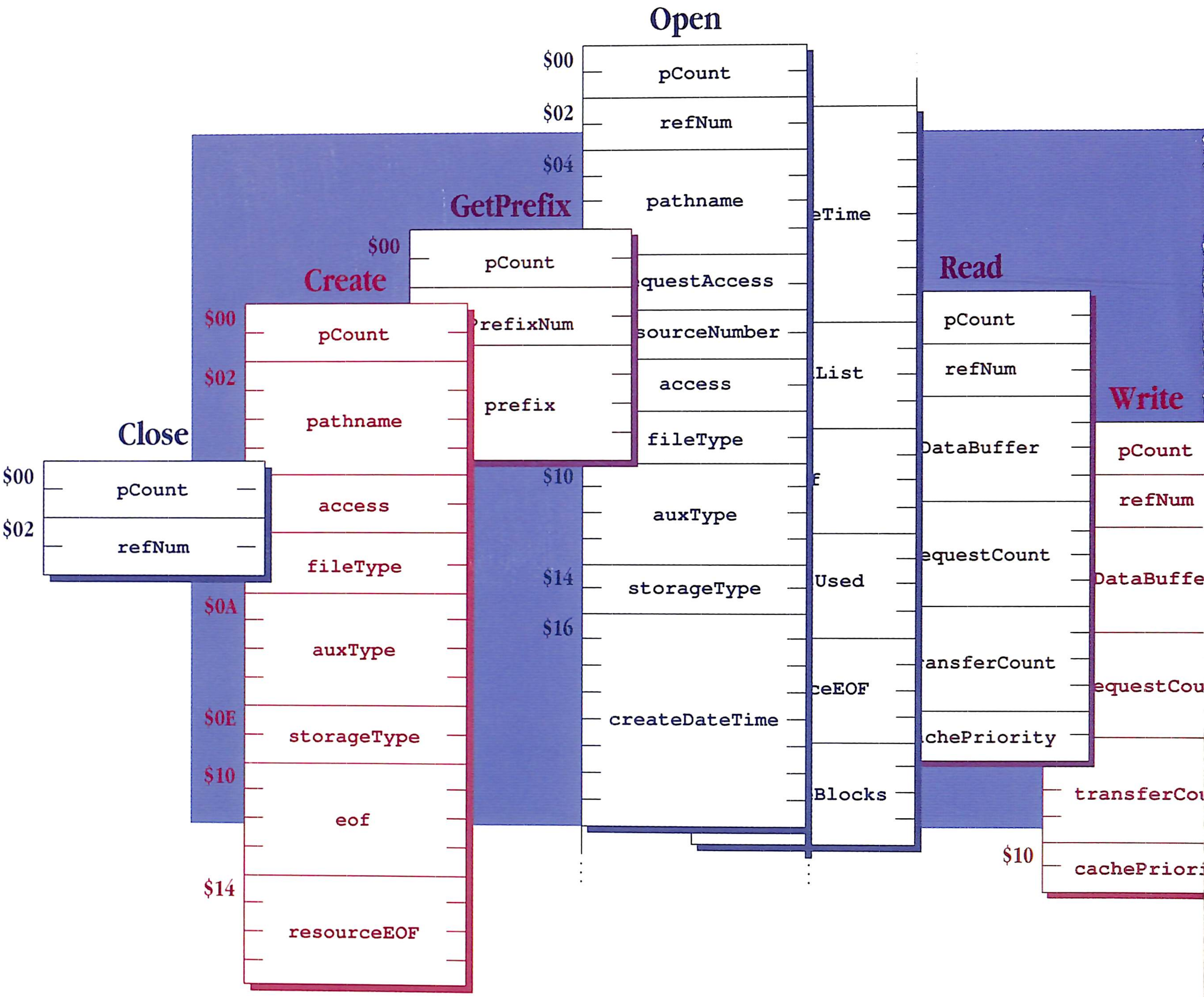
For GS/OS System Software Version 5.0 and Later

Apple IIGS® GS/OS® Reference

by Apple Computer, Inc.



For GS/OS System Software Version 5.0 and Later

Apple IIGS® GS/OS® Reference

Addison-Wesley Publishing Company, Inc.

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

APPLE COMPUTER, INC.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010

Simultaneously published in the United States and Canada.

ISBN 0-201-55020-2
ABCDEFGHIJ-MU-90
First printing, June 1990

Limited Warranty on Media and Replacement

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures and tables

Preface

The *Apple IIGS GS/OS Reference* describes a powerful operating system developed specifically for the Apple IIGS® computer. GS/OS® is characterized by fast execution, easy configurability, multiple-file-system access, character-file access, direct device access, device independence, compatibility with the large GS/OS memory space, and compatibility with standard Apple® II (ProDOS® 8–based) and early Apple IIGS (ProDOS 16–based) applications.

