProDOS 16 is a newly developed system; it takes advantage of Apple IIGS features that standard Apple II computers do not have. The Apple IIGS uses ProDOS 16 and puts the processor into native mode in order to run Apple IIGS applications.

The user need not worry about which operating system is active at any one time. Whenever the Apple IIGS loads an application, it automatically loads the proper operating system for it.

ProDOS 8 on the Apple IIGS functions identically to ProDOS 8 on other Apple II computers. For a complete description of ProDOS 8, see *ProDOS 8 Technical Reference Manual*.

What Is ProDOS 16?

ProDOS 16 is the central part, or **kernel**, of the Apple IIGS operating system. Although other software components (such as the System Loader described in this manual) may be thought of as parts of the overall operating system, ProDOS 16 is the key component. It manages the creation and modification of files. It accesses the **disk devices** on which the files are stored and retrieved. It dispatches interrupt signals to **interrupt handlers**. It also controls certain aspects of the Apple IIGS operating environment, such as pathname prefixes and procedures for quitting programs and starting new ones.

Programming levels in the Apple IIGS

Figure 1-1 is a simplified logical diagram of the Apple IIGS, from a programmer's point of view. Boxes representing parts of the system form a vertical hierarchy; arrows between the boxes show the flow of control or execution from one level to the next. At the highest level is the programmer or user; he directly manipulates the execution of the application program that runs on the machine. The application, in turn, interacts directly with the next lower level of software—the operating system. The operating system interacts with the very lowest level of software in the machine: the built-in firmware and toolbox routines. Those routines directly manipulate the switches, registers, and input/output devices that constitute the computer's hardware.

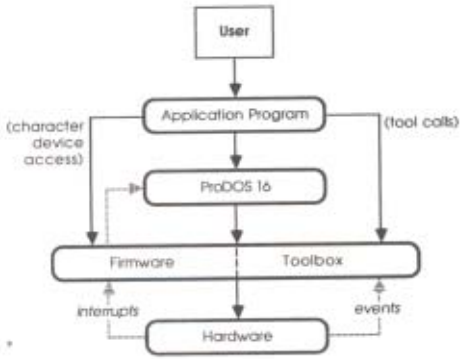


Figure 1-1
Programming levels in the Apple IIGS

This hierarchical view shows that the operating system is an intermediary between the application program and the computer hardware. A program need not know the details of individual hardware devices it accesses; instead, it makes operating system **calls**. The operating system then translates those calls into the proper instructions for whatever devices are connected to the system.

The lowest software level, between the operating system and hardware, is extensively developed in the Apple IIGS. It consists of two parts: the **firmware**, a collection of traditional ROM-based routines for performing such tasks as character I/O, interrupt handling, and memory manipulation; and the **toolbox**, a large set of assembly-language routines and macros useful to all levels of software. As the arrows on Figure 1-1 show, ProDOS 16 accesses the firmware/tools level of the Apple IIGS directly, but so do application programs. In other words, for tool calls and certain types of I/O, applications *bypass* ProDOS 16 and interact directly with low-level system software.

The arrows pointing *upward* along the diagram show a counterflow of information, in which lower levels in the machine notify higher levels of important hardware conditions. **Interrupts** from hardware devices are handled both by firmware and by ProDOS 16; **events** are similar to interrupts but are handled by applications through tool calls.

Disks, volumes, and files

ProDOS 16 communicates with several different types of disk drives, but the type of drive and its physical location (slot or port number) need not be known to a program that wants to access that drive. Instead, a program makes calls to ProDOS 16, identifying the disk it wants to access by its *volume name* or *device name*.

Information on a volume is divided into files. A **file** is an ordered collection of bytes that has several attributes, including a name and a file type. Files are either **standard files** (containing any type of code or data) or **directory files** (containing the names and disk locations of other files). When a disk is initially formatted, its **volume directory** file is created; the volume directory has the same name as the volume itself.

ProDOS 16 supports a **hierarchical file system**, meaning that volume directories can contain the names of either files or other directories, called **subdirectories**; subdirectories in turn can contain the names of files or other subdirectories. In a hierarchical file system, a file is identified by its **pathname**, a sequence of file names starting with the volume directory and ending with the name of the file. Figure 1-2 shows the relationships among files in a hierarchical file system.

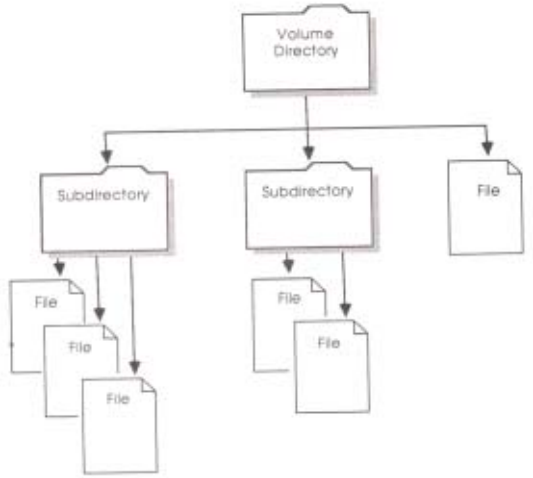


Figure 1-2
Example of a hierarchical file structure

See Chapter 2 and Appendix A for detailed information on ProDOS 16's file structure, organization, and formats.

Memory use

ProDOS 16 and application programs on the Apple IIGS are relieved of most memory management tasks. The Memory Manager, an Apple IIGS tool set, allocates all memory space, keeps track of available memory, and frees memory no longer needed by programs. If a program needs to allocate some memory space, it requests the space through a call to the Memory Manager. If a program makes a ProDOS 16 call that results in memory allocation, ProDOS 16 requests the space from the Memory Manager and allocates it to the program.

The Memory Manager is described further in Chapter 3 of this manual, and in *Apple IIGS Toolbox Reference*.

External devices

ProDOS 16 communicates only with **block devices**, such as disk drives. Programs that wish to access **character devices** such as printers and communication ports must do so directly, either through the device firmware or through Apple IIGS Toolbox routines written for those devices. See *Apple IIGS Firmware Reference* and *Apple IIGS Toolbox Reference*.

Certain devices generate interrupts to tell the computer that the device needs attention. ProDOS 16 is able to handle up to 16 interrupting devices. You may place an interrupt-handling routine into service through a ProDOS 16 call; your routine will then be called each time an interrupt occurs. If you install more than one routine, the routines will be polled in the order in which they were installed.

You may also remove an interrupt routine with a ProDOS 16 call. In writing, installing, and removing interrupt handling routines, be sure to follow the conventions and requirements given in Chapter 7, "Adding Routines to ProDOS 16."

ProDOS 16 and ProDOS 8

ProDOS 16, although derived from ProDOS 8, adds several capabilities to support the new features and operating configurations of the Apple IIGS. For example:

- Because the 65C816 microprocessor functions in both 8-bit (emulation) and 16-bit (native) execution modes, ProDOS 16 is designed to accept system calls from applications running in either 8-bit or 16-bit mode. ProDOS 8 accepts system calls from applications running in 8-bit mode only.
- Because the Apple IIGS has a total addressable memory space of 16Mb, ProDOS 16 has the ability to accept system calls from anywhere in that memory space (addresses up to $FF FFFF), and those calls can manipulate data anywhere in memory. Under ProDOS 8, system calls can be made from memory addresses below $FFFF only—the lowest 64K of memory.

- ProDOS 16 relies on a sophisticated memory management system (see Chapter 3), instead of the simple global page bit map used by ProDOS 8.
- Applications under ProDOS 16 must make calls to allocate memory or to access system global variables, such as date and time, system level, and I/O buffer addresses. ProDOS 8 maintains that information in the system global page in memory bank $00, but under ProDOS 16 the global page is not supported.
- ProDOS 16 also provides several programming conveniences not available under ProDOS 8, including named devices and multiple, user-definable file prefixes.

Upward compatibility

In a strict sense, ProDOS 16 is not upwardly compatible from ProDOS 8. Programs written to function under ProDOS 8 on an Apple II will not run on the Apple IIGS, *under ProDOS 16*, without some modification. Conceptually, however, ProDOS 16 *is* upwardly compatible from ProDOS 8, in at least two ways:

1. The two operating systems are themselves similar in structure:
   - The set of ProDOS 16 system calls is a superset of the ProDOS 8 calls; for (almost) every ProDOS 8 system call, there is a functionally equivalent ProDOS 16 call, usually with the same name.
   - The calls are made in nearly identical ways in both ProDOS systems, and the parameter blocks for passing values to functions are laid out similarly.
   - ProDOS 16 uses exactly the same file system as ProDOS 8. It can read from and write to any disk volume produced by ProDOS 8.
2. Both operating systems are included with the Apple IIGS. Most applications written for ProDOS 8 on standard Apple II computers will run without modification on the Apple IIGS—not under ProDOS 16, but under ProDOS 8.

Thus, even though the individual operating systems are not completely compatible, their *sum* on the Apple IIGs computer is completely upwardly compatible from other Apple II computers. You never need be concerned with which operating system is functioning—if you run an Apple II application, ProDOS 8 is automatically loaded; if you run an Apple IIGS application, ProDOS 16 is automatically loaded. Chapter 5 explains the details of how this is accomplished.

Downward compatibility

ProDOS 16 is not downwardly compatible to ProDOS 8. Applications written for ProDOS 16 will not run on the Apple II, IIc, or IIe. The extra memory needed by Apple IIGS applications and the additional instructions recognized by the 65C816 microprocessor make applications written for ProDOS 16 incompatible with standard Apple II computers.

Eliminated ProDOS 8 system calls

As mentioned under "Upward Compatibility," most ProDOS 8 calls have functionally exact equivalents in ProDOS 16. However, some ProDOS 8 calls do not appear in ProDOS 16 because they are unnecessary. The eliminated calls are

`RENAME`	The ProDOS 16 `CHANGE_PATH` call performs the same function.
`GET_TIME`	Under ProDOS 16, the time and date are obtained through a call to the Miscellaneous Tool Set (see *Apple IIGS Toolbox Reference*).
`SET_BUF`	Under ProDOS 16, the Memory Manager, rather than the application, allocates file I/O buffers.
`GET_BUF`	This call is unnecessary under ProDOS 16 because the `OPEN` call returns a handle to the file's I/O buffer.
`ONLINE`	This call is replaced in ProDOS 16 by the `VOLUME` call.

New ProDOS 16 system calls

The following operating system calls, not recognized by ProDOS 8, are part of ProDOS 16:

`CLEAR_BACKUP_BIT`	(clears one of a file's access bits)
`CHANGE_PATH`	(changes the pathname of a file within a volume)
`SET_LEVEL`	(sets the system file level)
`GET_LEVEL`	(returns the system file level)
`GET_DEV_NUM`	(returns the device number for a named device)
`GET_LAST_DEV`	(returns the number of the last device accessed)
`FORMAT`	(formats a disk volume)
`GET_NAME`	(returns the filename of the current application)
`GET_BOOT_VOL`	(returns the name of the volume that contains ProDOS 16)
`GET_VERSION`	(returns the current ProDOS 16 version)

These and all other ProDOS 16 calls are described in detail in Chapters 9 through 13.

Other features

Like ProDOS 8, ProDOS 16 supports block devices only. It does not support I/O operations for the built-in serial ports, mouse, Apple Desktop Bus™, sound generation system, or any other nonblock device. Applications must access these devices through the device firmware or the Apple IIGS Toolbox.

ProDOS 8 and ProDOS 16 have identical file structures. Each can *read* the other's files, but

- ProDOS 16 load files (types $B3 - $BE) cannot be executed under ProDOS 8
- ProDOS 8 system files (type $FF) or binary files (type $06) cannot be executed under ProDOS 16

The default operating system on the Apple IIGS (after a cold or warm restart) can be either ProDOS 8 or ProDOS 16, depending on the organization of files on the startup disk. See "System Startup" in Chapter 5.

Running under ProDOS 8 does not disable memory beyond the addresses ProDOS 8 can reach, nor does it disable any other advanced Apple IIGS features. All system resources are always available, even though an application itself may make use of only the "ProDOS 8–standard Apple II" portion.

Summary of ProDOS 16 features

The following lists summarize the principal features of ProDOS 16. Refer to the glossary and to appropriate chapters for definitions and explanations of terms that may be unfamiliar to you.

In general, ProDOS 16...

- is a single-task operating system
- supports a hierarchical, tree-structured file system
- allows device-independent I/O for block devices

ProDOS 16 system calls...

- use the JSL instruction and a parameter block
- return error status in the A and P registers
- preserve all other CPU registers
- can be made from 65C816 native mode or 6502 emulation mode
- can be made from anywhere in memory
- can access parameter blocks that are anywhere in memory
- can use pointers that point anywhere in memory
- can transfer data anywhere in memory

The ProDOS 16 file management system...

- uses a hierarchical file structure
- supports 9 pathname prefixes
- allows byte-oriented access to both directory files and data files
- allocates files dynamically and noncontiguously on block devices
- supports sparse files
- provides buffers automatically
- supports access attributes that enable/disable
  - reading
  - writing
  - renaming
  - destroying
  - backup
- assigns a system file level to open files
- automatically marks files with date and time
- uses a 512-byte block size
- allows volume sizes up to 32 megabytes
- allows data file sizes up to 16 megabytes
- allows up to 14 volumes on line
- allows up to 8 open files
- allows 64 characters per pathname
- allows 64-character prefixes
- allows 15 characters per volume name
- allows 15 characters per file name

The ProDOS 16 device management system...

- supports the ProDOS block device protocol
- names each block device
- allows 15 characters per device name
- allows 14 devices on line simultaneously
- provides a `FORMAT` call to initialize disks

The ProDOS 16 interrupt management system...

- receives hardware interrupts not handled by firmware
- dispatches interrupts to user-provided interrupt handlers
- allows installation of up to 16 interrupt handlers

For memory management, ProDOS 16...

- dynamically allocates and releases system buffers (through the Memory Manager)
- can directly access up to 2^{24} bytes (16 megabytes) of memory
- can run with a minimum of 256K memory

In addition, ProDOS 16...

- provides a `QUIT` call to cleanly exit one program and start another, with the option of returning later to the quitting program

Chapter 2

ProDOS 16 Files

The largest part of ProDOS 16 is its file management system. This chapter explains how files are named, how they are created and used, and a little about how they are organized on disks. It discusses ProDOS 16 *file access* and *file housekeeping* calls.

For more details on file format and organization, see Appendix A.

Using files

Filenames

Every ProDOS 16 file, whether it is a directory file, data file, or program file, is identified by a **filename**. A ProDOS 16 filename can be up to 15 characters long. It must begin with a letter, and may contain uppercase letters (A-Z), digits (0-9), and periods (.). Lowercase letters are automatically converted to uppercase. A filename must be unique within its directory. Some examples (taken from Figure 2-1) are

```
MEMOS
CHAP11
MY.PROGRAM
```

An entire disk is identified by its **volume name**, which is the filename of its volume directory. In Figure 2-1, the disk's volume name is `/DISK86`.

Pathnames

A ProDOS 16 pathname is a series of filenames, each preceded by a slash (/). The first filename in a pathname is the name of a volume directory. Successive filenames indicate the path, from the volume directory to the file, that ProDOS 16 must follow to find a particular file. The maximum length for a pathname is 64 characters, including slashes. Examples from Figure 2-1 are

```
/DISK86/CHARTS/SALES.JUN
/DISK86/MY.PROGRAM
/DISK86/MEMOS/CHAP11
```

All calls that require you to name a file will accept either a full pathname or a **partial pathname**. A partial pathname is a portion of a pathname; you can tell that it is not a full pathname because it doesn't begin with a slash and a volume name. The maximum length for a partial pathname is 64 characters, including slashes.

These partial pathnames are all derived from the sample pathnames above:

```
SALES.JUN
MY.PROGRAM
MEMOS/CHAP11
CHAP11
```

ProDOS 16 automatically adds a **prefix** to the front of partial pathnames to form full pathnames. A prefix is a pathname that indicates a directory; it always begins with a slash and a volume name. Several prefixes are stored internally by ProDOS 16.

For the partial pathnames listed above to indicate the proper files, their prefixes should be set to

```
/DISK86/CHARTS/
/DISK86/
/DISK86/
/DISK86/MEMOS/
```

respectively. The slashes at the end of these prefixes are optional; however, they are convenient reminders that prefixes indicate directory files.

The maximum length for a prefix is 64 characters. The minimum length for a prefix is zero characters, known as a **null prefix**. You set and read prefixes using the calls `SET_PREFIX` and `GET_PREFIX`.

❖ *Note:* Because both a prefix and a partial pathname can be up to 64 characters long, it is possibe to have a pathname (prefix plus partial pathname) whose *effective* length is up to 128 characters.

ProDOS 16 allows you to set more than one prefix, and then refer to each prefix by code numbers. When, as in the above examples, no particular prefix number is specified, ProDOS 16 adds the **default prefix** to the partial pathname you provide. See Chapter 5 for a more complete explanation and examples.

Figure 2-1 illustrates a hypothetical directory structure; it contains all the files mentioned above. Note that, even though there are two files named `PROFIT.3RD` in the volume directory `/DISK.86/`, they are easily distinguished because they are in different subdirectories (`MEMOS/` and `CHARTS/`). That is why a full pathname is necessary to completely specify a file.

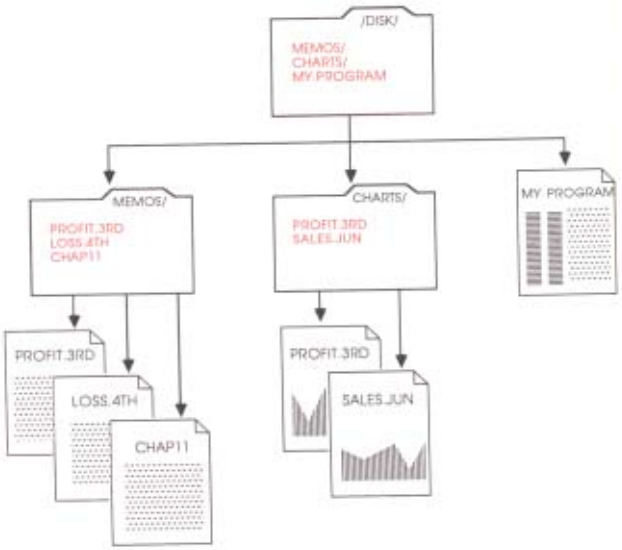


Figure 2-1
Example of a ProDOS 16 file structure

Creating files

A file is placed on a disk by the `CREATE` call. When you create a file, you assign it the following properties:

- A **pathname**. This pathname is a unique path by which the file can be identified and accessed. This pathname must place the file within an existing directory.
- An **access byte**. The value of this byte determines whether or not the file can be written to, read from, destroyed, or renamed.
- A **file type**. This byte indicates to other applications the type of information to be stored in the file. It does not affect, in any way, the contents of the file.
- A **storage type**. This byte determines the physical format of the file on the disk. There are only two different formats: one is used for directory files, the other for non-directory files.

When you create a file, the properties listed above are placed on the disk, along with the current system date and time (called **creation date** and **creation time**), in a format as shown in Appendix A. Once a file has been created, it remains on the disk until it is deleted (using the `DESTROY` call).

To check what the properties for a given file are, use the `GET_FILE_INFO` call. To alter its properties, use the `SET_FILE_INFO` call. To change the file's name, use the `CHANGE_PATH` call.

Opening files

Before you can read information from or write information to a file that has been created, you must use the `OPEN` call to open the file for access. When you open a file you specify it by pathname. The pathname you give must indicate an existing file; the file must be on a disk mounted in a disk drive.

The `OPEN` call returns a reference number (*ref_num*) and the location of a buffer (*io_buffer*) to be used for transferring data to and from the file. All subsequent references to the open file must use its reference number. The file remains open until you use the `CLOSE` call.

Each open file's I/O buffer is used by the system the entire time the file is open. Thus, to conserve memory space, it is wise to keep as few files open as possible. ProDOS 16 allows a maximum of 8 open files at a time.

When you open a file, some of the file's characteristics are placed into a region of memory called a **file control block**. Several of these characteristics—the location in memory of the file's buffer, a pointer to the end of the file (the **EOF**), and a pointer to the current position in the file (the file **Mark**)—are accessible to applications via ProDOS 16 calls, and may be changed while the file is open.

It is important to be aware of the differences between the file as it exists on the disk and when it is open in memory. Although some of the file's characteristics and some of its data may be in memory at any given time, the file itself still resides on the disk. This allows ProDOS 16 to manipulate files that are much larger than the computer's memory capacity. As an application writes to the file and changes its characteristics, new data and characteristics are written to the disk.

The EOF and Mark

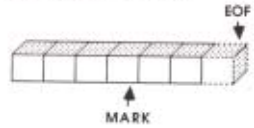
To aid reading from and writing to files, each open file has one pointer indicating the end of the file (the EOF), and another defining the current position in the file (the Mark). ProDOS 16 moves both EOF and Mark automatically when necessary, but an application program can also move them independently of ProDOS 16.

The EOF is the number of readable bytes in the file. Since the first byte in a file has number 0, the EOF, when treated as a pointer, points one position past the last character in the file.

When a file is opened, the Mark is set to indicate the first byte in the file. It is automatically moved forward one byte for each byte written to or read from the file. The Mark, then, always indicates the next byte to be read from the file, or the next byte position in which to write new data. It cannot exceed the EOF.

If during a write operation the Mark meets the EOF, both the Mark and the EOF are moved forward one position for every additional byte written to the file. Thus, adding bytes to the end of the file automatically advances the EOF to accommodate the new information. Figure 2-2 illustrates the relationship between the Mark and the EOF.

(a.) Beginning position



(b.) After writing or reading two bytes:

(c.) After writing two more bytes:

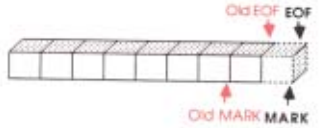


Figure 2-2
Automatic movement of EOF and Mark

An application can place the EOF anywhere, from the current Mark position to the maximum possible byte position. The Mark can be placed anywhere from the first byte in the file to the EOF. These two functions can be accomplished using the `SET_EOF` and `SET_MARK` calls. The current values of the EOF and the Mark can be determined using the `GET_EOF` and `GET_MARK` calls.

Reading and writing files

READ and WRITE calls to ProDOS 16 transfer data between memory and a file. For both calls, the application must specify three things:

- The reference number of the file (assigned when the file was opened).
- The location in memory of a buffer (`data_buffer`) that contains, or is to contain, the transferred data. Note that this cannot be the same buffer (`io_buffer`) whose location was returned when the file was opened.
- The number of bytes to be transferred.

When the request has been carried out, ProDOS 16 passes back to the application the number of bytes that it actually transferred.

A read or write request starts at the current Mark, and continues until the requested number of bytes has been transferred (or, on a read, until the end-of-file has been reached). Read requests can also terminate when a specified character is read. To turn on this feature and set the character(s) on which reads terminate, use the NEWLINE call. The newline read mode is typically used for reading lines of text that are terminated by carriage returns.

❖ *By the way:* Neither a READ nor a WRITE call necessarily causes a disk access. ProDOS I/O buffer for each open file is 1024 bytes in size, and can hold one block (512 bytes) of data; it is only when a read or write crosses a block boundary that a disk access occurs.

Closing and flushing files

When you finish reading from or writing to a file, you must use the CLOSE call to close the file. When you use this call, you specify only the reference number of the file (assigned when the file was opened).

CLOSE writes any unwritten data from the file's I/O buffer to the file, and it updates the file's size in the directory, if necessary. Then it frees the 1024-byte buffer space for other uses and releases the file's reference number and file control block. To access the file once again, you have to reopen it.

Information in the file's directory, such as the file's size, is normally updated only when the file is closed. If the user were to press Control-Reset (typically halting the current program) while a file is open, data written to the file since it was opened could be lost, and the integrity of the disk could be damaged. This can be prevented by using the FLUSH call.

FLUSH, like CLOSE, writes any unwritten data from the file's I/O buffer to the file, and updates the file's size in the directory. However, it keeps the file's buffer space and reference number active, and allows continued access to the file. In other words, the file stays open. If the user presses Control-Reset while an open but flushed file is in memory, there is no loss of data and no damage to the disk.

Both the CLOSE and FLUSH calls, when used with a reference number of 0, normally cause all open files to be closed or flushed. Specific groups of files can be closed or flushed using the *system file level* (see next).

File levels

When a file is opened, it is assigned a level, according to the value of a specific byte in memory (the **system file level**). If the file level is never changed, the CLOSE and FLUSH calls, when used with a reference number of 0, cause all open files to be closed or flushed. But if the level has been changed since the first file was opened, only those files opened when the file was greater than or equal to the current system file level are closed or flushed.

The system file level feature may be used, for example, by a controlling program such as a BASIC interpreter to implement an EXEC command:

1. The interpreter opens an EXEC program file when the level is $00.
2. The interpreter then sets the level to, say, $07.
3. The EXEC program opens whatever files it needs.
4. The EXEC program executes a BASIC CLOSE command, to close all the files it has opened. All files at or above level $07 are closed, but the EXEC file itself remains open.

You assign a value to the system file level with a SET_LEVEL call; you obtain the current value by making a GET_LEVEL call.

File format and organization

This portion of the chapter describes in general terms the organization of files on a disk. For more detailed information, see Appendix A.

In general, *structure* refers in this manual to the hierarchical relationships among files—directories, subdirectories, and files. *Format* refers to the arrangement of information (such as headers, pointers and data) within a file. *Organization* refers to the manner in which a single file is stored on disk, in terms of individual 512-byte **blocks**. The three concepts are separate but interrelated. For example, because of ProDOS 16's hierarchical file *structure*, part of the *format* of a directory file includes pointers to the files within that directory. Also, because files are *organized* as noncontiguous blocks on disk, part of the *format* of every file larger than one block includes pointers to other blocks.

Directory files and standard files

Every ProDOS 16 file is a named, ordered sequence of bytes that can be read from, and to which the rules of Mark and EOF apply. However, there are two types of files: **directory files** and **standard files**. Directory files are special files that describe and point to other files on the disk. They may be read from, but not written to (except by ProDOS 16). All nondirectory files are standard files. They may be read from and written to.

A directory file contains a number of similar elements, called *entries*. The first entry in a directory file is the header entry: it holds the name and other properties (such as the number of files stored in that directory) of the directory file. Each subsequent entry in the file describes and points to some other file on the disk. Figure 2-3 shows the format of a directory file.

The files described and pointed to by the entries in a directory file can be standard files or other directory files.

An application does not need to know the details of directory format to access files with known names. Only operations on unknown files (such as listing the files in a directory) require the application to examine a directory's entries. For such tasks, refer to Appendix A.

Standard files have no such predefined internal format: the arrangement of the data depends on the specific file type.

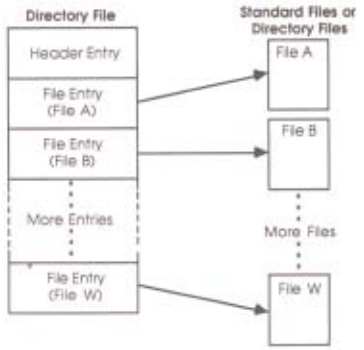


Figure 2-3
Directory file format

File organization

Because directory files are generally smaller than standard files, and because they are sequentially accessed, ProDOS 16 uses a simpler form of storage for directory files than it does for standard files. Both types of files are stored as a set of 512-byte blocks, but the way in which the blocks are arranged on the disk differs.

A directory file is a linked list of blocks: each block in a directory file contains a pointer to the next block in the directory file as well as a pointer to the previous block in the directory. Figure 2-4 illustrates this organization.

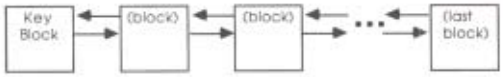


Figure 2-4
Block organization of a directory file

Data files, on the other hand, are often quite large, and their contents may be randomly accessed. It would be very slow to access such large files if they were organized sequentially. Instead, ProDOS 16 stores standard files using a **tree organization**. The largest possible standard file has a **master index block** that points to 128 **index blocks**. Each index block points to 256 **data blocks** and each data block can hold 512 bytes of data. The block organization of the largest possible standard file is shown in Figure 2-5.

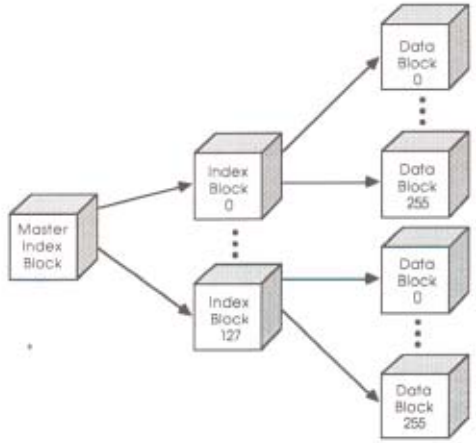


Figure 2-5
Block organization of a standard file

Most standard files do not have this exact organization. ProDOS 16 only writes a subset of this form to the file, depending on the amount of data written. This technique produces three distinct forms of standard file: seedling, sapling, and tree files. All three are explained in Appendix A.

Sparse files

In most instances a program writes data sequentially into a file. But by writing data, moving the EOF and Mark, and then writing more data, a program can also write nonsequential data to a file. For example, a program can open a file, write a few characters of data, and then move the EOF and Mark (thereby making the file bigger) by an arbitrary amount before writing a few more bytes of data. Only those blocks that contain nonzero information are actually allocated for the file, so it may take up as few as three blocks on the disk (a total of 1536 bytes). However, as many bytes as are specified by the value of EOF (up to 16 megabytes) can potentially be read from it. Such files are known as **sparse files**. Sparse files are explained in more detail in Appendix A.

Important In transferring sparse files, the fact that more data can be read from the file than actually resides on the disk can cause a problem. Suppose that you were trying to copy a sparse file from one disk to another. If you were to read data from one file and write it to another, the new file would be much larger than the original because data that is not actually on the disk can be read from the file. Thus if your application is going to transfer sparse files, you must use the information in Appendix A to determine which blocks should be copied, and which should not.

The file utility programs supplied with the Apple IIGS automatically preserve the structure of sparse files on a copy.

Chapter 3

ProDOS 16 and Apple IIGS Memory

Strictly speaking, memory management is separate from the operating system in the Apple IIGS. This chapter shows how ProDOS 16 uses memory and how it interacts with the Memory Manager.

Apple IIGS memory configurations

The Apple IIGS microprocessor is capable of directly addressing 16 megabytes (16Mb) of memory. As shipped, the basic memory configuration for Apple IIGS is 256 kilobytes (256K) of RAM and 128K of ROM, arranged within the 16Mb memory space as shown in Figure 3-1.

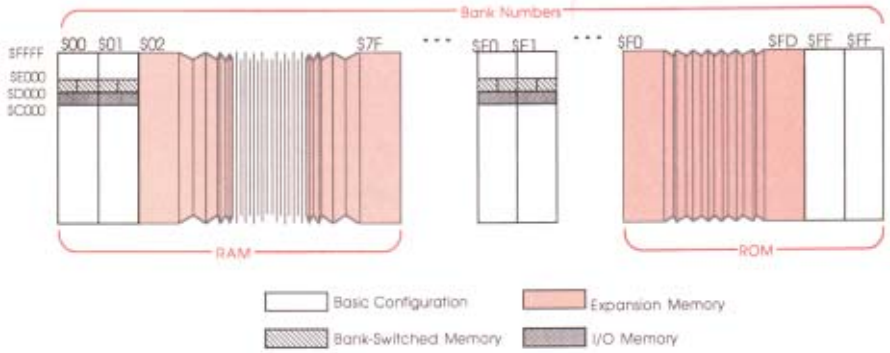


Figure 3-1
Apple IIGS memory map

The total memory space is divided into 256 **banks** of 64K bytes each (see Table 3-1). Banks $00 and $01 are used for system software, ProDOS 16 applications, and are the only memory space occupied by standard-Apple II programs running under ProDOS 8. Banks $E0 and $E1 are used principally for high-resolution video display, additional system software, and RAM-based tools. Specialized areas of RAM in these banks include I/O space, bank-switched memory, and display buffers in locations consistent with standard Apple II memory configurations (see "Special Memory and Shadowing," below). Banks $FF and $FE are ROM; they contain firmware and ROM-based tools. For more detailed pictures of Apple IIGS Memory, see *Technical Introduction to the Apple IIGS*, *Apple IIGS Hardware Reference* and *Apple IIGS Firmware Reference*.

Table 3-1
Apple IIGS memory units

Unit	Size
nibble	4 bits (one-half byte)
byte	8 bits
word	2 bytes
long word	4 bytes
page	256 bytes
block	512 bytes (for disk storage)
bank	65,536 bytes (256 pages)

With a 1-megabyte Apple IIGS Memory Expansion Card, 16 additional banks of memory are made available; they are numbered sequentially, from $02 to $11. Expansion banks have none of the specialized memory areas shown for banks $00-$01 and $E0-$E1—all 64K bytes in each bank are available for applications.

Special memory and shadowing

For running standard Apple II software, the Apple IIGS memory configuration is set so that banks $00 and $01 are identical to the Main and Auxiliary RAM and ROM on an Apple IIc or an Apple IIe with extended 80-column card. See *Apple IIc Technical Reference Manual* or *Apple IIe Technical Reference Manual* for details. Because they are used by standard Apple II programs, both banks $00 and $01, as well as the display pages in banks $E0 and $E1, are called **special memory**; there are restrictions on the placement of certain types of code in special memory. For example, any system software that must remain active in the standard Apple II configuration cannot be put in special memory. See "Memory Manager" in *Apple IIGS Toolbox Reference* for more details.

Shadowing is the term used to describe a process whereby any changes made to one part of the Apple IIGS memory are automatically and simultaneously made in another part. Shadowing is necesssary because standard Apple II programs can directly access banks $00 and $01 only, but all the fixed locations and data structures needed by those programs are maintained in banks $E0 and $E1 (see *Apple IIGS Hardware Reference*). When the proper shadowing is on, an application may, for example, update a display location in bank $00; that information is automatically shadowed to bank $E0, from where the video display is actually controlled.

ProDOS 16 and System Loader memory map

ProDOS 16 and the System Loader together occupy nearly all addresses from $D000 through $FFFF in both banks $00 and $01. This is the same memory space that ProDOS 8 occupies in a standard Apple II: all of the language card area (addresses above $D000), including most of bank-switched memory.

In addition, ProDOS 16 reserves (through the Memory Manager) approximately 10.7K bytes just below $C000 in bank $00 (in the region normally occupied by BASIC.SYSTEM in a standard Apple II), for I/O buffers, ProDOS 8 interface tables, and other code.

The part of ProDOS 16 that controls loading of both ProDOS 16 and ProDOS 8 programs is located in parts of bank-switched memory in banks $E0 and $E1. Other system software occupies most of the rest of the language card areas of banks $E0 and $E1.

None of these reserved memory areas is available for use by applications.

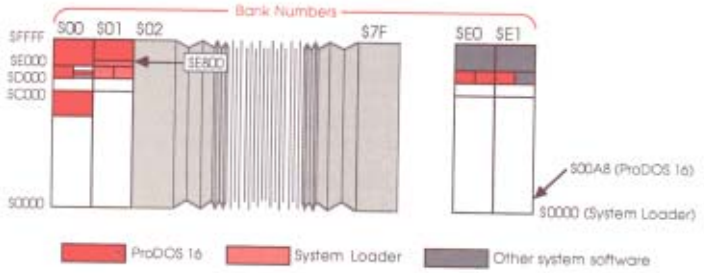


Figure 3-2
ProDOS 16 and System Loader memory map

Entry points and fixed locations

Because most Apple IIGS memory blocks are movable and under the control of the Memory Manager (see next section), there are very few fixed entry points available to applications programmers. References to fixed entry points in RAM are strongly discouraged, since they are inconsistent with flexible memory management and are sure to cause compatibility problems in future versions of the Apple IIGS. Informational system calls and referencing by handles (see "Pointers and Handles" in this chapter) should take the place of access to fixed entry points.

The single supported System Loader entry point is $E1 0000. That location is the entry point for all Apple IIGS tool calls.

The single supported ProDOS 16 entry point is $E1 00A8. That location is the entry point for all ProDOS 16 calls. In addition, ProDOS 16 supports a few other fixed locations in its bank $E1 vector space. Table 3-2 lists them.

Table 3-2
ProDOS 16 fixed locations

Address range	Explanation
$E1 00A8 – $E1 00AB	Entry vector for all ProDOS 16 system calls
$E1 00AC – $E1 00B9	(reserved)
$E1 00BA – $E1 00BB	Two null bytes (guaranteed to be zeros)
$E1 00BC	OS_KIND byte—indicates the currently running operating system: $00 = ProDOS 8 $01 = ProDOS 16
$E1 00BD	OS_BOOT byte—indicates the operating system that was initially booted: $00 = ProDOS 8 $01 = ProDOS 16
$E1 00BE – $E1 00BF	Flag word. The bits are defined as follows: bit 15 (ProDOS busy flag): 0 = ProDOS 16 is not busy 1 = ProDOS 16 is busy Bits 14 – 0: (reserved)

The ProDOS busy flag is explained under "Making Operating System Calls During Interrupts," in Chapter 7.

❖ *Note:* ProDOS 16 does *not* support the ProDOS 8 global page or any other fixed locations used by ProDOS 8.

Memory management

ProDOS 16 itself does no memory management. All allocation and deallocation of memory in the Apple IIGS is performed by the **Memory Manager**. The Memory Manager is an Apple IIGS tool set; for a complete description of its functions, see *Apple IIGS Toolbox Reference*.

The Memory Manager

The Memory Manager is a ROM-resident Apple IIGS tool set that controls the allocation, deallocation, and repositioning of memory blocks in the Apple IIGS. It works closely with ProDOS 16 and the System Loader to provide the needed memory spaces for loading programs and data and for providing buffers for input/output. All Apple IIGS software, including the System Loader and ProDOS 16, must obtain needed memory space by making requests (calls) to the Memory Manager.

The Memory Manager keeps track of how much memory is free and what parts are allocated to whom. Memory is allocated in **blocks** of arbitrary length; each block possesses several attributes that describe how the Memory Manager may modify it (such as moving it or deleting it), and how it must be aligned in memory (for example, on a page boundary). Table 3-3 lists the Memory Manager attributes that a memory block has.

Table 3-3
Memory block attributes

Attribute	Explanation
fixed (yes/no)	Must the block remain at the same location in memory?
fixed address (yes/no)	Must it be at a specific address?
fixed bank (yes/no)	Must it be in a particular memory bank?
bank-boundary limited (yes/no)	It is prohibited from extending across a bank boundary?
special memory not usable (yes/no)	Is it prohibited from residing in special memory (banks $00, $01, and parts of banks $E0, $E1)?
page-aligned (yes/no)	Must it be aligned to a page boundary?
purge level (0 to 3)	Can it be purged? If so, with what priority?
locked (yes/no)	Is the block locked (temporarily fixed and unpurgeable)?

Each block is also defined by it's **User ID**, a code number that shows what program owns it.

Besides creating and deleting memory blocks, the Memory Manager moves blocks when necessary to consolidate free memory. When it **compacts** memory in this way, it of course can move only those blocks that needn't be fixed in location. Therefore as many memory blocks as possible should be movable (not fixed), if the Memory Manager is to be efficient in compaction.

When a memory block is no longer needed, the memory Manager either **purges** it (deletes its contents but maintains its existence) or **disposes** it (completely removes it from memory).

Pointers and handles

To access an entry point in a movable block, an application cannot use a simple pointer, since the Memory Manager may move the block and change the entry point's address. Instead, each time the Memory Manager allocates a memory block, it returns to the requesting application a **handle** referencing that block.

A handle is a pointer to a pointer; it is the address of a fixed (nonmovable) location, called the **master pointer**, that contains the address of the block. If the Memory Manager changes the location of the block, it updates the address in the master pointer; the value of the handle itself is not changed. Thus the application can continue to access the block using the handle, no matter how often the block is moved in memory. Figure 3-3 illustrates the difference between a pointer and a handle.

If a block will always be fixed in memory (**locked** or **unmovable**), it can be referenced by a pointer instead of by its handle. To obtain a pointer to a particular block or location, an application can **dereference** the block's handle. The application reads the address stored in the location pointed to by the handle—that address is the pointer to the block. Of course, if the block is ever moved that pointer is no longer valid.

ProDOS 16 and the System Loader use both pointers and handles to reference memory locations. Pointers and handles must be at least three bytes long to access the full range of Apple IIGS memory. However, all pointers and handles used as parameters by ProDOS 16 are four bytes long, for ease of manipulation in the 16-bit registers of the 65C816 microprocessor.