The *Apple IIGS GS/OS Reference* describes how GS/OS gives your application access to the full range of Apple IIGS features.

About this book

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

- any program that creates or accesses files
- a program that catalogs disks or manipulates files
- a stand-alone program that automatically runs when the computer starts up
- a program that loads and runs other programs
- any program using segmented, dynamic code
- an interrupt handler
- a device driver

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

The software described in this book is part of the Apple IIGS system software, versions 5.0 and later. Apple IIGS system software is available from Apple dealers and from the Apple Programmers and Developers Association (APDA™).

◆ *Note:* System software versions earlier than version 4.0 contain ProDOS 16 rather than GS/OS. ProDOS 16 is described in the *Apple IIGS ProDOS 16 Reference.*

How to use this book

This book is primarily a reference tool that describes the application interface, the high-level parts of GS/OS that your application calls in order to access files or to modify the operating environment.

- The introduction describes GS/OS in general.
- Part I describes how applications interact with GS/OS and documents all application-level GS/OS calls.

- Part II documents the file system translators (FSTs), the software modules that allow your program to access files from many different file systems. Part II lists the application calls each FST supports and documents any differences in call handling from the standard descriptions in Part I.

The principal descriptions of all application-level GS/OS calls (other than device calls) are in Part I. Call descriptions elsewhere in the book document how the call differs from its standard description. Driver calls (low-level device calls used by device drivers) are described in the *GS/OS Device Driver Reference.*

If you are writing a typical application, this book is probably all you will need. If you need to access devices directly or if you are writing a device driver, you will need the *GS/OS Device Driver Reference.*

This manual does not explain 65816 assembly language. Refer to the *Apple IIGS Programmer's Workshop Assembler Reference* or the *MPW IIGS Assembler Reference* for information on Apple IIGS assembly-language programming.

This manual does not give a detailed description of ProDOS 8, the operating system for standard Apple II computers (Apple II Plus, Apple IIe, Apple IIc). For detailed information on ProDOS 8, see the *ProDOS 8 Technical Reference Manual.*

Other materials you'll need

In order to write Apple IIGS programs that run under GS/OS, you need an Apple IIGS computer and development-environment software. Furthermore, you need at least some of the reference materials listed later in the Preface under "Roadmap to the Apple IIGS Technical Manuals." In particular, if you intend to write desktop-style applications or desk accessories, which make use of the Apple IIGS Toolbox, you will need the *Apple IIGS Toolbox Reference.*

The GS/OS Exerciser can be useful for experimenting with GS/OS calls.

◆ *Note:* The GS/OS Exerciser is available through the Apple Programmers and Developers Association (APDA).

Visual cues

Certain typographical conventions in this manual provide visual cues alerting you, for example, to the introduction of a new term or to especially important information.

When a new term is introduced, it is printed in **boldface.** This lets you know that the term is defined at that place in the text and that there is an entry for it in the glossary.

Special messages are marked as follows:

◆ *Note:* Text set off in this manner—with the word *Note*—presents extra information or points to remember.

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

Terminology

This manual may define certain terms, such as *Apple II* and *ProDOS,* somewhat differently from what you are used to. Please note the following definitions:

Apple II: A general term for the Apple II family of computers, especially those that may use ProDOS 8 or ProDOS 16 as an operating system. It includes the 64 KB 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 earlier 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 is also a file system developed to operate with the ProDOS operating systems.

ProDOS 8: The 8-bit ProDOS operating system, through version 1.8, 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: The first 16-bit operating system developed for the Apple IIGS computer. ProDOS 16 is based on ProDOS 8.

GS/OS: A native-mode, 16-bit operating system developed for the Apple IIGS computer. GS/OS replaces ProDOS 16 as the preferred Apple IIGS operating system. GS/OS is the system described in this manual.