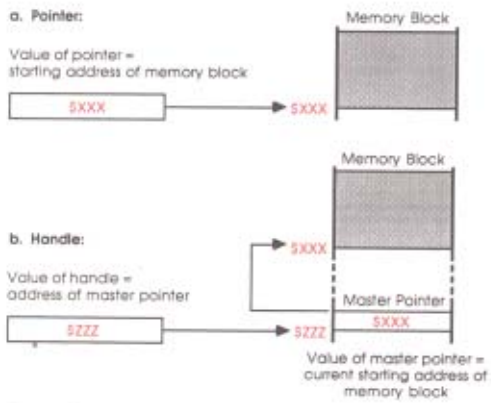


Figure 3-3
Pointers and handles

How an application obtains memory

Normal memory allocation and deallocation is completely automatic, as far as applications are concerned. When an application makes a ProDOS 16 call that requires allocation of memory (such as opening a file or writing from a file to a memory location), ProDOS 16 first obtains any needed memory blocks from the Memory Manager and then performs its tasks. Likewise, the System Loader requests any needed memory either directly or indirectly (through ProDOS 16 calls) from the Memory Manager. Conversely, when an application informs the operating system that it no longer needs memory, that information is passed on to the Memory Manager which in turn frees that application's allocated memory.

Any other memory that an application needs for its own purposes must be requested directly from the Memory Manager. The shaded areas in Figure 3-3 show which parts of the Apple IIGS memory can be allocated through requests to the Memory Manager. Applications for Apple IIGS should avoid requesting absolute (fixed-address) blocks. Chapters 6 and 16 of this manual discuss program memory management further; see also *Programmer's Introduction to the Apple IIGS* and *Apple IIGS Toolbox Reference*.

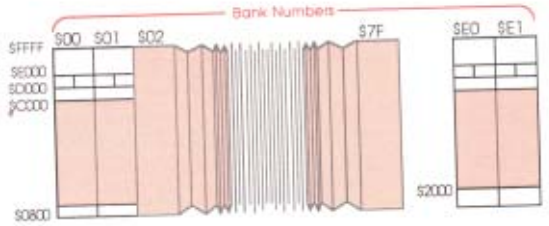


Figure 3-4
Memory allocatable through the Memory Manager

Chapter 4

ProDOS 16 and External Devices

An **external device** is a piece of equipment that transfers information to or from the Apple IIGS. Disk drives, printers, mice, and joysticks are external devices. The keyboard and screen are also considered external devices. An *input* device transfers information *to* the computer, an *output* device transfers information *from* the computer, and an *input/output* device transfers information both ways.

This chapter discusses how ProDOS 16 provides an interface between applications and certain external devices.

Block devices

A **block device** reads and writes information in multiples of one block of characters (512 bytes; see Table 3-1) at a time. Furthermore, it is a *random-access* device—it can access any block on demand, without having to scan through the preceding or succeeding blocks. Block devices are usually used for storage and retrieval of information, and are usually input/output devices. Disk drives are block devices.

ProDOS 16 supports access to block devices. That is, you may read from or write to a block device by making ProDOS 16 calls. In addition to `READ`, `WRITE`, and the other file calls described in Chapter 2, ProDOS 16 also provides five "lower-level" device-access calls. These calls allow you to access information on a block device without considering what files the information is in. The calls are

`GET_DEV_NUM`	returns the device number associated with a particular named device or online volume
`GET_LAST_DEV`	returns the device number of the last device accessed through ProDOS 16
`READ_BLOCK`	reads one block (512 bytes) of data from a specified device
`WRITE_BLOCK`	writes one block (512 bytes) of data to a specified device
`FORMAT`	formats (initializes) a volume in a device

A block device generally requires a **device driver** to translate ProDOS 16's *logical* block device model into the **tracks** and **sectors** by which information is actually stored on the *physical* device. The device driver may be circuitry within the disk drive itself (UniDisk™ 3.5), it may be included as part of ProDOS 16 (Disk II®), or it may be on a separate card in an expansion slot. This manual does not discuss device drivers.

❖ *Note on RAM disks:* RAM disks are internal software constructs that the operating system treats like external devices. Although ProDOS 16 provides no particular support for RAM disks, any RAM disk that behaves like a block device in all respects will be supported just as if it were an external device.

Character devices

A **character device** reads or writes a stream of characters in order, one at a time. It is a **sequential-access** device—it cannot access any position in a stream without first accessing all previous positions. It can neither skip ahead nor go back to a previous character. Character devices are usually used to pass information to and from a user or another computer; some are input devices, some are output devices, and some are input/output devices. The keyboard, screen, printer and communications port are character devices.

Current versions of ProDOS 16 do not support character devices; that is, you cannot access character devices through ProDOS 16 calls. Consult the appropriate firmware or tools documentation, such as *Apple IIGS Firmware Reference* or *Apple IIGS Toolbox Reference*, for instructions on how to make calls to the particular device you wish to use.

Accessing devices

Under ProDOS 16, you can access block devices through their device numbers, device names, or the volume names of the volumes mounted on them.

Named devices

ProDOS 16 permits block devices to have assigned names. This ability is a convenience for users, because they will no longer have to know the volume name to access a disk.

However, ProDOS 16's support for named devices is limited. Device names may be used only in the VOLUME, GET_DEV_NUM, and FORMAT calls. Other calls that access devices require either a volume name or the device number returned by the GET_DEV_NUM or GET_LAST_DEV call.

Devices are named according to a built-in convention; assigned names may not be changed. The naming convention is as follows:

Device	Name
Any block device	.D*n*
where	*n*= a 1-digit or 2-digit decimal number (assigned consecutively)

Last device accessed

An application may ask ProDOS 16 for the identity of the last block device accessed. The *last device accessed* is defined here as the device to which the most recent call involving a disk read or write (including a block read or write) was directed.

When an application makes the GET_LAST_DEV call, ProDOS 16 returns the device number of the last block device accessed. The application can then use that information as input to subsequent device calls.

Block read and block write

ProDOS 16 provides two device-access calls analogous to the file-access calls READ and WRITE. These calls, READ_BLOCK and WRITE_BLOCK, allow you to transfer information to and from a volume on a block device regardless of what files the volume contains.

The device number of a device (returned by GET_DEV_NUM) is a required input for the block read and write calls. The block read and write calls are powerful, but are not needed by most applications—the filing calls described in Chapter 2 are sufficient for normal disk I/O.

Formatting a disk

Your application can format (initialize) a disk in a device through the ProDOS 16 FORMAT call. The call requires both a device name and a volume name as input. The disk in the specified device is formatted and given the specified volume name.

The other required input to the FORMAT call is the file system ID. It specifies the class of operating system for which the disk is to be formatted (such as DOS, ProDOS, or Pascal). Under current versions of ProDOS 16, however, the FORMAT call can format disks for the ProDOS/SOS file system only (file system ID = 1).

Number of online devices

ProDOS 16 supports up to 14 active devices at a time. The Apple IIGS normally accepts up to 4 devices connected to its disk port (Smartport) and two devices per expansion slot (slots 1 through 7). It is possible, however, to have up to 4 devices on (a Smartport card in) slot 5. Nevertheless, the total number of devices on line still cannot exceed 14.

Device search at startup

When ProDOS 16 boots, it performs a device search to identify all built-in pseudo-slot ROMs (internal ROMs) and all real physical slot ROMs (card ROMs). Every block device found is incorporated into ProDOS 16's list of devices, and assigned a device number (*dev_num*) and device name (*dev_name*).

❖ *Note:* Control Panel settings determine whether internal ROM or card ROM is active for each slot. ProDOS 16 cannot simultaneously support both internal and external devices with the same slot number.

In general, the device search proceeds from highest-numbered slots downward. For example, a disk drive in slot 7 drive 1 will be device number 1; another drive in slot 7 drive 2 will then be device 2, and on downward through all the slots.

SmartPort (slot 5's internal ROM and diskport) is a special case. Up to 4 devices may be connected to SmartPort. However, because ProDOS 16 supports only 2 devices per slot, the third and fourth devices are treated as if they were in slot 2. Despite the mapping of devices 3 and 4 into slot 2, however, all devices connected to SmartPort are given consecutive numbers. Table 4-1 shown the relationships.

Table 4-1
SmartPort number, slot number, and device number assignments

SmartPort no.†	slot and drive	device number
1	slot 5 drive 1	*n*
2	slot 5 drive 2	*n*+1
3	slot 2 drive 1	*n*+2
4	slot 2 drive 2	*n*+3

† SmartPort device number 1 is connected directly to SmartPort. Subsequent devices are conected in daisy-chain fashion to the preceding ones, so that device number 4 is the farthest from SmartPort.

Apple Disk II and other related 5.25-inch disk drives are another special case. Because of the relatively long time required to access a Disk II drive and to determine whether a disk is present in it, Disk II drives are given the highest device numbers on the system. That way they will be searched last in any scan of online devices.

Volume control blocks

For each device with nonremovable media (such as a hard disk) found at boot time, a **volume control block** (VCB) is created in memory. The VCB keeps track of the characteristics of that online volume. For other devices (such as floppy disk drives) found at boot time, VCB's are created as files are opened on the volumes in those devices. A maximum of eight VCB's may exist at any one time; if you try to open a file on a device whose volume presently has no open files, and if there are already eight VCB entries, error $55 (VCB table full) is returned. Thus, even though there may be up to 14 devices connected to your system, only eight (at most) can be active (have open files) at any one moment.

Interrupt handling

On the Apple IIGS, interrupts may be handled at either the firmware or the software level. The built-in interrupt handers are in firmware (see *Apple IIGS Firmware Reference*); user-installed interrupt handlers are software and may be installed through ProDOS 16.

When the Apple IIGS detects an interrupt that is to be handled through ProDOS 16, it dispatches execution through the interrupt vector at $00 03FE (page 3 in bank zero). At this point the microprocessor is running in emulation mode, using the standard clock speed and 8-bit registers. The vector at $00 03FE has only two address bytes; in order to allow access to all of Apple IIGS memory, it points to another bank zero location. The vector in that location then passes control to the ProDOS 16 interrupt dispatcher. The interrupt dispatcher switches the processor to full native mode (including higher clock speed) and then polls the user-installed interrupt handlers.

Figure 4-1 is a simplified picture of what happens when a device generates an interrupt that is handled through a ProDOS 16 interrupt handler.

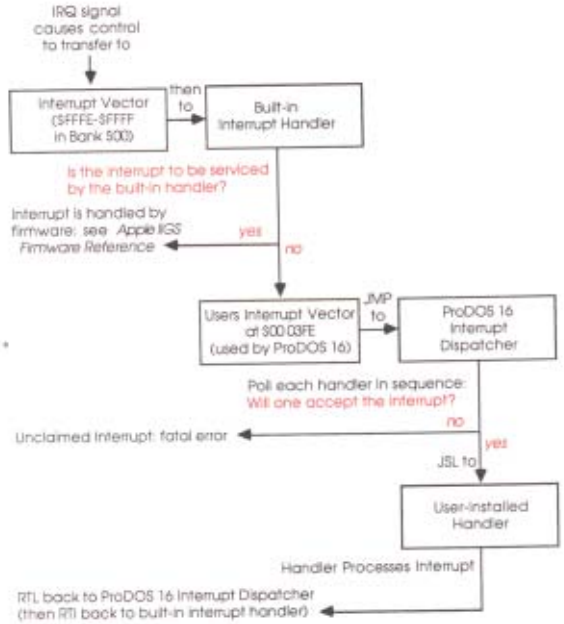


Figure 4-1
Interrupt handling through ProDOS 16

ProDOS 16 supports up to 16 user-installed interrupt handlers. When an interrupt occurs that is not handled by firmware, ProDOS 16 transfers control to each handler successively until one of them claims it. There is no grouping of interrupts into classes; their priority rankings are reflected only by the order in which they are polled.

If you write an interrupt-handling routine, to make it active you must install it with the `ALLOC_INTERRUPT` call; to remove it, you must use the `DEALLOC_INTERRUPT` call. Be sure to enable the hardware generating the interrupt only *after* the routine to handle it is allocated; likewise, disable the hardware *before* the routine is deallocated. See Chapter 7 for further details on writing and installing interrupt handlers.

Unclaimed interrupts

An *unclaimed interrupt* is defined as the condition in which the hardware Interrupt Request Line (IRQ) is active (being pulled low), indicating that an interrupt-producing device needs attention, but none of the installed interrupt handlers claims responsibility for the interrupt. When an interrupt occurs and ProDOS 16 can find no handler to claim it, it assumes that a serious hardware error has occurred. It issues a fatal error message to the System Failure Manager (see *Apple IIGS Toolbox Reference*), and stops processing the current application. Processing cannot resume until the user reboots the system

Chapter 5

ProDOS 16 and the Operating Environment

ProDOS 16 is one of the many components that make up the Apple IIGS **operating environment**, the overall hardware and software setting within which Apple IIGS application programs run. This chapter describes how ProDOS 16 functions in that environment and how it relates to the other components.

Apple IIGS system disks

An Apple IIGS **system disk** is a disk containing the system software needed to run any application you wish to execute. Most system disks contain one or both operating systems (ProDOS 16 and ProDOS 8), the System Loader, RAM-based tool sets, RAM patches to ROM-based tool sets, fonts, desk accessories, boot-time initialization programs, and possibly one or more applications.

There are two basic types of system disks: *complete* system disks and *application* system disks. A complete system disk has a full set of Apple IIGS system software, as listed in table 5-1. It is a resource pool from which application system disks can be constructed. An application system disk has one or more application programs and only the specific system software it needs to run the application(s). For example, a word processor system disk may include a large selection of fonts, whereas a spreadsheet system disk may have only a few fonts.

Software developers may create application system disks for their programs. Users may also create application system disks, perhaps by combining several individual application disks into a multi-application system disk. Apart from the essential files listed in table 5-2, there is no single set of required contents for application system disks.

Complete system disk

Every Apple IIGS user (and developer) needs at least one complete system disk. It is a pool of system software resources, and may contain files missing from any of the available application system disks. Table 5-1 lists the contents of a complete system disk.

Table 5-1
Contents of a complete Apple IIGS system disk

Directory/File	Description
PRODOS	a routine that loads the proper operating system and selects an application, both at boot time and whenever an application quits
SYSTEM/	a subdirectory containing the following files:
P8	ProDOS 8 operating system
P16	ProDOS 16 operating system and Apple IIGS System Loader
START	typically a program selector
LIBS/	a subdirectory containing the standard system libraries
TOOLS/	a subdirectory containing all RAM-based tools
FONTS/	a subdirectory containing all fonts
DESK.ACCS/	a subdirectory containing all desk accessories
SYSTEM.SETUP/	a subdirectory containing system initialization programs
TOOL.SETUP	a load file containing patches to ROM and a program to install them. This is the only *required* file in the SYSTEM.SETUP/ subdirectory; it is executed before any others that may be in the subdirectory.
BASIC.SYSTEM	The Applesoft BASIC system interface program

The complete system disk is an 800K byte, double-sided 3.5-inch diskette; the required files will not fit on a 140K, single-sided 5.25-inch diskette.

When you boot a complete system disk, it executes the file SYSTEM/START. From the START file, you may choose to call Applesoft BASIC, the only application program available on the disk.

The SYSTEM.SETUP/ subdirectory

The SYSTEM.SETUP/ subdirectory may contain several different types of files, all of which need to be loaded and initialized at boot time. They include the following:

- **The file TOOL.SETUP:** This file must always be present; it is executed before any others in SYSTEM.SETUP/. TOOL.SETUP installs and initializes any RAM patches to ROM-based tool sets. After TOOL.SETUP is finished, ProDOS 16 loads and executes the remaining files in the SYSTEM.SETUP/ subdirectory, which may belong to any of the categories listed below.

- **Permanent initialization files (filetype $B6):** These files are loaded and executed just like standard applications (type $B3), but they are not shut down when finished. They also must have certain characteristics:
  1. They must be loaded in non-special memory.
  2. They cannot permanently allocate any stack/direct-page space.
  3. They must terminate with an RTL (Return from subroutine Long) rather than a QUIT.
- **Temporary initialization files (type $B7):** These files are loaded and executed just like standard applications (type $B3), and they are shut down when finished. They must terminate with an RTL rather than a QUIT.
- **New desk accessories (type $B8):** These files are loaded but not executed. They must be in non-special memory.
- **Classic desk accessories (type $B9):** These files are loaded but not executed. They must be in non-special memory.

Application system disks

Each application program or group of related programs comes on its own application system disk. The disk has all of the system files needed to run that application, but it may not have all the files present on a complete system disk. Different applications may have different system files on their application system disks.

For example, the *ProDOS 16 Exerciser* disk, included with this manual, is an application system disk. It contains all the system files listed above, plus the file EXERCISER (the exerciser itself).

Table 5-2 shows which files must be present on all application system disks, and which files are needed only for particular applications. In some very restricted instances, it may be possible to fit an application and its required system files onto a 5.25-inch (140K) diskette; most applications, however, require an 800K diskette.

Table 5-2
Required contents of an Apple IIgs application system disk

Directory/File	Required/(Required If...)
PRODOS	**required**
SYSTEM/	**required**
P8	(required if the application is ProDOS 8-based)
P16	**required**
START	(required if the program selector is to be used)
LIBS/	(required if system library routines are needed)
TOOLS/	(required if the application needs RAM-based tools)
FONTS/	(required if the application needs fonts)
DESK.ACCS/	(required if desk accessories are to be provided)
SYSTEM.SETUP/	**required**
TOOL.SETUP	**required**
BASIC.SYSTEM	(required if the application is written in Applesoft BASIC)

Important The files PRODOS, P8 and P16 all have version numbers. Whenever it loads an operating system (at startup or when launching an application), PRODOS checks the P8 or P16 version number against its own. If they do not match, it is a fatal error. Be careful not to construct an application system disk using incompatible versions of PRODOS, P8 and P16.

System startup

Disk blocks 0 and 1 on an Apple IIGS system disk contain the startup (boot) code. They are identical to the boot blocks on standard Apple II system disks (ProDOS 8 system disks). This allows ProDOS 8 system disks to boot on an Apple IIGS, and it also means that the initial part of the ProDOS 16 bootstrap procedure is identical to that for ProDOS 8.

Boot initialization

Figure 5-1 shows the boot initialization procedure. First, the boot firmware in ROM reads the boot code (blocks 0 and 1) into memory and executes it. For a system disk with a volume name /V,

1. The boot code searches the disk's volume directory for the first file named /V/PRODOS with the file type $FF.
2. If the file is found, it is loaded and executed at location $2000 of bank $00.

From this point on, an Apple IIGS system disk behaves differently from a standard Apple II system disk. On a standard Apple II system disk, the file named PRODOS is the ProDOS 8 operating system. On an Apple IIGS system disk, however, this PRODOS file is not the operating system itself; it is an operating system loader and application selector. When it receives control from the boot code, /V/PRODOS performs the following tasks (see also Figure 5-1):

3. It relocates the part of itself named PQUIT to an area in memory where PQUIT will reside permanently. PQUIT contains the code required to terminate one program and start another (either ProDOS 8 or ProDOS 16 application).
4. /V/PRODOS loads the ProDOS 16 operating system and Apple IIGS System Loader (file /V/SYSTEM/P16).
5. /V/PRODOS performs any necessary boot initialization of the system, by executing the files in the subdirectory /V/SYSTEM/SYSTEM.SETUP/. If there is a file named TOOL.SETUP in that subdirectory, it is executed first—it loads RAM-based tools and RAM patches to ROM-based tools.

   Every file in the subdirectory /V/SYSTEM/SYSTEM.SETUP/ must be an Apple IIGS load file of type $B6, $B7, $B8, or $B9. These file types are described under "The SYSTEM.SETUP/ Subdirectory," earlier in this chapter. After executing TOOL.SETUP, /V/PRODOS loads and executes, in turn, every other file that it finds in the subdirectory.

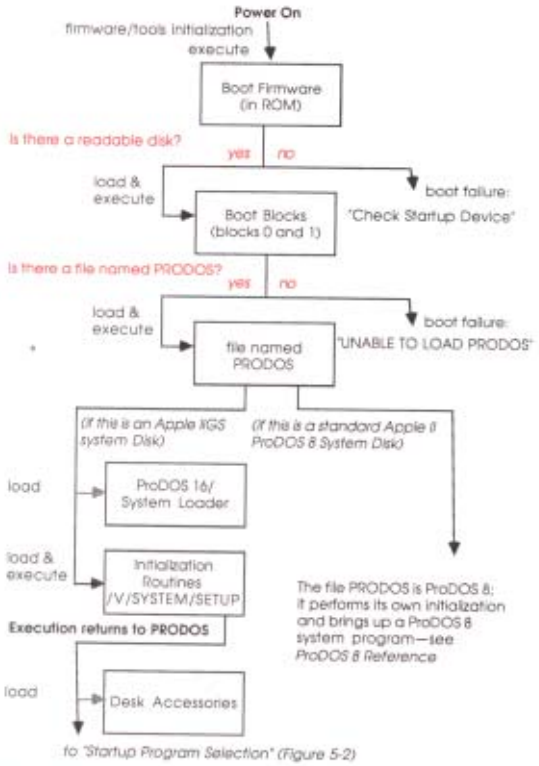


Figure 5-1
Boot initialization sequence

Startup program selection

6. Now `/V/PRODOS` selects (determines the pathname of) the system program or application to run. Figure 5-2 shows this procedure.

   a. It first searches for a type $B3 file named `/V/SYSTEM/START`. Typically, that file is a program selector, but it could be any Apple IIGS application. If `START` is found, it is selected.

   b. If there is no `START` file, `/V/PRODOS` searches the boot volume directory for a file that is either one of the following:

      - a ProDOS 8 system program (type $FF) with the filename extension `.SYSTEM`
      - a ProDOS 16 application (type $B3) with the filename extension `.SYS16`

      Whichever is found first is selected.

   ❖ *Note:* If a ProDOS 8 system program is found first, but the ProDOS 8 operating system (file `/V/SYSTEM/P8`) is not on the system disk, `/V/PRODOS` will then search for and select the first ProDOS 16 application (ProDOS 16 is *always* on the system disk).

   c. If `/V/PRODOS` cannot find a file to execute (for example, if there is no `START` file and there are no ProDOS 8 or ProDOS 16 applications), it will bring up an interactive routine that prompts the user for the filename of an application to load.

7. Finally, `/V/PRODOS` passes control to an entry point in `PQUIT`. It is `PQUIT`, not `/V/PRODOS`, that actually loads the selected program. The next section describes that procedure.

❖ *Note:* `PRODOS` will write an error messsage to the screen if you try to boot it on an Apple II computer other than an Apple IIGS. This is because ProDOS 8 on an Apple IIGS disk is in the file `V/SYSTEM/P8`, not in the file `PRODOS`.

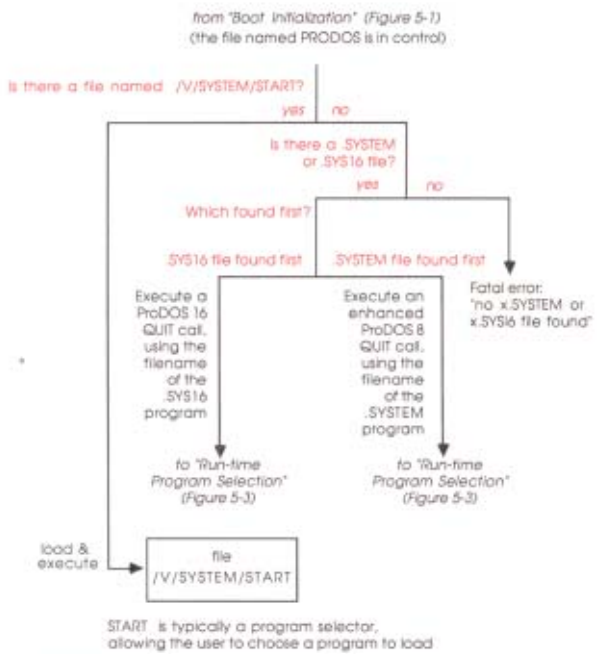


Figure 5-2
Startup program selection

Starting and quitting applications

The Apple IIGS startup sequence ends when control is passed to the program selection routine (PQUIT). This routine is entered both at boot time and whenever an application terminates with a ProDOS 16 or ProDOS 8 QUIT call.

PQUIT

PQUIT is the ProDOS program dispatcher for the Apple IIGS. It determines which ProDOS 8 or ProDOS 16 program is to be run next, and runs it. After startup, PQUIT is permanently resident in memory; PQUIT loads ProDOS 16 programs through calls to the System Loader.

PQUIT has two entry points: P8PQUIT and P16PQUIT. Whenever a ProDOS 8 application executes a QUIT call, control passes through the P8PQUIT entry point. Whenever a ProDOS 16 application executes a QUIT call, control passes through the P16PQUIT entry point. To launch the first program at system startup, /V/PRODOS passes control to PQUIT as if executing a ProDOS 8 or ProDOS 16 QUIT call.

PQUIT supports three types of quit call: the standard ProDOS 8 QUIT call, an enhanced ProDOS 8 QUIT call, and the ProDOS 16 QUIT call.

Standard ProDOS 8 QUIT call

The standard ProDOS 8 QUIT call's parameter block consists of a one-byte parameter count field (which must have the value $04), followed by four null fields in this order: byte, word, byte, word. As ProDOS 8 is currently defined, all fields must be present and all must be set to zero. There is thus no way for a program to use the standard QUIT call to specify the pathname of the next program to run.

Enhanced ProDOS 8 QUIT call

The enhanced ProDOS 8 QUIT call differs from the standard call only in the permissible values of the first two parameters (its parameter count field must still have the value $04). In the enhanced QUIT call, the first (byte) parameter is defined as the *quit type*. If it is zero, the call is identical to a standard QUIT call; if it is $EE, PQUIT interprets the following (word) parameter as a pointer to a string which is the pathname of the next program to run.

The enhanced ProDOS 8 QUIT call is meaningful only on the Apple IIGS, and only when PQUIT is present to interpret it (that is, only when the computer has been booted with an Apple IIGS system disk). It behaves like the standard ProDOS 8 QUIT call in any other situation.

❖ *Note:* Because of the way ProDOS uses memory, a ProDOS 8 application must not make an enhanced QUIT call (with a quit type of $EE) from any location in page 2 of bank $00 (addresses $00 02 00 – $00 02 FF).

ProDOS 16 QUIT call

The ProDOS 16 QUIT call has two parameters: a pointer to the pathname of the next program to execute, and a pair of boolean flags: one (the *return flag*) notifies PQUIT whether or not control should eventually return to the program making the QUIT call; the other one (the *restart-from-memory flag*) lets the System Loader know whether the quitting program can be restarted from memory when it returns.

If the value of the return flag is true, PQUIT pushes the User ID of the calling (quitting) program onto an internal stack. As subsequent programs run and quit, several User ID's may be pushed onto the stack. With this mechanism, multiple levels of shells may execute subprograms and subshells, while ensuring that they eventually regain control when their subprograms quit.

For example, the program selector (START file) might pass control to a software development system shell, using the QUIT call to specify the shell and placing its own ID on the stack. The shell in turn could hand control to a debugger, likewise puting its own ID on the stack. If the debugger quits without specifying a pathname, control would pass automatically back to the shell; when the shell quit, control would pass automatically back to the START file.

This automatic return mechanism is specific to the ProDOS 16 QUIT call, and therefore is not available to ProDOS 8 programs. When a ProDOS 8 application quits, it cannot put its ID on the internal stack.

QUIT procedure

This is a brief description of how PQUIT handles all three types of QUIT call. Refer also to Figure 5-3.

1. If a ProDOS 16 or enhanced ProDOS 8 QUIT call specifies a pathname, PQUIT attempts to execute the specified file. Under certain conditions this may not be possible: the file may not exist, there may be insufficient memory , and so on. In that case the QUIT call executes the interactive routine described below (step 3).

❖ *Note:* PQUIT will load programs of file type $B3, $B5, or $FF *only*.

2. If the QUIT call specifies no pathname, PQUIT pulls a User ID off its internal ID stack and attempts to execute that program. Typically, programs with User ID's on the stack are in the System Loader's **dormant** state (see "User Shutdown" in Chapter 17), and it may be possible to restart them without reloading them from disk. Under certain conditions it may not be possible to execute the program: the file may not exist, there may be insufficient memory , and so on. In that case the QUIT call executes the interactive routine described next (step 3).
3. If the QUIT call specifies no pathname and the ID stack is empty, PQUIT executes an interactive routine that allows the user to do any of these:
   - □ reboot the system
   - □ execute the file /V/SYSTEM/START
   - □ enter the pathname of a program to execute
4. If the quitting program is a ProDOS 16 program, PQUIT calls the loader's User Shutdown routine to place that program in a dormant state.
5. Once it has determined which program to load, PQUIT knows which operating system is required. If it is not the *current* system,
   a. PQUIT shuts down the current operating system and loads the required one.
   b. PQUIT then makes Memory Manager calls to free memory used by the former operating system and allocate memory needed by the new system. If the new operating system is ProDOS 8, PQUIT allocates all special memory for the program.

6. The new program is loaded. PQUIT calls the System Loader to load ProDOS 16 programs; for ProDOS 8 programs, PQUIT passes control to ProDOS 8, which then loads and executes its own program directly.
7. Finally (if it is a ProDOS 16 program), PQUIT sets up various aspects of the program's environment, including the direct-register and stack-pointer values, and passes control to the program.

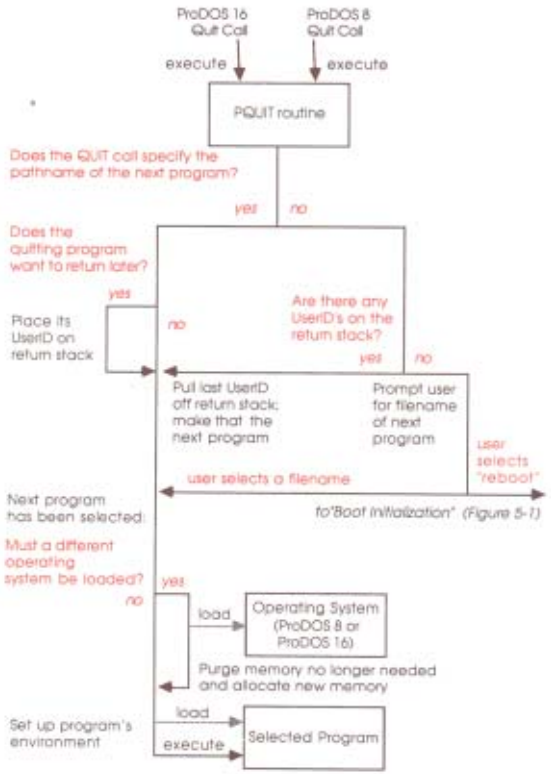


Figure 5-3
Run-time program selection (QUIT call)

Machine configuration at application launch

PQUIT initializes certain hardware and software components of the Apple IIGS before it passes control to a program. There are many other factors the machine's state that are not considered here, such as memory used by other software and the state of the dozens of soft switches and pseudoregisters available on the Apple IIGS. This section summarizes only the aspects of machine state that are explicitly set by ProDOS 16.

- **Reserved bank $00 space:**

  Addresses above approximately $9600 in bank zero are reserved for ProDOS 16, and therefore unavailable to the application. A direct-page/stack space, of a size determined either by ProDOS 16 or by the application itself, is reserved for the application (see Chapter 6); it is located in bank $00 at an address determined by the Memory Manager. ProDOS 16 requires no other space in RAM (other that the language-card areas in banks $00, $01, $E0, and $E1—see Figure 3-2).

- **Hardware registers:**

  The accumulator contains the User ID assigned to the application.
  The X- and Y-registers contain zero ($0000).
  The e-, m-, and x-flags in the processor status register are all set to zero, meaning that the processor is in *full native mode*.
  The stack register contains the address of the top of the direct-page/stack space (see Chapter 6).
  The direct register contains the address of the bottom of the direct-page/stack space (see Chapter 6).

- **Standard input/output:**

  For both $B3 and $B5 files, the standard input, output, and error locations are set to the Pascal 80-column character device vectors. See "Text Tool Set" in *Apple IIGS Toolbox Reference*.

- **Shadowing:**

  The value of the Shadow register is $1E, which means:

language card and I/O spaces:	shadowing ON
text pages:	shadowing ON
graphics pages:	shadowing OFF

- **Vector space values:**

  Addresses between $00A8 and $00BF in bank $E1 constitute ProDOS 16's *vector space*—so named because it contains the entry point (vector) to all ProDOS 16 calls. It also contains other information useful to system software such as AppleTalk®. The specific values an application finds in the vector space are listed in Table 3-2. These are the only fixed locations supported by ProDOS 16.

- **Pathname prefix values:**

  The nine available pathname prefixes are set as described in the next section.

Pathname prefixes

A pathname **prefix** is a part of a pathname that starts with a volume name and ends with the name of a subdirectory. A preassigned prefix is convenient when many files in the same subdirectory are accessed, because it shortens the pathname references. A *set* of prefixes is convenient when files in several different subdirectories must be repeatedly accessed. The System Loader, for example, makes use of multiple prefixes. Once the pathname prefixes are assigned, an application can refer to the prefixes by code instead of keeping track of all the different pathnames.

ProDOS 16 supports 9 prefixes, referred to by the **prefix numbers** `0/`, `1/`, `2/`,...,`7/`, and `*/`. Each prefix number includes a terminating slash to separate it from the rest of the pathname. A prefix number at the beginning of a partial pathname replaces the actual prefix. One of the prefix numbers has a fixed value, and the others have default values assigned by ProDOS 16 (see Table 5-4). The most important predefined prefixes are

`*/` the boot prefix—it is the name of the volume from which the presently running ProDOS 16 was booted.

`0/` the default prefix (automatically attached to any partial pathname that has no prefix number)—it has a value dependent on how the current program was launched. In some cases it is equal to the boot prefix.

`1/` the application prefix—it is the pathname of the subdirectory that contains the currently running application.

`2/` the system library prefix—it is the pathname of the subdirectory (on the boot volume) that contains the library files used by applications.

Your application may assign the rest of the prefixes. In fact, once your application is running, it may also change the values of prefixes `0/`, `1/`, or `2/` (applications may not change prefix `*/`).

Prefix `0/` is similar to the ProDOS 8 *system prefix*, in that ProDOS 16 automatically attaches prefix `0/` to any partial pathname for which you specify no prefix. However, its initial value is not always equivalent to the ProDOS 8 system prefix's initial value. See *ProDOS 8 Technical Reference Manual*.

The prefix numbers are set (assigned to specific pathnames) and retrieved through the `SET_PREFIX` and `GET_PREFIX` calls. Although a prefix number may be used as an input to the `SET_PREFIX` call, prefixes are always stored in memory as full pathnames (that is, they include no prefix numbers themselves).

Table 5-3 shows some examples of prefix use. They assume that prefix `0/` is set to `/VOLUME1/` and that prefix `5/` is set to `/VOLUME1/TEXT.FILES/`. The pathname provided by the caller is compared with the full pathname constructed by ProDOS 16.

Table 5-3
Examples of prefix use

	as supplied	as expanded by ProDos 16
Full pathname provided:	`/VOLUME1/TEXT.FILES/CHAP.3`	`/VOLUME1/TEXT.FILES/CHAP.3`
Partial pathname—implicit use of prefix /0:	`PRODOS`	`/VOLUME1/PRODOS`
Explicit use of prefix /0:	`0/SYSTEM/FINDER`	`/VOLUME1/SYSTEM/FINDER`
Use of prefix 5/:	`5/CHAP.12`	`/VOLUME1/TEXT.FILES/CHAP.12`

Initial ProDOS 16 prefix values

When an application is launched, all nine prefix numbers are assigned to specific pathnames (some are meaningful pathnames, whereas others may be null strings). Remember, an application may change the assignment of any prefix number except the boot prefix (*/). Furthermore, in some cases certain initial prefix values may be left over from the previous application. Therefore, beware of assuming a value for any particular prefix.

Table 5-4 shows the initial values of the prefix numbers that a ProDOS 16 application receives, under the three different launching conditions possible on the Apple IIGS. At all times during execution, GET_NAME returns the filename of the current application (regardless of whether prefix 1/ has been changed), and GET_BOOT_VOL returns the boot volume name, equal to the value of prefix */ (regardless of whether prefix 0/ has been changed).

Table 5-4
Initial ProDOS 16 prefix values

	Prefix no.	Initial value
ProDOS 16 application launched at boot time:	0/	boot volume name
	1/	full pathname of the directory containing the current application
	2/	full pathname of the application library directory (*/boot volume name*/SYSTEM/LIBS)
	3/	null string
	4/	null string
	5/	null string
	6/	null string
	7/	null string
	*/	boot volume name

Table 5-4 (continued)
Initial ProDOS 16 prefix values

	Prefix no.	Initial value
ProDOS 16 application launched after a ProDOS 8 application has quit:	0/	unchanged from the ProDOS 8 *system prefix* under the previous application
	1/	full pathname of the directory containing the current application
	2/	full pathname of the application library directory (*/boot volume name*/SYSTEM/LIBS)
	3/	null string
	4/	null string
	5/	null string
	6/	null string
	7/	null string
ProDOS 16 application launched after a ProDOS 16 application has quit:	0/	unchanged from the previous application
	1/	full pathname of the directory containing the current application
	2/	unchanged from the previous application
	3/	unchanged from the previous application
	4/	unchanged from the previous application
	5/	unchanged from the previous application
	6/	unchanged from the previous application
	7/	unchanged from the previous application
	*/	unchanged from the previous application

ProDOS 8 prefix and pathname convention

ProDOS 8 supports a single prefix, called the *system prefix* (or *current prefix*). It has no prefix number—it is attached automatically to any partial pathname (one that does not begin with a slash and a volume name). Like the ProDOS 16 prefixes, the ProDOS 8 system prefix may be changed by a SET_PREFIX call. On a standard Apple II, the default value of the system prefix at startup is the boot volume name; however, system programs such as the Applesoft BASIC interpreter commonly reset the system prefix to other values.

An application that is running under ProDOS 8 can always find its own pathname by looking at location $0280 (in bank $00 on an Apple IIGS); ProDOS 8 stores the application's full or partial pathname there. For details of this and other ProDOS 8 pathname conventions, see *ProDOS 8 Technical Reference Manual*.

On the Apple IIGS, the PQUIT routine allows a ProDOS 8 application to be launched at boot time, or after another ProDOS 8 application has quit, or after a ProDOS 16 application has quit. The initial values of the system prefix and the pathname at location $0280 are dependent on which way the application was launched. Table 5-5 lists the possibilities.

Table 5-5
Initial ProDOS 8 prefix and pathname values

	system prefix	location $0280 pathname
ProDOS 8 application launched at boot time:	boot volume name	filename of the just-launched application
ProDOS 8 application launched through an enhanced ProDOS 8 QUIT call:	unchanged from the previous (ProDOS 8) application	the full or partial pathname given in the enhanced ProDOS 8 QUIT call
ProDOS 8 application launched through a ProDOS 16 QUIT call: (If the ProDOS 16 QUIT call specified a *full* pathname)	the previous (ProDOS 16) application's prefix 0/	the full pathname given in the ProDOS 16 QUIT call
ProDOS 8 application launched through a ProDOS 16 QUIT call: (If the ProDOS 16 QUIT call specified a *partial* pathname)	the prefix specified in the ProDOS 16 QUIT call	the partial pathname (minus the prefix number) given in the ProDOS 16 QUIT call

❖ *Note:* Conditions (2) through (3b) in Table 5-4 apply only to ProDOS 8 applications launched from an Apple IIGS *booted on an Apple IIGS system disk*. If a ProDOS 8 application on a *standard Apple II* system disk is booted on an Apple IIGS, the Apple IIGS acts like a standard Apple II and condition (1) is the only possibility.

Tools, firmware, and system software

Although ProDOS 16 is the principal part of the Apple IIGS operating system, several "operating system-like" functions are actually carried out by other software components. This section briefly describes some of those components; for detailed information see the references listed with each one.

The Memory Manager

As explained in Chapter 3, the Memory Manager takes care of all memory allocation, deallocation, and housekeeping chores. Applications obtain needed memory space either directly, through requests to the Memory Manager, or indirectly through ProDOS 16 or System Loader calls (which in turn obtain the memory through requests to the Memory Manager).

The Memory Manager is a ROM-resident Apple IIGS tool set; for more detailed information on its functions and how to call them, see *Apple IIGS Toolbox Reference*.

The System Loader

The System Loader is an Apple IIGS tool set that works very closely with ProDOS 16 and the Memory Manager. It resides on the system disk, along with ProDOS 16 and other system software (see "Apple IIGS System Disks" in this chapter). All programs and data are loaded into memory by the System Loader.

The System Loader supports both static and dynamic loading of segmented programs and subroutine libraries. It loads files that conform to a specific format (**object module format**); such files are produced by the **APW Linker** and other components of the **Apple IIGS Programmer's Workshop** (see *Apple IIGS Programmer's Workshop Reference*).

The System Loader is described in Part III of this manual.

The Scheduler

The **Scheduler** is a tool set that functions in conjunction with the Apple IIGS Heartbeat Interrupt signal (see "Scheduler" in *Apple IIGS Toolbox Reference*). Its purpose is to coordinate the execution of interrupt handlers and other interrupt-based routines such as desk accessories.