Language notation

This manual uses certain conventions in common with Apple IIGS language manuals. Words and symbols that are computer code appear in a monospaced font:

```
            _CallName_C1 parmblock  ;Name of call
            bcs   error             ;handle error if carry set on return
            .
            .
            .
error                               ;code to handle error return
            .
            .
            .
parmblock                           ;parameter block
```

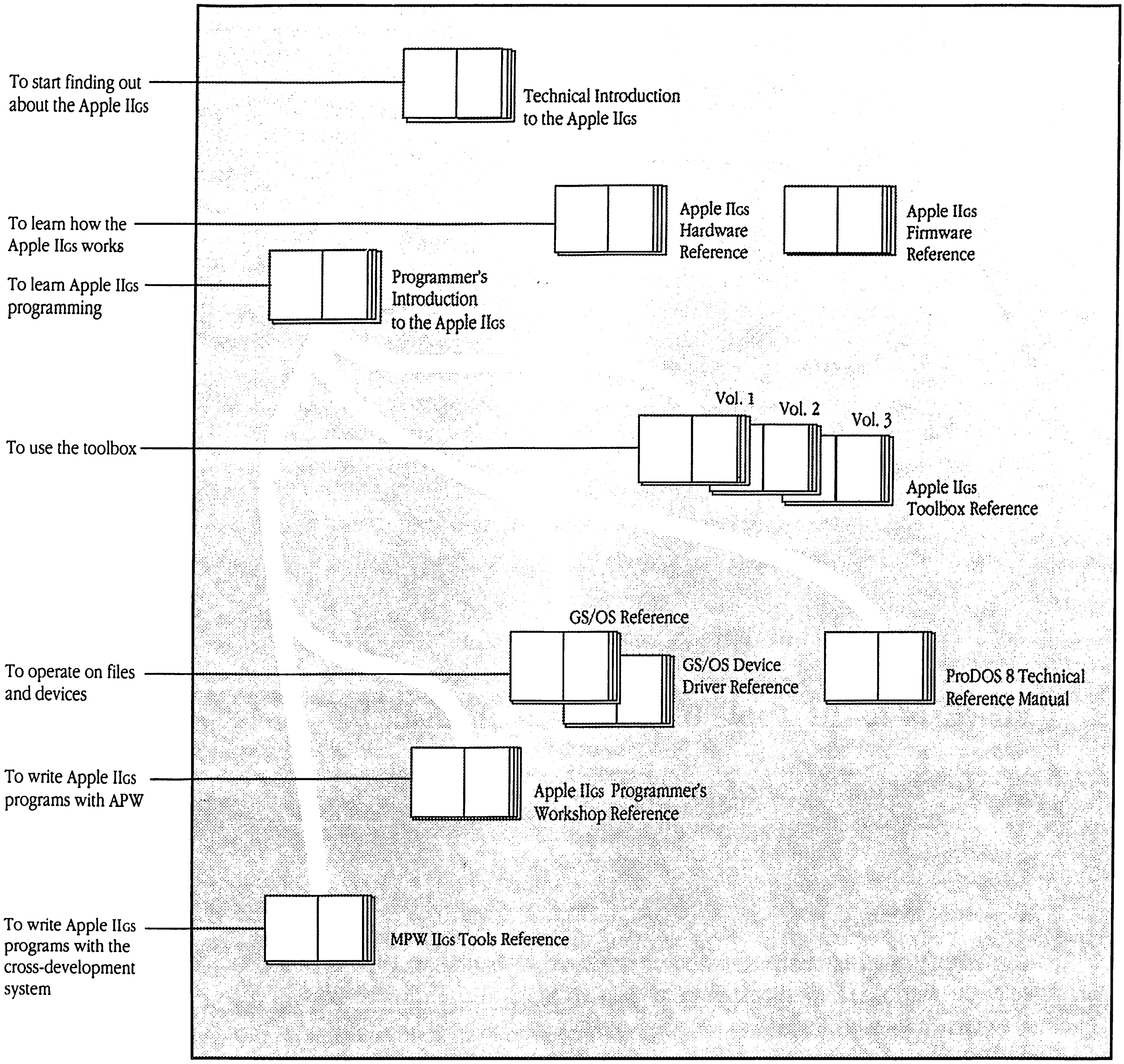
Assembly-language labels, entry points, and filenames that appear in text passages are also printed in a monospaced font. System software functions, except standard GS/OS call names, are printed in a monospaced font in uppercase and lowercase letters (for example, buffTooSmall). The subclass of GS/OS calls that are compatible with ProDOS 16 are printed in all uppercase letters and often include underscore characters (for example, GET_DIR_ENTRY).

Roadmap 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 computer. To describe the Apple IIGS 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 Apple IIGS technical manuals document Apple IIGS hardware, Apple IIGS system software, and two development environments for writing Apple IIGS programs. Figure P-1 is a diagram showing the relationships among the principal manuals; Table P-1 is a complete list of all manuals. Individual descriptions of the manuals follow.