The Scheduler is required only when an interrupt routine needs to call a piece of system software, such as ProDOS 16, that is not **reentrant**. If ProDOS 16 is in the middle of a call when an interrupt occurs, the interrupting routine cannot itself call ProDOS 16, because that would disrupt the first (not yet completed) call. The system needs a way of telling an interrupt routine to hold off until the system software it needs is no longer busy.

The Scheduler accomplishes this by periodically checking a word-length flag called the **Busy word** and maintaining a queue of processes that may be activated when the Busy word is cleared. Interrupt routines that make operating system calls must go through the Scheduler (see Chapter 7).

The User ID Manager

The User ID Manager is a Miscellaneous tool set that provides a way for programs to obtain unique identification numbers. Every memory block allocated by the Memory Manager is marked with a User ID that shows what system software, application, or desk accessory it belongs to.

Part of each block's 2-byte User ID is a **TypeID** field, describing the category of load segment that occupies it. All ProDOS 8 and ProDOS 16 blocks are type 3; System Loader blocks are type 7; blocks of controlling programs (such as a shell or switcher) are type 2; and blocks containing application segments are type 1. Appendix D diagrams the format for the User ID word. See "Miscellaneous Tool Sets" in *Apple IIGS Toolbox Reference* for further details.

ProDOS 16 and the System Loader rely on User ID's to help them restart or reload applications. See "Quit Procedure" in this chapter, and "Restart" and "User Shutdown" in Chapter 17.

The System Failure Manager

All fatal errors, including fatal ProDOS 16 errors, are routed through the System Failure Manager, a Miscellaneous Tool Set. It displays a default message on the screen, or, if passed a pointer when it is called, displays an ASCII string with a user-chosen message. Program execution halts when the System Failure Manager is called.

The System Failure Manager is described under "Miscellaneous Tool Sets" in *Apple IIGS Toolbox Reference*.

Chapter 6

Programming With ProDOS 16

This chapter presents requirements and suggestions for writing Apple IIGS programs that use ProDOS 16.

Programming suggestions for the System Loader are in Chapter 16 of this manual. More general information on how to program for the Apple IIGS is available in *Programmer's Introduction to the Apple IIGS*. For language-specific programming instructions, consult the appropriate language manual in the Apple IIGS Programmer's Workshop (see "Apple IIGS Programmer's Workshop" in this chapter).

Application requirements

As used in this manual, an **application** is a complete program, typically called by a user, that can communicate directly with ProDOS 16 and any other system software or firmware it needs. For example, word processors, spreadsheet programs, and programming-language interpreters are examples of applications. Data files and source-code files, as well as subroutines, libraries, and utilities that must be called from other programs are not applications.

To be an application, an Apple IIGS program must

- consist of executable machine-language code
- be in Apple IIGS object module format (see Appendix D)
- be file type $B3 (specialized applications may have other file types—see Appendix A)
- have a filename extension of `.SYS16` (if you want it to be self-booting at system startup—see Chapter 5)
- make ProDOS 16 calls as described in this manual (see Chapter 8)
- observe the ProDOS 16 `QUIT` conventions (see Chapter 5)
- observe all other applicable ProDOS 16 conventions, such as the conventions for interrupt handlers (see Chapter 7)
- get all needed memory from the Memory Manager (see Chapter 3)

Most other aspects of the program are up to you. The rest of this chapter presents conventions and suggestions to help you create an efficient and useful application, consistent with Apple IIGS programming concepts and practices.

Stack and direct page

In the Apple IIGS, the 65C816 microprocessor's stack-pointer register is 16 bits wide; that means that, in theory, the hardware stack may be located anywhere in bank $00 of memory, and the stack may be as much as 64K bytes deep.

The **direct page** is the Apple IIGS equivalent to the standard Apple II **zero page**. The difference is that it need not be page zero in memory. Like the stack, the direct page may theoretically be placed in any unused area of bank $00—the microprocessor's **direct register** is 16 bits wide, and all zero-page (direct-page) addresses are added as offsets to the contents of that register.

In practice, however, there are several, restrictions on available space. First, only the lower 48K bytes of bank $00 can be allocated—the rest is reserved for I/O space and system software. Also, because more than one program can be active at a time, there may be more than one stack and more than one direct page in bank $00. Furthermore, many applications may want to have parts of their code as well as their stacks and direct pages in bank $00.

Your program should therefore be as efficient as possible in its use of stack and direct-page space. The total size of both should probably not exceed about 4K bytes in most cases. Still, that gives you the opportunity to write programs that require stacks and direct pages much larger than the 256 bytes available for each on standard Apple II computers.

Automatic allocation of stack and direct page

Only you can decide how much stack and direct-page space your program will need when it is running. The best time to make that decision is during program development, when you create your source file(s). If you specify at that time the total amount of space needed, ProDOS 16 and the System Loader will automatically allocate it and set the stack and direct registers each time your program runs.

Definition during program development

You define your program's stack and direct-page needs by specifying a "direct-page/stack" object segment (KIND = $12; see Appendix D) when you assemble or compile your program (Figure 6-1). The size of the segment is the total amount of stack and direct-page space your program needs. It is not *necessary* to create this segment; if you need no such space or if the ProDOS 16 default is sufficient (see "ProDOS 16 default stack and direct page" later in this section), you may leave it out.

When the program is linked, it is important that the direct-page/stack segment not be combined with any other object segments to make a load segment—the linker must create a single load segment corresponding to the direct-page/stack object segment. If there is no direct-page/stack object segment, the linker will not create a corresponding load segment.

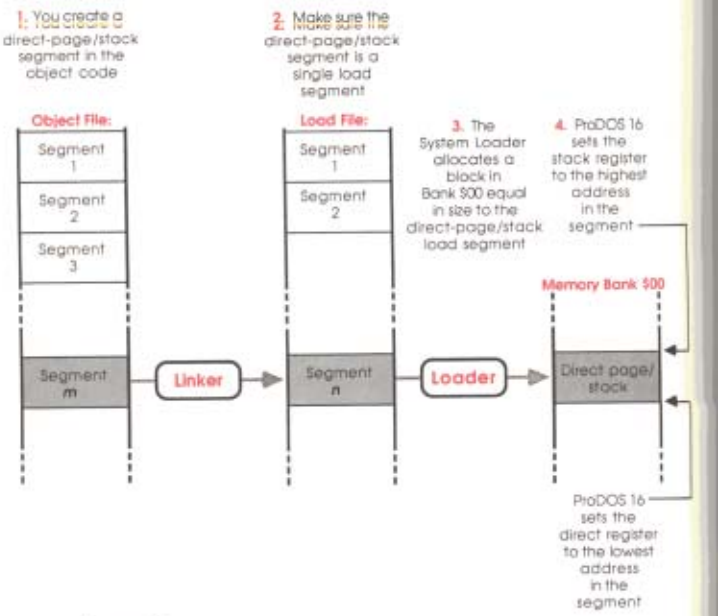


Figure 6-1
Automatic direct-page/stack allocation

Allocation at run time

Each time the program is started, the System Loader looks for a direct-page/stack load segment. If it finds one, the loader calls the Memory Manager to allocate a page-aligned, locked memory block of that size in bank $00. The loader loads the segment and passes its base address and size, along with the program's User ID and starting address, to ProDOS 16. ProDOS 16 sets the A (accumulator), D (direct), and S (stack) registers as shown, then passes control to the program:

A = User ID assigned to the program
D = address of the first (lowest-address) byte in the direct-page/stack space
S = address of the last (highest-address) byte in the direct-page/stack space

By this convention, direct-page addresses are offsets from the base of the allocated space, and the stack grows downward from the top of the space.

Important ProDOS 16 provides no mechanism for detecting stack overflow or underflow, or collision of the stack with the direct page. Your program must be carefully designed and tested to make sure this cannot occur.

When your program terminates with a `QUIT` call, the System Loader's Application Shutdown function makes the direct-page/stack segment purgeable, along with the program's other static segments. As long as that segment is not subsequently purged, its contents are preserved until the program restarts. See "Application Shutdown" and "Restart" in Chapter 17.

❖ *Note:* There is no provision for extending or moving the direct-page/stack space after its initial allocation. Because bank $00 is so heavily used, any additional space you later request may be unavailable—the memory adjoining your stack is likely to be occupied by a locked memory block. Make sure that the amount of space you specify at link time fills all your program's needs.

ProDOS 16 default stack and direct page

If the loader finds no direct-page/stack segment in a file at load time, it still returns the program's User ID and starting address to ProDOS 16, but it does not call the Memory Manager to allocate a direct-page/stack space and it returns zeros as the base address and size of the space. ProDOS 16 then calls the Memory Manager itself, and allocates a 1K direct-page/stack segment with the following attributes:

size:	1,024 bytes
owner:	program with the User ID returned by the loader
fixed/movable:	fixed
locked/unlocked:	locked
purge level:	1
may cross bank boundary?	no
may use special memory?	yes
alignment:	page-aligned
absolute starting address?	no
fixed bank?	yes—bank $00

See *Apple IIGS Toolbox Reference* for a general description of memory block attributes assigned by the Memory Manager.

Once allocated, the default direct-page/stack is treated just as it would be if it had been specified by the program: ProDOS 16 sets the A, D, and S registers before handing control to the program, and at shutdown time the System Loader purges the segment.

Manual allocation of stack and direct page

Your program may allocate its own stack and direct-page space at run time, if you prefer. When ProDOS 16 transfers control to your program, be sure the program saves the User ID value left in the accumulator before doing the following:

1. Using the starting or ending address left in the D or S register by ProDOS 16, it should make a `FindHandle` call to the Memory Manager, to get the memory handle of the automatically-provided direct-page/stack space. Then, using that handle, it should get rid of the space with a `DisposeHandle` call.
2. It can now allocate its own direct-page/stack space through the Memory Manager `NewHandle` call. The allocated block must be *purgeable*, *fixed*, and *locked*.
3. Finally, the program must place the appropriate values (beginning and end addresses of the segment) in the D and S registers.

Managing system resources

Various hardware and software features of the Apple IIGS can provide an application with useful information, or can otherwise increase its flexibility. This section suggests ways to use those features.

Global variables

Under ProDOS 8, a fixed-address **global page** maintains the values of important variables and addresses for use by applications. The global page is at the same address in any machine or machine configuration that supports ProDOS 8, so an application can always access those variables at the same addresses.

ProDOS 16 does not provide a global page. Such a set of fixed locations is inconsistent with the flexible and dynamic memory management system of the Apple IIGS. Instead, calls to ProDOS 16, tools, or firmware give you the information formerly provided by the global page. Table 6-1 shows the Apple IIGS calls used to obtain information equivalent to ProDOS 8 global page values.

Table 6-1
Apple IIgs equivalents to ProDOS 8 global page information

Global page information	Apple IIgs Equivalent
Global page entry points	(not supported)
Device driver vectors	(not supported)
List of active devices	returned by `VOLUME` call (ProDOS 16)
Memory Map	(responsibility of the Memory Manager)
Pointers to I/O buffers	returned by `OPEN` call (ProDOS 16)
Interrupt vectors	returned by `ALLOC_INTERRUPT` call (ProDOS 16)
Date/Time	returned by `ReadTime` call (Misc. tool set)
System Level	returned by `GET_LEVEL` call (ProDOS 16)
`MACHID`	(not supported)
Application version	(not supported)
ProDOS 16 Version	returned by `GET_VERSION` call (ProDOS 16)

Of course, the Apple IIGS always supports the ProDOS 8 global page *when a ProDOS 8 application is running*.

Prefixes

The nine available prefixes described in Chapter 5 offer convenience in coding pathnames and flexibility in writing for different system and application disk volumes. For example, any files on the boot disk can always be accessed through the prefix `*/`, regardless of the volume name of that particular boot disk. Any library routine in the system library subdirectory will have the prefix `2/`, regardless of which system disk is on line (unless your program has changed the value of the prefix). If you put routines specific to your application in the same subdirectory as your application, they can always be called with the prefix `1/`, regardless of what subdirectory or disk your program inhabits.

Your application can always change the values of any of the prefixes except */. For example, it may change prefix 2/ if it wishes to access libraries (or any other files) on a volume other than the boot volume. But be careful: once you change prefix 1/, for example, you can no longer use it as the application prefix. Be sure to save the value of a prefix number before you change it, if you may want to recover it later.

Native mode and emulation mode

You can make ProDOS 16 calls when the processor is in either emulation mode or native mode. So if part of your program requires the processor to be in emulation mode, you needn't reset it to native mode before calling ProDOS 16. However, emulation-mode calls to ProDOS 16 must be made *from bank $00*, and they can reference information (such as parameter blocks) in bank $00 only. Furthermore, interrupts must be *disabled*.

ProDOS 8 programs run entirely in emulation mode. If you wish to modify a ProDOS 8 program to run under ProDOS 16, or if you wish to use Apple IIGS features available only in native mode, see "Revising a ProDOS 8 Application for ProDOS 16" in this chapter. See also *Programmer's Introduction to the Apple IIGS*.

Setting initial machine configuration

When an Apple IIGS application (type $B3) is first launched, the Apple IIGS is in full native mode with graphics shadowing off (see "Machine Configuration at Application Launch" in Chapter 5). If your program needs a different machine configuration, it must make the proper settings once it gains control.

ProDOS 16 does not initialize soft switches, firmware registers, or any hardware registers other than those listed in Chapter 5. Your program is responsible for initializing any needed switches and registers.

Allocating memory

All memory allocation is done through calls to the Memory Manager, described in *Apple IIGS Toolbox Reference*. Memory space you request may be either movable or unmovable (fixed). If it is movable, you access it through a memory handle; if it is unmovable, you may access it through a handle or through a pointer. Since the Memory Manager does not return a pointer to an allocated block, you obtain the pointer by dereferencing the handle (see Chapter 3).

ProDOS 16 parameter blocks are referenced by pointers; if you do not code them into your program segments and reference them with labels, you must put them in unmovable memory blocks. See "Setting up a Parameter Block in Memory" in Chapter 8.

Loading another program

If you do not want your program to load another program when it finishes, it should use a ProDOS 16 `QUIT` call with all parameters set to 0. The `QUIT` routine performs all necessary functions to shut down the current application, and normally brings up a program selector which allows the user to choose the next program to load. Most applications function this way.

However, if you want your application to load and execute another application, there are several ways to do it. If you wish to pass control *permanently* to another application, use the ProDOS 16 `QUIT` call with only a pathname pointer, as described in Chapter 5. If you wish control to *return* to your application once the next application is finished, use also the return flag parameter in the `QUIT` call. That way your program can function similarly to a shell—whenever it quits to another specified program, it knows that it will eventually be re-executed.

If you wish to load but not necessarily pass control to another program, or if you want your program to remain in memory after it passes control to another program, use the System Loader's Initial Load function (described in Chapter 17). When your program actively loads other program files, it is called a **controlling program**; the APW Shell (see "Apple IIGS Programmer's Workshop" in this chapter) is a controlling program. Chapter 16 gives suggestions for writing controlling programs.

You can load a ProDOS 8 application (type $FF) through the ProDOS 16 QUIT call, but you cannot do so with the System Loader's Initial Load call; the System Loader will load only ProDOS 16 load files (types $B3-$BE).

❖ *Note:* Because ProDOS 8 will not load type $B3 files, ProDOS 8-based applications that load and run other applications cannot run any ProDOS 16 applications. This restriction is a natural consequence of the lack of downward compatibility. If you wish to modify an older application to be able to use it with ProDOS 16, see "Revising a ProDOS 8 Application for ProDOS 16," later in this chapter.

Using interrupts

ProDOS 16 provides conventions (see Chapter 7) to ensure that interrupt-handling routines will function correctly. If you are writing a print spooler, game, communications program or other routine that uses interrupts, please follow those conventions.

As explained in Chapter 4, an *unclaimed interrupt* causes a system failure: control is passed to the System Failure Manager and execution halts. Your program may pass a message to the System Failure Manager to display on the screen when that happens. In addition, because the System Failure Manager is a tool, and because all tools may be replaced by user-written routines, you may substitute your own error handler for unclaimed interrupts. See *Apple IIGS Toolbox Reference* for information on the System Failure Manager and for instructions on writing your own tool set.

If ProDOS 16 is called while it is in the midst of another call, it issues a "ProDOS is busy" error. This situation normally arises only when an interrupt handler makes ProDOS 16 calls; a typical application will always find ProDOS 16 free to accept a call. Chapter 7 provides instructions on how to avoid this error when writing interrupt handlers; nevertheless, *all programs* should be able to handle the "ProDOS is busy" error code in case it occurs.

Accessing devices

Under ProDOS 8, block devices on Apple II computers are specified by a *unit number*, related to slot and drive number (such as slot 5, drive 1). ProDOS 16 does not directly support that numbering system; instead, it identifies devices by *device number* and *device name*. As explained in Chapter 4, device numbers are assigned in order of the device search at system startup, and device names are assigned according to a simple ProDOS 16 convention. You must use device numbers or names in ProDOS 16 device calls.

For filing calls and for one device call (GET_DEV_NUM), you may also access a device through the name of the volume on the device. In addition, you may use the GET_LAST_DEV call to identify the last device accessed, in case you wish to access it again.

File creation/modification date and time

The information in this section is important to you if you are writing a file or disk utility program, or any routine that copies files.

All ProDOS 16 files are marked with the date and time of their creation. When a file is first created, ProDOS 16 stamps the file's directory entry with the current date and time from the system clock. If the file is later modified, ProDOS 16 then stamps it with a modification date and time (its creation date and time remain unchanged).

The creation and modification fields in a file entry refer to the *contents* of the file. The values in these fields should be changed only if the contents of the file change. Since data in the file's directory entry itself are not part of the file's contents, the modification field should not be updated when another field in the file entry is changed, *unless* that change is due to an alteration in the file's contents. For example, a change in the file's name is not a modification; on the other hand, a change in the file's EOF always reflects a change in its contents and therefore is a modification.

Remember also that a file's entry is a part of the contents of the directory or subdirectory that contains that *entry*. Thus, whenever a file entry is changed in any way (whether or not its modification field is changed), the modification fields in the entries for all its enclosing subdirectories—including the volume directory—must be updated.

Finally, when a file is *copied*, a utility program must be sure to give the copy the same creation and modification date and time as the original file, and *not* the date and time at which the copy was created.

To implement these concepts, file utility programs should note the following procedures:

1. To create a new file:

a. Set the creation and modification fields of the file's entry to the current system date and time.

b. Set the modification fields in the entries of all subdirectories in the path containing the file to the current system date and time.

2. To rename a file:

a. Do not change the file's modification field.

b. Set the modification fields of all subdirectories in the path containing the file to the current system date and time.

3. To alter the contents of a file:

a. ProDOS 16 considers a file's contents to have been modified if any `WRITE` or `SET_EOF` operation has been performed on the file while it is open. If that condition has been met, set the file's modification field to the current system date and time when the file is closed.

b. Also set the modification fields of all subdirectories in the path containing the file to the current system date and time.

4. To delete a file:

a. Delete the file's entry from the directory or subdirectory that contains it.

b. Set the modification fields of all subdirectories in the path containing the deleted file to the current system date and time.

5. To copy a file:

a. Make a `GET_FILE_INFO` call on the source file (the file to be copied), to get its creation and modification dates and times.

b. Make a `CREATE` call to create the destination file (the file to be copied to). Give it the creation date and time values obtained in step (a).

c. Open both the source and destination files. Use `READ`s and `WRITE`s to copy the source to the destination. Close both files.

❖ *Note:* The procedure for copying sparse files is more complicated than this. See Chapter 2 and Appendix A.

d. Make a `SET_FILE_INFO` call on the destination file, using all the information returned from `GET_FILE_INFO` in step (a). This sets the modification date and time values to those of the source file.

ProDOS 16 automatically carries out all steps in procedures (1) through (4). Procedure (5) is the responsibility of the file-copying utility.

Revising a ProDOS 8 application for ProDOS 16

If you have written a ProDOS 8-based program for a standard Apple II (64K Apple II Plus, Apple IIe, or Apple IIc), it will run without modification on the Apple IIGS. The only noticeable difference will be its faster execution because of the greater clock speed of the Apple IIGS. However, the program will not be able to take advantage of any advanced Apple IIGS features such as large memory, the toolbox, the mouse-based interface, and new graphics and sound abilities. This section discusses some of the basic alterations necessary to upgrade a ProDOS 8 application for native-mode execution under ProDOS 16 on the Apple IIGS.

Because ProDOS 16 closely parallels ProDOS 8 in function names and calling structure, it is not difficult to change system calls from one ProDOS to the other. But several other aspects of your program also must be redesigned if it is to run in native mode under ProDOS 16. Depending on the program's size and structure and the new features you wish to install, those changes may range from minor to drastic.

Memory management

Because the Apple IIGS supports segmented load files, one of the first decisions to make is whether and how to segment the program (both the original program and any added parts), and where in memory to put the segments.

To help decide where in memory to place pieces of your program, consider that execution speed is related to memory location: banks $E0 and $E1 execute at standard clock speed, and all the other banks execute at fast clock speed (see *Apple IIGS Hardware Reference*). Those parts of your program that are executed most often should probably go into fast memory, while less-used parts and data segments may be appropriate in standard-speed memory. On the other hand, because all I/O goes through banks $E0 or $E1, program segments that make heavy use of I/O instructions might work best in standard-speed memory. Performance testing of the completed program is the only way to accurately determine where segments should go.

Memory management methods are completely different under ProDOS 16 than under ProDOS 8. If your ProDOS 8 program manages memory by allocating its own memory space and marking it off in the global page bit map, the ProDOS 16 version must be altered so that it requests all needed space from the Memory Manager. Whereas ProDOS 8 does not check to see if you are using only your marked-off space, the Memory Manager under ProDOS 16 will not assign to your program any part of memory that has already been allocated.

Hardware configuration

ProDOS 8 applications run only in 6502 emulation mode on the Apple IIGS. That does not mean that applications converted to run under ProDOS 16 must necessarily run in native mode. There are at least three configurations possible:

- The program may run in emulation mode, but make ProDOS 16 calls.
- The program may run in native mode with the m- and x-bits set. The accumulator and index registers will remain 8 bits wide.
- The program may run in full native mode (m- and x-bits cleared).

Modifying a program for the first configuration probably involves the least effort, but returns the least benefit.

Modifying a program to run in full native mode is the most difficult, but it makes best use of all Apple IIGS features.

Converting system calls

For most ProDOS 8 calls, there is an equivalent ProDOS 16 call with the same name. Each call block must be modified for ProDOS 16: the `JSR` (Jump to Subroutine) assembly-language instruction replaced with a `JSL` (Jump to Subroutine *Long*), the call number field made 2 bytes long, and the parameter list pointer made 4 bytes long. The only other conversion required is the reconstruction of the parameter block to the ProDOS 16 format.

For other ProDOS 8 calls, the ProDOS 16 equivalent performs a slightly different task, and the original code will have to be changed to account for that. For example, in ProDOS 8 an `ON_LINE` call can be used directly to determine the names of all online volumes; in ProDOS 16 a succession of `VOLUME` calls is required. Refer to the detailed descriptions in Chapters 9 through 13 to see which ProDOS 16 calls are different from their ProDOS 8 counterparts.

Still other ProDOS 8 calls have no equivalent in ProDOS 16. They are listed and described under "Eliminated ProDOS 8 System Calls," in Chapter 1. If your program uses any of these calls, they will have to be replaced as shown.

Modifying interrupt handlers

If you have written an interrupt handling routine, it needs to be updated to conform with the ProDOS 16 interrupt handling conventions. There may be very few changes necessary: it must return with an `RTL` (Return from subroutine *Long*) rather than an `RTS` (Return from Subroutine), and it must start and end in 65C816 native mode. See Chapter 7.

Converting stack and zero page

The fixed stack and zero-page locations provided for your program by ProDOS 8 are not available under ProDOS 16. You may either let ProDOS 16 assign you a default 1,024-byte space, or you may define a direct-page/stack segment in your object code. In either case, the loader may place the segment anywhere in bank $00—you cannot depend on any specific address being within the space. See "Stack and Direct Page," earlier in this chapter.

Compilation/assembly

Once your source code has been modified and augmented as desired, you need to recompile/reassemble it. You must use an assembler or compiler that produces object files in Apple IIGS object module format (OMF); otherwise the program cannot be properly linked and loaded for execution. Using a different compiler or assembler may mean that, in addition to modifying your program code, you might have to change some assembler directives to follow the syntax of the new assembler.

If you have been using the EDASM assembler, you will not be able to use it to write Apple IIGS programs. The Apple IIGS Programmer's Workshop is a set of development programs that allow you to produce and edit source files, assemble/compile object files, and link them into proper OMF load files. See "Apple IIGS Programmer's Workshop" in this chapter.

After your revised program is linked, assign it the proper Apple IIGS application file type (normally $B3) with the APW `File Type` utility.

Apple IIGS Programmer's Workshop

The Apple IIGS Programmer's Workshop (APW) is a powerful set of development programs designed to facilitate the creation of Apple IIGS applications. If you are planning to write programs for the Apple IIGS, APW will make your job much easier. The Workshop includes the following components:

- Shell
- Editor
- Linker
- Debugger
- Assembler
- C Compiler

All these components work together (under the Shell) to speed the writing, compiling or assembling, and debugging of programs. The Shell acts as a command interpreter and an interface to ProDOS 16, providing several operating system functions and file utilities that can be called by users and by programs running under the Shell.

See the following manuals for more information on the Apple IIGS Programmer's Workshop:

- *Apple IIGS Programmer's Workshop Reference* (describes the Shell, Editor, Linker, and Debugger)
- *Apple IIGS Programmer's Workshop Assembler Reference*
- *Apple IIGS Programmer's Workshop C Reference*

Human Interface Guidelines

All people who develop application programs for the Apple IIGS computer are strongly encouraged to follow the principles presented in *Human Interface Guidelines: The Apple Desktop Interface*. That manual describes the **desktop user interface** through which the computer user communicates with his computer and the applications running on it. This section briefly outlines a few of the human interface concepts; please refer to the manual for specific design information.

The Apple Desktop Interface, first introduced with the Macintosh™ computer, is designed to appeal to a nontechnical audience. Whatever the purpose or structure of your application, it will comunicate with the user in a consistent, standard, and non-threatening manner if it adheres to the Desktop Interface standards. These are some of the basic principles:

- **Human control:** Users should feel that they are controlling the program, rather than the reverse. Give them clear alternatives to select from, and act on, their selections consistently.
- **Dialog:** There should be a clear and friendly dialog between human and computer. Make messages and requests to the user in plain English.
- **Direct Manipulation and Feedback:** The user's physical actions should produce physical results. When a key is pressed, place the corresponding letter on the screen. Use highlighting, animation, and dialog boxes to show users the possible actions and their consequences.
- **Exploration:** Give the user permission to test out the possibilities of the program without worrying about negative consequences. Keep error messages infrequent. Warn the user when risky situations are approached.

- **Graphic design:** Good graphic design is a key feature of the guidelines. Objects on the screen should be *simple* and *clear*, and they should have *visual fidelity* (that is, they should look like what they represent). *Icons* and *palettes* are common graphic elements that need careful design.
- **Avoiding modes:** a *mode* is a portion of an application that the user has to formally enter and leave, and that restricts the operations that can be performed while it's in effect. By restricting the user's options, modes reinforce the idea that computers are unnatural and unfriendly. Use modes sparingly.
- **Device-independence:** Make your program as hardware-independent as possible. Don't bypass the tools and resources in ROM—your program may become incompatible with future products and features.
- **Consistency:** As much as possible, all applications should use the same interface. Don't confuse the user with a different interface for each program.
- **Evolution:** Consistency does not mean that you are restricted to using existing desktop features. New ideas are essential for the evolution of the Human Interface concept. If your application has a feature that is described in *Human Interface Guidelines*, you should implement it exactly as described; if it is something new, make sure it cannot be confused with an existing feature. It is better to do something completely different than to half agree with the guidelines.

Chapter 7

Adding Routines to ProDOS 16

This chapter discusses additional specific routines that may be used with ProDOS 16. Because these routines are directly connected to ProDOS 16 and interact with it at a low level, they are essentially transparent to applications and can be considered "part of" ProDOS 16. Interrupt handlers are the only such extensions to ProDOS 16 presently supported.

Interrupt handlers

The Apple IIGS has extensive firmware interrupt support (see *Apple IIGS Firmware Reference*). In addition, ProDOS 16 supports up to 16 user-installed interrupt handlers (see Chapter 4). If you write an interrupt handler, it should follow the conventions described here. Note also the precautions you must take if your handler makes operating system calls.

Interupt handler conventions

Interrupt handling routines written for the Apple IIGS must follow certain conventions. The interrupt dispatcher will set the following machine state before passing control to an interrupt handler:

```
e     = 0
m     = 0
x     = 0
i     = 1
c     = 1
speed = high
```

Before returning to ProDOS 16, the interrupt handler must restore the machine to the following state:

```
e     = 0
m     = 0
x     = 0
i     = 1
speed = high
```


In addition the c flag must be cleared (= 0) if the handler serviced the interrupt, and set (= 1) if the handler did not service the interrupt. The handler must return with an `RTL` instruction.

When an interrupt is passed to ProDOS 16, ProDOS 16 first sets the processor to full native mode, then successively polls the installed interrupt handlers. If one of them services the interrupt, ProDOS 16 knows because it checks the value of the c flag when the routine returns. If the c flag is cleared, ProDOS 16 switches back to a standard Apple II configuration in emulation mode, and performs an `RTI` to the Apple IIGS firmware interrupt handling system. If *no* handler services the interrupt, it is an unclaimed interrupt and it will result in system failure (see Chapter 4).

Installing interrupt handlers

Interrupt handlers are installed with the `ALLOC_INTERRUPT` call and removed with the `DEALLOC_INTERRUPT` call. The ProDOS 16 interrupt dispatcher maintains an **interrupt vector table**, an array of up to 16 vectors to interrupt handlers. As each successive `ALLOC_INTERRUPT` call is made, the dispatcher adds another entry to the end of the table. Each time a `DEALLOC_INTERRRUPT` call is made to delete a vector from the table, the remaining vectors are moved toward the beginning of the array, filling in the gap. Interrupt handling routines are polled by ProDOS 16 in the order in which their vectors occur in the interrupt vector table.

There is no way to reorder interrupt vectors except by allocating and deallocating interrupts. Interrupts that occur often or require fast service should be allocated first, so their vectors will be near the beginning of the interrupt vector table. If you need *extremely* fast interrupt service, install your interrupt handler directly in the Apple IIGS firmware interrupt dispatcher, rather than through ProDOS 16. See *Apple IIGS Firmware Reference* for further information.

Be sure to enable the hardware generating the interrupt only *after* the routine to handle it is allocated; likewise, disable the hardware *before* the routine is deallocated. Otherwise, a fatal unclaimed interrupt error may occur (see "Unclaimed Interrupts" in Chapter 4).

Making operating system calls during interrupts

ProDOS 16 is not reentrant. That is, it does not save its own state when interrupted. It therefore is illegal to make an operating system call while another operating system call is in progress; if a call is attempted, ProDOS 16 will return an error (number $07, "ProDOS is busy").

For applications this is not a problem; the operating system is always free to accept a call from them. Only routines that are started through interrupts (such as interrupt handlers and desk accessories) need be careful not to call ProDOS 16 while it is busy.

One acceptable procedure is for the interrupt handler to consult the ProDOS busy flag at location $E100BE–$E100BF (see Table 3-3), and simply not make the system call unless ProDOS 16 is not busy.

If an interrupt handler really needs to make an operating system call, it must be prepared to deal with a returned "ProDOS is busy" error. If that happens the handler should

1. Defer itself temporarily
2. Return control to the operating system so that the operating system may complete the current call
3. Regain control when the operating system is no longer busy, and make its own system call

The Scheduler, part of a ROM-based tool set, allows interrupt handlers to follow these procedures in a simple, standard way. The Scheduler consults a system **Busy word** that keeps track of non-reentrant system software that is in use. ProDOS 16 executes the Scheduler routine `INCBUSYFLAG` whenever it is called, and `DECBUSYFLAG` before it returns from a call. An interrupt handler may use the Scheduler's `SCHADDTASK` routine to place itself in a queue of tasks waiting for ProDOS 16 to complete any calls in progress. See *Apple IIGS Toolbox Reference* for detailed information.

Part II

ProDOS 16 System Call Reference

This part of the manual describes the ProDOS 16 system calls in detail. The calls are grouped into five categories:

- File housekeeping calls (Chapter 9)
- File access calls (Chapter 10)
- Device calls (Chapter 11)
- Environment calls (Chapter 12)
- Interrupt control calls (Chapter 13)

Chapter 8 shows how to make the calls, and explains the format for the call descriptions in Chapters 9 through 13. See Appendix E for a list of all ProDOS 16 errors returned by the calls.

Chapter 8

Making ProDOS 16 Calls

Any independent program in the Apple IIGS that makes system calls is known as a ProDOS 16 *calling program* or *caller*. The current application, a desk accessory, and an interrupt handler are examples of potential callers. A ProDOS 16 caller makes a system call by executing a **call block**. The call block contains a pointer to a **parameter block**. The parameter block is used for passing information between the caller and the called function; additional information about the call is reflected in the state of certain hardware registers. This chapter discusses these aspects of system calls and compares them with the calling method used in ProDOS 8.

❖ *Note:* The phrase *system call* as used here is synonymous with *operating system call* or *ProDOS 16 call*, and is equivalent to **MLI call** for ProDOS 8. It includes all calls to the operating system for accessing system information and manipulating open or closed files. It is not restricted to what are called "system calls" in the *ProDOS 8 Technical Reference Manual*.

The call block

A system call block consists of a `JSL` (Jump to Subroutine *Long*) to the ProDOS 16 entry point, followed by a 2-byte system call number and a 4-byte parameter block pointer. ProDOS 16 performs the requested function, if possible, and returns execution to the instruction immediately following the call block.

All applications written for the Apple IIGS under ProDOS 16 must use the system call block format. When making the call, the caller may have the processor in emulation mode or full native mode or any state in between (see *Technical Introduction to the Apple IIGS*).

❖ *Note:* To call ProDOS 16 while running in emulation mode, your program must be in bank $00 and interrupts must be disabled.

The call block looks like this:

```
PRODOS      GEQU  $E100A8          ; fixed entry vector
            .
            .
            .
            JSL   PRODOS           ; Dispatch call to ProDOS 16 entry
            DC    I2'CALLNUM'      ; 2-byte call number
            DC    I4'PARMBLOCK'    ; 4-byte parameter block pointer
            BCS   ERROR            ; If carry set, go to error handler
                                   ; otherwise, continue…
            .
            .
            .
ERROR                              ; error handler
            .
            .
            .
PARMBLOCK                          ; parameter block
```

The call block itself consists of only the JSL instruction and the DC (Define Constant) assembler directives. The BCS (Branch on Carry Set) instruction in this example is a conditional branch to an error handler called ERROR.

A JSL rather than a JSR (Jump to Subroutine) is required because the JSL uses a 3-byte address, allowing a caller to make the call from anywhere in memory. The JSR instruction uses only a 2-byte address, restricting it to jumps and returns within the current (64K) block of memory.

The parameter block

A parameter block is a specifically formatted table that occupies a set of contiguous bytes in memory. It consists of a number of fields that hold information that the calling program supplies to the function it calls, as well as information returned by the function to the caller.

Every ProDOS 16 call requires a valid parameter block (PARMBLOCK in the example just given), referenced by a 4-byte pointer in the call block. The caller is responsible for constructing the parameter block for each call it makes; the list may be anywhere in memory. Formats for individual parameter blocks accompany the detailed system call descriptions in Chapters 9 through 13.

Types of parameters

Each field in a parameter block contains a single parameter. There are three types of parameters: values, results, and pointers. Each is either an *input* to ProDOS 16 from the caller, or an *output* from ProDOS 16 to the caller. The minimum field size for a parameter is one **word** (2 bytes; see Table 3-1).

- A **value** is a numerical quantity, 1 or more words long, that the caller passes to ProDOS 16 through the parameter block. It is an *input* parameter.
- A **result** is a numerical quantity, 1 or more words long, that ProDOS 16 places into the parameter block for the caller to use. It is an *output* parameter.
- A **pointer** is the 4-byte (**long word**) address of a location containing data, code, an address, or buffer space in which ProDOS 16 can receive or place data. The pointer itself is an input for all ProDOS 16 calls; the data it points to may be either *input* or *output*.

A parameter may be both a value and a result. Also, a pointer may designate a location that contains a value, a result, or both.

❖ *Note:* A handle is a special type of pointer; it is a pointer to a pointer. It is the 4-byte address of a location that *itself* contains the address of a location containing data, code, or buffer space. ProDOS 16 uses a handle parameter only in the OPEN call (Chapter 10); in that call the handle is an *output* (result).

Parameter block format

All parameter fields that contain block numbers, block counts, file offsets, byte counts, and other file or volume dimensions are 4 bytes long. Requiring 4-byte fields ensures that ProDOS 16 will accommodate future large devices using **guest file systems**.

All parameter fields contain an even number of bytes, for ease of manipulation by the16-bit 65C816 processor. Thus pointers, for example, are 4 bytes long even though 3 bytes are sufficient to address any memory location. Wherever such extra bytes occur they must be set to zero by the caller; if they are not, compatibility with future versions of ProDOS 16 will be jeopardized.

Pointers in the parameter block must be written with the low-order byte of the low-order word at the lowest address.

Comparison of ProDOS 16 parameter blocks with their ProDOS 8 counterparts reveals that in some cases the order of parameters is slightly different. These alterations have been made to facilitate sharing a single parameter block among a number of calls. For example, most file access calls can be made with a single parameter block for each open file; under ProDOS 8 this sharing of parameter blocks is not possible.

Important

A parameter's field width in a ProDOS 16 parameter block is often very different from the range of permissible values for that parameter. The fact that all fields are an even number of bytes is one reason. Another reason is that certain fields are larger than presently needed in anticipation of the requirements of future guest file systems. For example, the ProDOS 16 `CREATE` call's parameter block includes a 4-byte `aux_type` field, even though, on disk, the `aux_type` field is only 2 bytes wide (see "Format and Organization of Directory Files" in Appendix A). The two high-order bytes in the field must therefore always be zero.

Ranges of permissible values for all parameters are given as part of the system call descriptions in the following chapters. When coding a parameter block, note carefully the range of permissible values for each parameter, and make sure that the value you assign is within that range.

Setting up a parameter block in memory

Each ProDOS 16 call uses a 4-byte pointer to point to its parameter block, which may be anywhere in memory. All applications must obtain needed memory from the Memory Manager, and therefore cannot know in advance where the memory block holding such a parameter block will be.

There are two ways to set up a ProDOS 16 parameter block in memory:

1. Code the block directly into the program, referencing it with a label. This is the simplest and most typical way to do it. The parameter block will always be correctly referenced, no matter where in memory the program code is loaded.

2. Use Memory Manager and System Loader calls to place the block in memory:
   a. Request a memory block of the proper size from the Memory Manager. Use the procedures described in *Apple IIGS Toolbox Reference*. The block should be either *fixed* or *locked*.
   b. Obtain a pointer to the block, by dereferencing the memory handle returned by the Memory Manager (that is, read the contents of the location pointed to by the handle, and use that value as a pointer to the block).
   c. Set up your parameter block, starting at the address pointed to by the pointer obtained in step (b).

Register values

There are no register requirements on entry to a ProDOS 16 call. ProDOS 16 saves and restores all registers except the accumulator (A) and the processor status register (P); those two registers store information on the success or failure of the call. On exit, the registers have these values:

A	zero if call successful; if nonzero, number is the error code
X	unchanged
Y	unchanged
S	unchanged
D	unchanged
P	(see below)
DB	unchanged
PB	unchanged
PC	address of location following the parameter block pointer

"Unchanged" means that ProDOS 16 initially saves, and then restores when finished, the value the register had just before the JSL `PRODOS 8` instruction.

On exit, the processor status register (P) bits are

n	undefined
v	undefined
m	unchanged
x	unchanged
d	unchanged
i	unchanged
z	undefined
c	zero if call successful, 1 if not
e	unchanged

❖ *Note:* ProDOS 16 treats several flags differently than ProDOS 8. The n and z flags are undefined here; under ProDOS 8, they are set according to the value in the accumulator. Here the caller may check the c flag to see if an error has occurred; under ProDOS 8, both the c and z flags determine error status.

Comparison with the ProDOS 8 call method

With the exceptions noted in Chapter 1, ProDOS 16 provides an identical call for each ProDOS 8 system call. The ProDOS 16 call performs exactly the same function as its ProDOS 8 equivalent, but it is in a format that fits the Apple IIGS environment:

- As in ProDOS 8, the system call is issued through a subroutine jump to a fixed system entry point. However, the jump instruction is a JSL rather than a JSR, and it is to a location in bank $E1, rather than bank $00.
- The parameter block pointer in the system call is 4 bytes long rather than 2, so the parameter block can be anywhere in memory.
- All memory pointer fields within the parameter block are also 4 bytes long, so they can reference data anywhere in memory.
- All 1-byte parameters are extended to 1 word in length, for efficient manipulation in 16-bit processor mode.
- All file-position (such as EOF) and block-specification (such as block number or block count) fields in the parameter block are 4 bytes long, in anticipation of future **guest file systems** that may support files larger than 16 Mb or volumes larger than 32 Mb.