■ **Figure P-1** Roadmap to Apple IIGS technical manuals


To start finding out about the Apple IIGS
Technical Introduction to the Apple IIGS
To learn how the Apple IIGS works
Apple IIGS Hardware Reference
Apple IIGS Firmware Reference
To learn Apple IIGS programming
Programmer's Introduction to the Apple IIGS
To use the toolbox
Vol. 1
Vol. 2
Vol. 3
Apple IIGS Toolbox Reference
GS/OS Reference
To operate on files and devices
GS/OS Device Driver Reference
ProDOS 8 Technical Reference Manual
To write Apple IIGS programs with APW
Apple IIGS Programmer's Workshop Reference
To write Apple IIGS programs with the cross-development system
MPW IIGS Tools Reference

■ **Table P-1** 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, Volume 1	How tools work, some specifications
Apple IIGS Toolbox Reference, Volume 2	More toolbox specifications
Apple IIGS Toolbox Reference, Volume 3	More toolbox specifications, and corrections and clarifications to the first two volumes
ProDOS 8 Technical Reference Manual	Standard Apple II operating system
Apple IIGS ProDOS 16 Reference	ProDOS 16 operating system and loader
GS/OS Device Driver Reference	Device drivers and GS/OS
Human Interface Guidelines: The Apple Desktop Interface	Apple's standards for the desktop interface
Apple Numerics Manual	Standard Apple Numerics Environment
Apple IIGS Programmer's Workshop Reference	Using APW™
Apple IIGS Programmer's Workshop Assembler Reference	Using the APW Assembler
Apple IIGS Programmer's Workshop C Reference	Using the APW C Compiler
MPW IIGS Tools Reference	Using the cross-development system
MPW IIGS Assembler Reference	Using the MPW® IIGS Assembler
MPW IIGS C Reference	Using the MPW IIGS C Compiler
MPW IIGS Pascal Reference	Using the MPW IIGS Pascal Compiler
AppleShare Programmer's Guide for the Apple II	Developing network-specific applications for the Apple IIGS
Apple IIGS Debugger Reference	Debugger for all Apple IIGS programs

Introductory Apple IIGS manuals

The introductory Apple IIGS manuals are for developers, computer enthusiasts, and other Apple IIGS owners who need basic technical information. Their purpose is to help the technical reader understand the features and programming techniques that make the Apple IIGS different from other computers.

- 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.

  You should read the *Technical Introduction* no matter what kind of programming you intend to do, because it will help you understand the powers and limitations of the machine.

- When you start writing programs that use the Apple IIGS user interface (with windows, menus, and the mouse), the *Programmer's Introduction to the Apple IIGS* provides the concepts and guidelines you need. It is not a complete course in programming, only a starting point for programmers writing applications for the Apple IIGS.

  The *Programmer's Introduction* gives an overview of the routines in the Apple IIGS Toolbox and the operating environment they run under. It includes a sample event-driven program that demonstrates how a program uses the toolbox and the operating system.

Apple IIGS machine reference manuals

The machine itself has two reference manuals. They contain detailed specifications for people who want to know exactly what's inside the machine.

- The *Apple IIGS Hardware Reference* is required reading for hardware developers and anyone else who wants to know how the machine works. Information of special interest to developers includes the mechanical and electrical specifications of all connectors, both internal and external. Information of general interest includes descriptions of the internal hardware and how it affects the machine's features.

- The *Apple IIGS Firmware Reference* describes the programs and subroutines stored in the machine's read-only memory (ROM). The *Firmware Reference* includes information about interrupt routines and low-level input/output (I/O) subroutines for the serial ports, the disk port, and the Apple Desktop Bus™ interface, which controls the keyboard and the mouse. The *Firmware Reference* also describes the Monitor program, a low-level programming and debugging aid for assembly-language programs.

Apple IIGS toolbox manuals

Like the Macintosh®, the Apple IIGS has a built-in toolbox. Volume 1 of the *Apple IIGS Toolbox Reference* introduces concepts and terminology and explains how to use some of the tools. Volume 2 contains information about more tools and explains how to write and install your own tool set. Volume 3 adds more tools and includes corrections and clarifications to the other two volumes.

If you are developing an application that uses the **desktop interface,** or if you want to use the Super Hi-Res graphics display, you'll find the *Toolbox Reference* indispensable.

Apple IIGS operating-system manuals

The Apple IIGS has two operating systems: GS/OS and ProDOS 8. GS/OS uses the full power of the Apple IIGS and can access files in multiple file systems. This book describes GS/OS and includes information about the System Loader, which works closely with GS/OS to load programs into memory.