❖ *Note:* Although only 3 bytes are needed for memory pointers and block numbers in the Apple IIGS, 4-byte pointers are used for ease of programming. The high-order byte in each case is reserved and must be zero.

The ProDOS 16 Exerciser

To help you learn to make ProDOS 16 calls, there is a small program called the *ProDOS 16 Exerciser*, on a disk included with this manual. It allows you to execute system calls from a menu, and examine the results of your calls. It has a hexadecimal memory editor for reviewing and altering the contents of memory buffers, and it includes a catalog command.

When you use the *Exerciser* to make a ProDOS 16 call, you first request the call by its call number and then specify its parameter list, just as if you were coding the call in a program. The call is executed when you press Return. You may then use the memory editor or catalog command to examine the results of your call.

Instructions for using the ProDOS 16 *Exerciser* program are in Appendix C.

Format for system call descriptions

The following five chapters list and describe all ProDOS 16 operating system functions that may be called by an application.

Each description includes these elements:

- the function's name and call number
- a short explanation of its use
- a diagram of its required parameter block
- a detailed description of all parameters in the parameter block
- a list of all possible operating system error messages.

The parameter block diagram accompanying each call's description is a simplified representation of the parameter block in memory. The width of the diagram represents one byte; the numbers down the left side represent byte offsets from the base address of the parameter block. Each parameter field is further identified as containing a value, result, or pointer.

The detailed parameter description that follows the diagram has the following headings:

- **Offset:** The position of the parameter (relative to the block's base address)
- **Label:** The suggested assembly-language label for the parameter
- **Description:** Detailed information on the parameter, including:

  parameter name: The full name of the parameter.

  size and type: The size of the parameter (word or long word), and its classification (value, result, or pointer). A word is 2 bytes; a long word is 4 bytes.

  range of values: The permissible range of values of the parameter. A parameter may have a range much smaller than its size in bytes.

  Any additional explanatory information on the parameter follows.

Chapter 9

File Housekeeping Calls

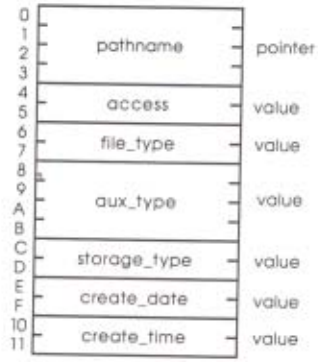
These calls might also be called "closed-file" calls; they are made to get and set information about files that need not be open when the calls are made. They do not alter the *contents* of the files they access.

The ProDOS 16 file housekeeping calls are described in this order:

Number	Function	Purpose
$01	CREATE	creates a new file
$02	DESTROY	deletes a file
$04	CHANGE_PATH	changes a file's pathname
$05	SET_FILE_INFO	assigns attributes to a file
$06	GET_FILE_INFO	returns a file's attributes
$08	VOLUME	returns the volume on a device
$09	SET_PREFIX	assigns a pathname prefix
$0A	GET_PREFIX	returns a pathname prefix
$0B	CLEAR_BACKUP_BIT	zeroes a file's backup attribute

CREATE ($01)

Every disk file except the volume directory file (and any Apple II Pascal region on a partitioned disk) must be created with this call. It establishes a new directory entry for an empty file.



CREATE ($01)
Parameter block

Parameter description

Offset	Label	Description
$00–$03	pathname	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname of the file to create.

Parameter description (continued)

Offset	Label	Description
$04–$05	access	**parameter name:** access **size and type:** word value (high-order byte zero) **range of values:** $0000–$00E3 with exceptions A word whose low-order byte determines how the file may be accessed. The access byte's format is

Bit:	7	6	5	4	3	2	1	0
Value:	D	RN	B	reserved			W	R

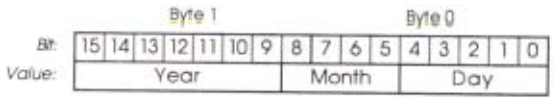
where D = destroy-enable bit
RN = rename-enable bit
B = backup-needed bit
W = write-enable bit
R = read-enable bit

and for each bit, 1 = enabled, 0 = disabled. Bits 2 through 4 are reserved and must always be set to zero (disabled). The most typical setting for the access byte is $C3 (11000011).

Offset	Label	Description
$06–$07	file_type	**parameter name:** file type **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF A number that categorizes the file by its contents (such as text file, binary file, ProDOS 16 application). Currently defined file types are listed in Appendix A.
$08–$0B	aux_type	**parameter name:** auxiliary type **size and type:** long word value (high-order word zero) **range of values:** $0000 0000–$0000 FFFF A number that indicates additional attributes for certain file types. Example uses of the auxiliary type field are given in Appendix A.

Parameter description (continued)

Offset	Label	Description
$0C–$0D	storage_type	**parameter name:** storage type **size and type:** word value/result (high-order byte zero) **range of values:** $0000–$000D with exceptions A number that describes the logical organization of the file (see Appendix A): $00 = inactive entry $01 = seedling file $02 = sapling file $03 = tree file $04 = Apple II Pascal region on a partitioned disk $0D = directory file $01 and $0D are the most typical input values for this field in the CREATE call; any value in the range $00 through $03 is automatically converted to an input (and output) of $01. ❖ *Note:* $0E and $0F are not valid storage types; they are subdirectory and volume key block identifiers.
$0E–$0F	create_date	**parameter name:** creation date **size and type:** word value **range of values:** limited range The date on which the file was created. Its format is



	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	Year							Month				Day				

If the value in this field is zero, ProDOS 16 supplies the date obtained from the system clock.

Parameter description (continued)

Offset	Label	Description
$10–$11	create_time	**parameter name:** creation time **size and type:** word value **range of values:** limited range The time at which the file was created. Its format is

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	0	0	0	Hour					0	0	Minute					

*If the value in this field is zero, ProDOS 16 supplies the time obtained from the system clock.

Possible ProDOS 16 errors

$07	ProDOS is busy
$10	Device not found
$27	I/O error
$2B	Disk write-protected
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$47	Duplicate pathname
$48	Volume full
$49	Volume directory full
$4B	Unsupported storage type
$52	Unsupported volume type
$53	Invalid parameter
$58	Not a block device
$5A	Block number out of range

DESTROY ($02)

This function deletes the file specified by pathname. It removes the file's entry from the directory that owns it and returns the file's blocks to the volume bit map.

Volume directory files, files with unrecognized storage types (other than $01, $02, $03, or $0D), and open files cannot be destroyed. Subdirectory files must be empty before they can be destroyed.

❖ *Note:* When a file is destroyed, any index blocks it contains are inverted—that is, the first half of the block and the second half swap positions. That reverses the order of the bytes in all pointers the block contains. Disk scavenging programs can use this information to help recover accidentally deleted files. See Appendix A for a description of index block structure.

DESTROY ($02)
Parameter block

Parameter description

Offset	Label	Description
$00–$03	pathname	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname of the file to delete.

Possible ProDOS 16 errors

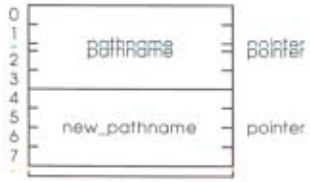
$07	ProDOS is busy
$10	Device not found
$27	I/O error
$2B	Disk write-protected
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type
$4E	Access: file not destroy-enabled
$50	File is open
$52	Unsupported volume type
$58	Not a block device
$5A	Block number out of range

CHANGE_PATH ($04)

This function performs an intravolume file move. It moves a file's directory entry from one subdirectory to another within the same volume (the file itself is never moved). The specified pathname and new pathname may be either full or partial pathnames in the same volume. See Chapter 5 for an explanation of partial pathnames.

To rename a volume, the specified pathname and new pathname must be volume names only.

If the two pathnames are identical except for the rightmost file name (that is, if both the old and new names are in the same subdirectory), this call produces the same result as the RENAME call in ProDOS 8.



CHANGE_PATH ($04)
Parameter block

❖ *Note:* In initial releases of ProDOS 16, CHANGE_PATH is restricted to a filename change only—that is, it is functionally identical to the RENAME call in ProDOS 8.

Parameter description

Offset	Label	Description
$00–$03	pathname	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the file's present pathname.
$04–$07	new_pathname	**parameter name:** new pathname **size and type:** long word pointer (high-order byte zero) **range of values:** 0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the file's new pathname.

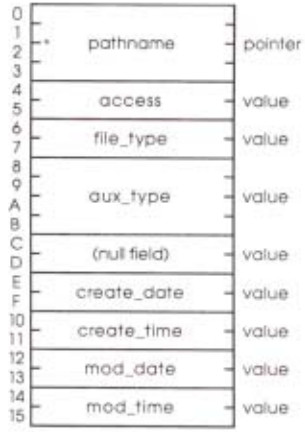
Possible ProDOS 16 errors

$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$47	Duplicate pathname
$4A	Version error
$4B	Unsupported storage type
$4E	Access: file not rename-enabled
$50	File is open
$52	Unsupported volume type
$57	Duplicate volume
$58	Not a block device

SET_FILE_INFO ($05)

This function modifies the information in the specified file's directory entry. The call can be made whether the file is open or closed; however, any changed access attributes are not recognized by an open file until the next time the file is opened. In other words, this call does not modify the accessibility of memory-resident information.

❖ *Note:* Current versions of ProDOS 16 ignore input values in the `create_date` and `create_time` fields of this function.



SET_FILE_INFO ($05)
Parameter block

Parameter description

Offset	Label	Description
$00–$03	pathname	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the file's pathname.
$04–$05	access	**parameter name:** access **size and type:** word value (high-order byte zero) **range of values:** $0000–$00E3 with exceptions A word whose low-order byte determines how the file may be accessed. The access byte's format is Bit: 7 6 5 4 3 2 1 0 Value: D RN B reserved W R where D = destroy-enable bit RN = rename-enable bit B = backup-needed bit W = write-enable bit R = read-enable bit and for each bit, 1 = enabled, 0 = disabled. Bits 2 through 4 are reserved and must always be set to zero (disabled). The most typical setting for the access byte is $C3 (11000011).
$06–$07	file_type	**parameter name:** file type **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF A number that categorizes the file by its contents (such as text file, binary file, ProDOS 16 application). Currently defined file types are listed in Appendix A.
$08–$0B	aux_type	**parameter name:** auxiliary type **size and type:** long word value (high-order word zero) **range of values:** $0000 0000–$0000 FFFF A number that indicates additional attributes for certain file types. Example uses of the auxiliary type field are given in Appendix A.

Parameter description (continued)

Offset	Label	Description
$0C–$0D	(null field)	**parameter name:** (none) **size and type:** word value **range of values:** (undefined) Values in this field are ignored.
$0E–$0F	create_date	**parameter name:** creation date **size and type:** word value **range of values:** limited range The date on which the file was created. Its format is Byte 1 Byte 0 Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Value: Year Month Day (Values in this field are ignored.)
$10–$11	create_time	**parameter name:** creation time **size and type:** word value **range of values:** limited range The time at which the file was created. Its format is Byte 1 Byte 0 Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Value: 0 0 0 Hour 0 0 Minute (Values in this field are ignored.)
$12–$13	mod_date	**parameter name:** modification date **size and type:** word value **range of values:** limited range The date on which the file was last modified. Its format is identical to the create_date format: Byte 1 Byte 0 Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Value: Year Month Day If the value in this field is zero, ProDOS 16 supplies the date obtained from the system clock.

Parameter description (continued)

Offset	Label	Description
$14–$15	mod_time	**parameter name:** modification time **size and type:** word value **range of values:** limited range The time at which the file was last modified. Its format is identical to the create_time format:

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	0	0	0	Hour					0	0	Minute					

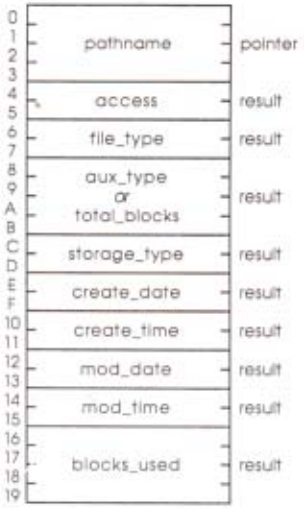
If the value in this field is zero, ProDOS 16 supplies the time obtained from the system clock.

Possible ProDOS 16 errors

$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type
$4E	Access: file not write-enabled
$52	Unsupported volume type
$53	Invalid parameter
$58	Not a block device

GET_FILE_INFO ($06)

This function returns the information that is stored in the specified file's directory entry. The call can be made whether the file is open or closed. However, if you make the SET_FILE_INFO call to change the access byte of an open file, the access information returned by GET_FILE_INFO may not be accurate until the file is closed.



GET_FILE_INFO ($06)
Parameter block

Parameter description

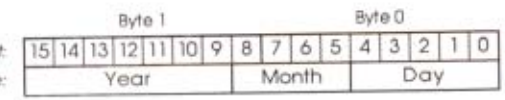
Offset	Label	Description
$00–$03	pathname	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname.
$04–$05	access	**parameter name:** access **size and type:** word result (high-order byte zero) **range of values:** $0000–$00E3 with exceptions A word whose low-order byte determines how the file may be accessed. The access byte's format is Bit: 7 6 5 4 3 2 1 0 Value: D RN B reserved W R where D = destroy-enable bit RN = rename-enable bit B = backup-needed bit W = write-enable bit R = read-enable bit and for each bit, 1 = enabled, 0 = disabled. Bits 2 through 4 are reserved and must always be set to zero (disabled). The most typical setting for the access byte is $C3 (11000011).
$06–$07	file_type	**parameter name:** file type **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF A number that categorizes the file by its contents (such as text file, binary file, ProDOS 16 application). Currently defined file types are listed in Appendix A.

Parameter description (continued)

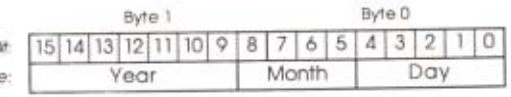
Offset	Label	Description
$08–$0B	aux_type	**parameter name:** auxiliary type **size and type:** long word result (high-order word zero) **range of values:** $0000 0000–$0000 FFFF A number that indicates additional attributes for certain file types. Example uses of the auxiliary type field are given in Appendix A.
	or	
	total_blocks	**parameter name:** total blocks **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF If the call is for a volume directory file, the total number of blocks on the volume is returned in this field.
$0C–$0D	storage_type	**parameter name:** storage type **size and type:** word result (high-order byte zero) **range of values:** $0000–$000D with exceptions A number that describes the logical organization of the file (see Appendix A): $00 = inactive entry $01 = seedling file $02 = sapling file $03 = tree file $04 = UCSD Pascal region on a partitioned disk $0D = directory file ❖ *Note:* $0E and $0F are not valid storage types; they are subdirectory and volume key block identifiers.
$0E–$0F	create_date	**parameter name:** creation date **size and type:** word result **range of values:** limited range The date on which the file was created. Its format is Byte 1 Byte 0 Bit: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Value: Year Month Day

Parameter description (continued)

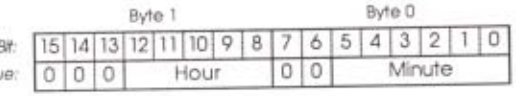
Offset	Label	Description
$10–$11	create_time	**parameter name:** creation time **size and type:** word result **range of values:** limited range The time at which the file was created. Its format is



$12–$13	mod_date	**parameter name:** modification date **size and type:** word result **range of values:** limited range The date on which the file was last modified. Its format is identical to the create_date format:



$14–$15	mod_time	**parameter name:** modification time **size and type:** word result **range of values:** limited range The time at which the file was last modified. Its format is identical to the create_time format:

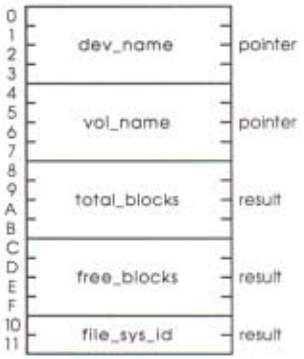


$16–$19	blocks_used	**parameter name:** blocks used **size and type:** long word result **range of values:** $0000 0000–$FFFF FFFF The total number of blocks used by the file. It equals the value of the blocks_used parameter in the file's directory entry. *or* The total number of blocks used by all files on the volume (if the call is for a volume directory).

Possible ProDOS 16 errors

$07	ProDOS is busy
$27	I/O error
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type
$52	Unsupported volume type
$53	Invalid parameter
$58	Not a block device

VOLUME ($08)



VOLUME ($08)
Parameter block

When given the name of a device, this function returns:

- the name of the volume that occupies that device
- the total number of blocks on the volume
- the current number of free (unallocated) blocks on the volume
- the file system identification number of the volume

The volume name is returned with a leading slash (/).

To generate a list of all mounted volumes (equivalent to calling ON_LINE in ProDOS 8 with a unit number of zero), call VOLUME repeatedly with successive device names (.D1, .D2, and so on). When there are no more online volumes to name, ProDOS 16 returns error $11 (Invalid device request).

❖ *Note:* In certain cases (for example, when polling Disk II drives) ProDOS 16 cannot detect the difference between an empty device and a nonexistent device. It may therefore assign a device name where there is no device connected, just to make sure it hasn't skipped over an empty device. Because of this, in making VOLUME calls, you may occasionally find that there are more "valid" device names than there are devices on line.

Parameter description

Offset	Label	Description
$00–$03	dev_name	**parameter name:** device name **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the device name.
$04–$07	vol_name	**parameter name:** volume name **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the volume name (including a leading slash).

Parameter description (continued)

Offset	Label	Description
$08–$0B	total_blocks	**parameter name:** total blocks **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The total number of blocks the volume contains.
$0C–$0F	free_blocks	**parameter name:** free blocks **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The number of free (unallocated) blocks in the volume.
$10–$11	file_sys_id	**parameter name:** file system ID **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF A word whose low-order byte identifies the file system to which the specified file or volume belongs. The currently defined file system identification numbers include 0 = (reserved) 1 = ProDOS/SOS 2 = DOS 3.3 3 = DOS 3.2, 3.1 4 = Apple II Pascal 5 = Macintosh 6 = Macintosh (HFS) 7 = LISA® 8 = Apple CP/M 9-255 = (reserved)

Possible ProDOS 16 errors

$07	ProDOS is busy
$10	Device not found
$11	Invalid device request
$27	I/O error
$28	No device connected
$2E	Disk switched: files open
$2F	Device not on line
$40	Invalid pathname
$45	Volume not found
$4A	Version error
$52	Unsupported volume type
$55	Volume control block full
$57	Duplicate volume
$58	Not a block device

SET_PREFIX ($09)

This function assigns any of 8 prefix numbers to the pathname indicated by the pointer `prefix`. A prefix number consists of a digit followed by a slash: `0/`, `1/`, `2/`,..., `7/`. When an application starts, the prefixes have default values that depend on the manner in which the program was launched. See Chapter 5.

The input pathname to this call may be

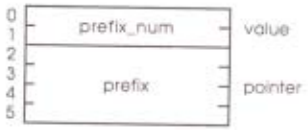
- a full pathname.
- a partial pathname with a prefix number. The trailing slash on the prefix number is optional.
- a partial pathname with the special prefix number */ (asterisk-slash), which means "boot volume name." The trailing slash is optional.
- a partial pathname without a prefix number. In this case ProDOS 16 does *not* attach the default prefix (number 0/). Instead, it *appends* the input pathname to the prefix specified in the `prefix_num` field.

❖ *Note:* This method can be used to append a partial pathname to an *existing* prefix only. If the specified prefix is presently null, error $40 (invalid pathname syntax) is returned.

Specifying a pathname whose length byte is zero, or whose syntax is otherwise illegal, sets the designated prefix to null (unassigned).

❖ *Note:* ProDOS 16 does *not* check to make sure that the designated volume is on line when you specify a prefix; it only checks the pathname string for correct syntax.

The boot volume prefix (*/) cannot be changed through this call.



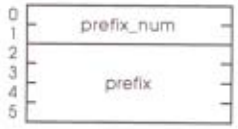
SET_PREFIX ($09)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	prefix_num	**parameter name:** prefix number **size and type:** word value **range of values:** $0000–$0007 One of the 8 prefix numbers, in binary (without a terminating slash).
$02–$05	prefix	**parameter name:** prefix **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing a directory pathname.

Possible ProDOS 16 errors

$07	ProDOS is busy
$40	Invalid pathname syntax
$53	Parameter out of range



GET_PREFIX ($0A)
Parameter block

GET_PREFIX ($0A)

This function returns any of the current prefixes (specified by number), placing it in the buffer pointed to by *prefix*. The returned prefix is bracketed by slashes (such as `/APPLE/` or `/APPLE/BYTES/`). If the requested prefix has been set to null (see `SET_PREFIX`), a count of zero is returned as the length byte in the prefix buffer.

The boot volume prefix (`*/`) cannot be returned by this call. Instead, use `GET_BOOT_VOL` to find the boot volume's name.

Parameter description

Offset	Label	Description
$00–$01	prefix_num	**parameter name:** prefix number **size and type:** word value **range of values:** $0000–$0007 One of the 8 prefix numbers, in binary (without a terminating slash).
$02–$05	prefix	**parameter name:** prefix **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer, in which ProDOS 16 places a length byte followed by an ASCII string representing a directory pathname.

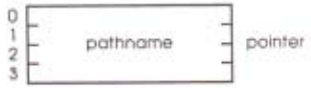
Possible ProDOS 16 errors

$07	ProDOS is busy
$53	Parameter out of range

CLEAR_BACKUP_BIT ($0B)

This is the only call that will clear the backup bit in a file's access byte. Once cleared, the bit indicates that the file has not been altered since the last backup. ProDOS 16 automatically resets the backup bit every time a file is altered.

Important Only disk backup programs should use this function!



CLEAR_BACKUP_BIT ($0B)
Parameter block

Parameter description

Offset	Label	Description
$00–$03	pathname	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the file's pathname.

Possible ProDOS 16 errors

$07	ProDOS is busy
$40	Invalid pathname syntax
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$52	Unsupported volume type
$58	Not a block device

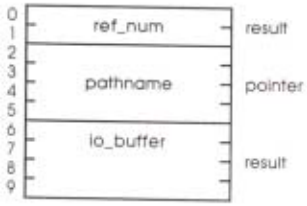
Chapter 10

File Access Calls

These might be called "open-file" calls. They are made to access and change the information *within* files, and therefore in most cases the files must be open before the calls can be made.

The ProDOS 16 file access calls are described in the following order:

Number	Function	Purpose
$10	OPEN	prepares file for access
$11	NEWLINE	enables newline read mode
$12	READ	transfers data from file
$13	WRITE	transfers data to file
$14	CLOSE	ends access to file
$15	FLUSH	empties I/O buffer to file
$16	SET_MARK	sets current position in file
$17	GET_MARK)	returns current position in file
$18	SET_EOF	sets size of file
$19	GET_EOF	returns size of file
$1A	SET_LEVEL	sets system file level
$1B	GET_LEVEL	returns system file level



OPEN ($10)
Parameter block

OPEN ($10)

This function prepares a file to be read from or written to. It creates a file control block (FCB) that keeps track of the current characteristics of the file specified by pathname. It sets the current position in the file (Mark) to zero, and returns a reference number (*ref_num*) for the file; subsequent file access calls must refer to the file by its reference number. It also returns a memory handle to a 1024-byte I/O buffer used by ProDOS 16 for reading from and writing to the file.

Up to 8 files may be open simultaneously.

❖ *Note:* Normally, attempting to open a file that is already open causes an error ($50). However, if a file is not *write-enabled*, it may be opened more than once.

Parameter description

Offset	Label	Description
$00–$01	ref_num	**parameter name:** reference number **size and type:** word result (high-order byte zero) **range of values:** $0001–$00FF An identifying number assigned to the file by ProDOS 16. It is used in place of the pathname in all subsequent file access calls.
$02–$05	pathname	**parameter name:** pathname **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the pathname of the file to open.
$06–$09	io_buffer	**parameter name:** I/O buffer **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF A memory handle. It points to a location where the address of the I/O buffer allocated by ProDOS 16 is stored.

Possible ProDOS 16 errors

$07	ProDOS is busy
$27	I/O error
$40	Invalid pathname syntax
$42	File control block table full
$44	Path not found
$45	Volume not found
$46	File not found
$4A	Version error
$4B	Unsupported storage type
$50	File is open
$52	Unsupported volume type

NEWLINE ($11)

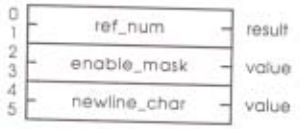
This function enables or disables the **newline** read mode for an open file. When newline is disabled, a READ call (described next) terminates only when the requested number of characters has been read (unless the end of the file is encountered first). When newline is enabled, the READ will also terminate when a newline character (as defined in the parameter block) is read.

When a READ call is made and newline mode is enabled,

1. Each character read in is first transferred to the user's data buffer.
2. The character is ANDed with the low-order byte of the newline enable mask (specified in the NEWLINE call's parameter block).
3. The result is compared with the low-order byte of the newline character.
4. If there is a match, the read is terminated.

The enable mask is typically used to mask off unwanted bits in the character that is read in. For example, if the mask value is $7F (binary 0111 1111), a newline character will be correctly matched whether or not its high bit is set. If the mask value is $FF (1111 1111), the character will pass through the AND operation unchanged.

Newline read mode is disabled by setting the enable mask to $0000.



NEWLINE ($11)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	ref_num	**parameter name:** reference number **size and type:** word result (high-order byte zero) **range of values:** $0001–$00FF The identifying number assigned to the file by the OPEN function.
$02–$03	enable_mask	**parameter name:** enable mask **size and type:** word value (high-order byte zero **range of values:** $0000–$00FF The current character is ANDed with the low order byte of this word.
$04–$05	newline_char	**parameter name:** newline character **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF Whatever character occupies the low-order byte of this field is defined as the newline character.

Possible ProDOS 16 errors

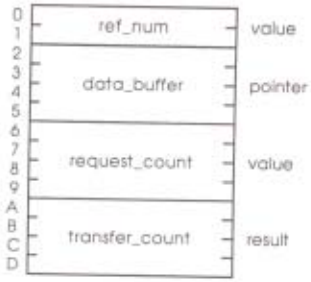
$07	ProDOS is busy
$43	Invalid reference number

READ ($12)

When called, this function attempts to transfer the requested number of bytes (starting at the current position of the file specified by *ref_num*) into the buffer pointed to by *data_buffer*. When finished, the function returns the number of bytes actually transferred.

If, during a read, the end-of-file is reached before *request_count* bytes have been read, `transfer_count` is set to the number of bytes transferred. If newline mode is enabled and a newline character is encountered before *request_count* bytes have been read, `transfer_count` is set to the number of bytes transferred (including the newline byte).

No more than 16,777,215 ($FF FF FF) bytes may be read in a single call.



READ ($12)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	ref_num	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0001–$00FF The identifying number assigned to the file by the OPEN function.
$02–$05	data_buffer	**parameter name:** data buffer **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer should be large enough to hold the requested data.
$06–$09	request_count	**parameter name:** request count **size and type:** long word value (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The number of bytes to be transferred.
$0A–$0D	transfer_count	**parameter name:** transfer count **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The actual number of bytes transferred.

Possible ProDOS 16 errors

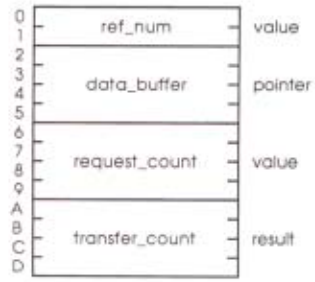
$07	ProDOS is busy
$27	I/O error
$43	Invalid reference number
$4C	EOF encountered (Out of data)
$4E	Access: file not read-enabled

WRITE ($13)

When called, this function attempts to transfer the specified number of bytes from the buffer pointed to by *data_buffer* to the file specified by *ref_num* (starting at the current position in the file). When finished, the function returns the number of bytes actually transferred.

After a write, the current file position (Mark) is increased by the transfer count. If necessary, the end-of-file (EOF) is extended to accomodate the new data.

No more than 16,777,216 ($FF FF FF) bytes may be written in a single call.



WRITE ($13)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	ref_num	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0001–$00FF The identifying number assigned to the file by the OPEN function.
$02–$05	data_buffer	**parameter name:** data buffer **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer should be large enough to hold the requested data.
$06–$09	request_count	**parameter name:** request count **size and type:** long word value (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The number of bytes to be transferred.
$0A–$0D	transfer_count	**parameter name:** transfer count **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The actual number of bytes transferred.

Possible ProDOS 16 errors

$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$43	Invalid reference number
$48	Volume full
$4E	Access: file not write-enabled
$5A	Block number out of range

CLOSE ($14)

This function is called to release all resources used by an open file and terminate further access to it. The file control block (FCB) is released; if necessary, the file's I/O buffer is emptied (written to disk) and the directory entry for the file is updated. Once a file is closed, any subsequent calls using its ref_num will fail (until that number is assigned to another open file).

If the specified ref_num is 0, all open files at or above the current file level (see SET_LEVEL and GET_LEVEL calls) are closed. For example, if files are open at levels 0, 1, and 2 and you have set the current level to 1, a CLOSE call with ref_num set to 0 will close all files at levels 1 and 2, but leave files at level 0 open.

CLOSE ($14)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	ref_num	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The identifying number assigned to the file by the OPEN function.

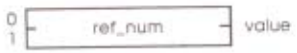
Possible ProDOS 16 errors

$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$43	Invalid reference number
$5A	Block number out of range

FLUSH ($15)

This function is called to empty an open file's buffer and update its directory. If `ref_num` is zero, all open files are flushed.

❖ *Note:* Current versions of ProDOS 16 ignore *ref_num* in this call. The `FLUSH` call flushes all open files.



FLUSH ($15)
Parameter block

Parameter description

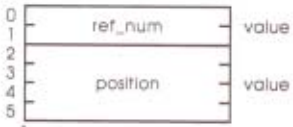
Offset	Label	Description
$00–$01	`ref_num`	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The identifying number assigned to the file by the `OPEN` function.

Possible ProDOS 16 errors

$07	ProDOS is busy
$27	I/O error
$2B	Disk write-protected
$43	Invalid reference number
$48	Volume full
$5A	Block number out of range

SET_MARK ($16)

For the specified open file, this function sets the current position (Mark, the position at which subsequent reading and writing will occur) to the point specified by the position parameter. The value of the current position may not exceed EOF (end-of-file; the size of the file in bytes).



SET_MARK ($16)
Parameter block

Parameter description

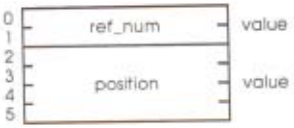
Offset	Label	Description
$00–$01	`ref_num`	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0001–$00FF The identifying number assigned to the file by the `OPEN` function.
$02–$05	`position`	**parameter name:** position **size and type:** long word value (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The value assigned to Mark. It is the position, in bytes relative to the beginning of the file, at which the next read or write will occur.

Possible ProDOS 16 errors

$07	ProDOS is busy
$27	I/O error
$43	Invalid reference number
$4D	Position out of range
$5A	Block number out of range

GET_MARK ($17)

This function returns the current position (Mark, the position at which subsequent reading and writing will occur) for the specified open file.



GET_MARK ($17)
Parameter block

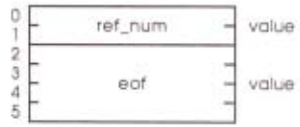
Parameter description

Offset	Label	Description
$00–$01	ref_num	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0001–$00FF The identifying number assigned to the file by the OPEN function.
$02–$05	position	**parameter name:** position **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The current value of Mark. It is the position, in bytes relative to the beginning of the file, at which the next read or write will occur.

Possible ProDOS 16 errors

$07	ProDOS is busy
$43	Invalid reference number

SET_EOF ($18)



SET_EOF ($18)
Parameter block

For the specified file, this function sets its logical size (in bytes) to the value specified by EOF (end-of-file). If the specified EOF is less than the current EOF, then disk blocks past the new EOF are released to the system and index-block pointers to those blocks are zeroed. However, if the specified EOF is equal to or greater than the current EOF, no new blocks are allocated until data are actually written to them.

The value of EOF cannot be changed unless the file is write-enabled.

Parameter description

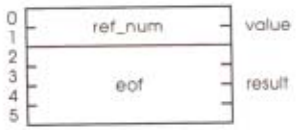
Offset	Label	Description
$00–$01	ref_num	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0001–$00FF The identifying number assigned to the file by the OPEN function.
$04–$07	eof	**parameter name:** end-of-file **size and type:** long word value (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The specified logical size of the file. It represents the total number of bytes that may be read from the file.

Possible ProDOS 16 errors

$07	ProDOS is busy
$27	I/O error
$43	Invalid reference number
$4D	Position out of range
$4E	Access: file not write-enabled
$5A	Block number out of range

GET_EOF ($19)

For the specified open file, this function returns its logical size, or EOF (end-of-file; the number of bytes that can be read from it).



GET_EOF ($19)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	ref_num	**parameter name:** reference number **size and type:** word value (high-order byte zero) **range of values:** $0001–$00FF The identifying number assigned to the file by the OPEN function.
$04–$07	eof	**parameter name:** end-of-file **size and type:** long word result (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The current logical size of the file. It represents the total number of bytes that may be read from the file.

Possible ProDOS 16 errors

$07	ProDOS is busy
$43	Invalid reference number

SET_LEVEL ($1A)

This function sets the current value of the system file level (see Chapter 2). All subsequent OPEN calls will assign this level to the files opened. All subsequent CLOSE calls for *multiple* files (that is, those calls using a specified ref_num of 0) will be effective only on those files that were opened when the system level was greater than or equal to the new level.

The range of legal system level values is $0000–$00FF. The file level initially defaults to zero.

SET_LEVEL ($1A)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	level	**parameter name:** system file level **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The specified value of the system file level.

Possible ProDOS 16 errors

$07	ProDOS is busy
$59	Invalid file level

GET_LEVEL ($1B)

This function returns the current value of the system file level (see Chapter 2). All subsequent OPEN calls will assign this level to the files opened. All subsequent CLOSE calls for *multiple* files (that is, those calls using a specified ref_num of 0) will be effective only on those files that were opened when the system level was greater than or equal to its current level.

GET_LEVEL ($1B)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	level	**parameter name:** system file level **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF The current value of the system file level.

Possible ProDOS 16 errors

$07	ProDOS is busy

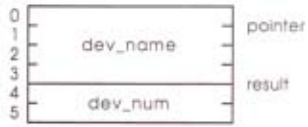
Chapter 11

Device Calls

Device calls access storage devices directly, rather than through the logical structure of the volumes or files on them.

The ProDOS 16 device calls are described in the following order:

Number	Function	Purpose
$20	GET_DEV_NUM	returns a device's number
$21	GET_LAST_DEV	returns the last device accessed
$22	READ_BLOCK	transfers 512 bytes from a device
$23	WRITE_BLOCK	transfers 512 bytes to a device
$24	FORMAT	formats a volume in a device



GET_DEV_NUM ($20)
Parameter block

GET_DEV_NUM ($20)

For the device specified by name or by the name of the volume mounted on it, this function returns its device number. All other device calls (except for FORMAT) must refer to the device by its number.

Device numbers are assigned by ProDOS 16 at system startup (boot) time. They are consecutive integers, assigned in the order in which ProDOS 16 polls external devices (see Chapter 4).

❖ *Note:* Because a device may hold different volumes and because volumes may be switched among devices, the device number returned for a particular volume name may change. Likewise, the volume name associated with a particular device number may change.

Parameter description

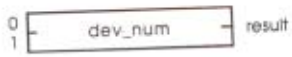
Offset	Label	Description
$00–$03	dev_name	**parameter name:** device name/volume name **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the device name or the volume name.
$04–$05	dev_num	**parameter name:** device number **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF The device's reference number, to be used in other device calls.

Possible ProDOS 16 errors

$07	ProDOS is busy
$10	Device not found
$11	Invalid device request
$40	Invalid device name syntax
$45	Volume not found

GET_LAST_DEV ($21)

This function returns the device number of the last device accessed. The *last device accessed* is the last device to which a command was directed that caused a read or write to occur.



GET_LAST_DEV ($21)

Parameter description

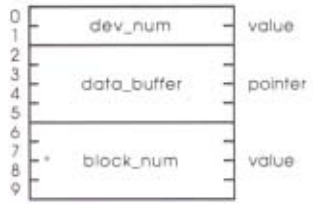
Offset	Label	Description
$00–$01	dev_num	**parameter name:** device number **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF The device's reference number, to be used in other device calls.

Possible ProDOS 16 errors

$07	ProDOS is busy
$60	Data unavailable

READ_BLOCK ($22)

This function reads one block of information from a disk device (specified by *dev_num*) into memory starting at the address pointed to by *data_buffer*. The buffer must be at least 512 bytes in length, because existing devices define a block as 512 bytes.



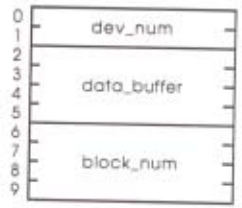
READ_BLOCK ($22)
Parameter block

Parameter description

Offset	Label	Description
$00–$01	dev_num	**parameter name:** device number **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The device's reference number, as returned by GET_DEV_NUM.
$02–$05	data_buffer	**parameter name:** data buffer **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer that will hold the data to be read in.
$06–$09	block_num	**parameter name:** block number **size and type:** long word value (high-order word zero) **range of values:** $0000 0000–$0000 FFFF The number of the block to be read in.

Possible ProDOS 16 errors

$07	ProDOS is busy
$11	Invalid device request
$27	I/O error
$28	No device connected
$2F	Device not on line
$53	Parameter out of range



WRITE_BLOCK ($23)
Parameter block

WRITE_BLOCK ($23)

This function transfers one block of data from the memory buffer pointed to by *data_buffer* to the disk device specified by *dev_num*. The block is placed in the specified logical block of the volume occupying that device. For currently defined devices, the data buffer must be at least 512 bytes long.

Parameter description

Offset	Label	Description
$00–$01	`dev_num`	**parameter name:** device number **size and type:** word value (high-order byte zero) **range of values:** $0000–$00FF The device's reference number, as returned by `GET_DEV_NUM`.
$02–$05	`data_buffer`	**parameter name:** data buffer **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer that holds the data to be written.
$06–$09	`block_num`	**parameter name:** block number **size and type:** long word value (high-order word zero) **range of values:** $0000 0000–$0000 FFFF The number of the block to be written to.

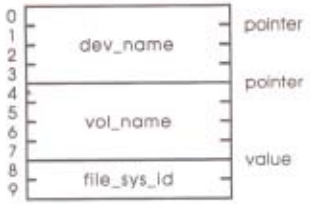
Possible ProDOS 16 errors

$07	ProDOS is busy
$11	Invalid device request
$27	I/O error
$28	No device connected
$2B	Disk write-protected
$2F	Device not on line
$53	Parameter out of range

FORMAT ($24)

This function formats the volume (disk) in the specified (by name) device, giving it the specified volume name. The volume is formatted according to the specified file system ID.

❖ *Note:* Current versions of ProDOS 16 support formatting for the ProDOS/SOS file system only (file system ID = 1). Specifying any other file system will generate error $5D.



FORMAT ($24)
Parameter block

Parameter description

Offset	Label	Description
$00–$03	dev_name	**parameter name:** device name **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the device name.
$04–$07	vol_name	**parameter name:** volume name **size and type:** long word pointer (high-order byte zero) **range of values:** $0000 0000–$00FF FFFF The long word address of a buffer. The buffer contains a length byte followed by an ASCII string representing the volume name (including a leading slash).

$08–$09	file_sys_id	**parameter name:** file system ID **size and type:** word result (high-order byte zero) **range of values:** $0000–$00FF A word whose low-order byte identifies the file system to which the formatted volume belongs. The currently defined file system identification numbers include 0 = (reserved) 1 = ProDOS/SOS 2 = DOS 3.3 3 = DOS 3.2, 3.1 4 = Apple II Pascal 5 = Macintosh 6 = Macintosh (HFS) 7 = LISA 8 = Apple CP/M

Possible ProDOS 16 errors

$07	ProDOS is busy
$10	Device not found
$11	Invalid device request
$27	I/O error
$5D	File system not available

Chapter 12

Environment Calls

These calls deal with the Apple IIGS operating environment, the software and hardware configuration within which applications run. They include calls to start and end ProDOS 16 applications, and to determine pathnames and versions of system software.

The ProDOS 16 environment calls are described in the following order:

Number	Function	Purpose
$27	GET_NAME	returns application filename
$28	GET_BOOT_VOL	returns ProDOS 16 volume name
$29	QUIT	terminates present application
$2A	GET_VERSION	returns ProDOS 16 version