◆ *Note:* GS/OS is compatible with and replaces ProDOS 16, the first operating system developed for the Apple IIGS computer. ProDOS 16 is described in the *Apple IIGS ProDOS 16 Reference.*

ProDOS 8, previously called simply *ProDOS,* is the standard operating system for most Apple II computers with 8-bit CPUs. As a developer of Apple IIGS programs, you need to use ProDOS 8 only if you are developing programs to run on standard (8-bit) Apple II computers as well as on the Apple IIGS. ProDOS 8 is described in the *ProDOS 8 Technical Reference Manual.*

All-Apple manuals

Two manuals apply to all Apple computers: *Human Interface Guidelines: The Apple Desktop Interface* and the *Apple Numerics Manual.* If you develop programs for any Apple computer, you should know about these manuals.

The *Human Interface Guidelines* manual describes Apple's standards for the desktop interface to any program that runs on an Apple computer. If you are writing a commercial application for the Apple IIGS, you should be fully familiar with the contents of this manual.

The *Apple Numerics Manual,* second edition, is the reference for the Standard Apple Numerics Environment (SANE®), a full implementation of the IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std 754-1985). If your application requires floating-point or robust arithmetic, you'll probably want it to use the SANE routines in the Apple IIGS.

The APW manuals

Apple provides two development environments for writing Apple IIGS programs. One is the Apple IIGS Programmer's Workshop (APW). APW is a native Apple IIGS development system—it runs on the Apple IIGS and produces Apple IIGS programs. There are three principal APW manuals:

- The *Apple IIGS Programmer's Workshop Reference* describes the APW Shell, Editor, Linker, and utility programs; these are the parts of the workshop that all developers need, regardless of which programming language they use. The APW reference manual includes a sample program and describes object module format (OMF), the file format used by all APW compilers to produce files loadable by the Apple IIGS System Loader.
- The *Apple IIGS Programmer's Workshop Assembler Reference* includes the specifications of the 65816 language and of the Apple IIGS libraries, and describes how to use the assembler.
- The *Apple IIGS Programmer's Workshop C Reference* includes the specifications of the APW C implementation and of the Apple IIGS interface libraries, and describes how to use the compiler.

Other compilers can be used with the workshop, provided they follow the standards defined in the *Apple IIGS Programmer's Workshop Reference.* Several such compilers, for languages such as Pascal, are now available.

◆ *Note:* The APW manuals, along with APW itself, are available through APDA.

The MPW IIGS manuals

The Macintosh Programmer's Workshop (MPW) is the other development environment Apple provides for writing Apple IIGS programs (see Figure P-1). MPW is principally a sophisticated, powerful development environment for the Macintosh computer. It includes assemblers and compilers, linkers, and a variety of diagnostic and debugging tools. When used to write Apple IIGS programs, MPW is a cross-development system—it runs on the Macintosh, but produces executable programs for the Apple IIGS.

MPW is documented in several manuals, but the parts needed for cross-development—the editor and the build tools—are described in the *Macintosh Programmer's Workshop Reference.*

Four manuals describe the cross-development system. Each programming language has its own manual. Whichever language you program in, you also need the *MPW IIGS Tools Reference.*

- **Tools:** The *MPW IIGS Tools Reference* describes the tools needed to create Apple IIGS applications under MPW. It describes the linker, the file-conversion tool, and several other conversion and diagnostic programs.
- **Assembler:** The *MPW IIGS Assembler Reference* describes how to write Apple IIGS assembly-language programs under MPW. It also documents a utility program that converts source files written for the APW assembler to files compatible with the MPW IIGS assembler.
- **C compiler:** The *MPW IIGS C Reference* describes how to write Apple IIGS programs in C under MPW.

◆ *Note:* The MPW IIGS manuals are available through APDA.

The AppleShare programmer's manual

The *AppleShare Programmer's Guide for the Apple II* describes in detail how to develop new network-specific applications for the Apple IIGS and the Apple IIe computer. If you need more help about writing such applications, see that manual.

The debugger manual

Neither MPW IIGS nor APW includes a debugger as part of the development environment. However, the Apple IIGS Debugger, an independent product, is a machine-language debugger that runs on the Apple IIGS and can be used to debug programs produced by either MPW IIGS or APW.

The Apple IIGS Debugger is described in the *Apple IIGS Debugger Reference.*

Introduction **What Is GS/OS?**

GS/OS® is the first completely new operating system designed for the Apple IIGS® computer. It is similar in interface and call style to the ProDOS® operating systems, but it has far greater capabilities because it has many new calls, and it has much faster execution because it is written entirely in 65816 assembly language.

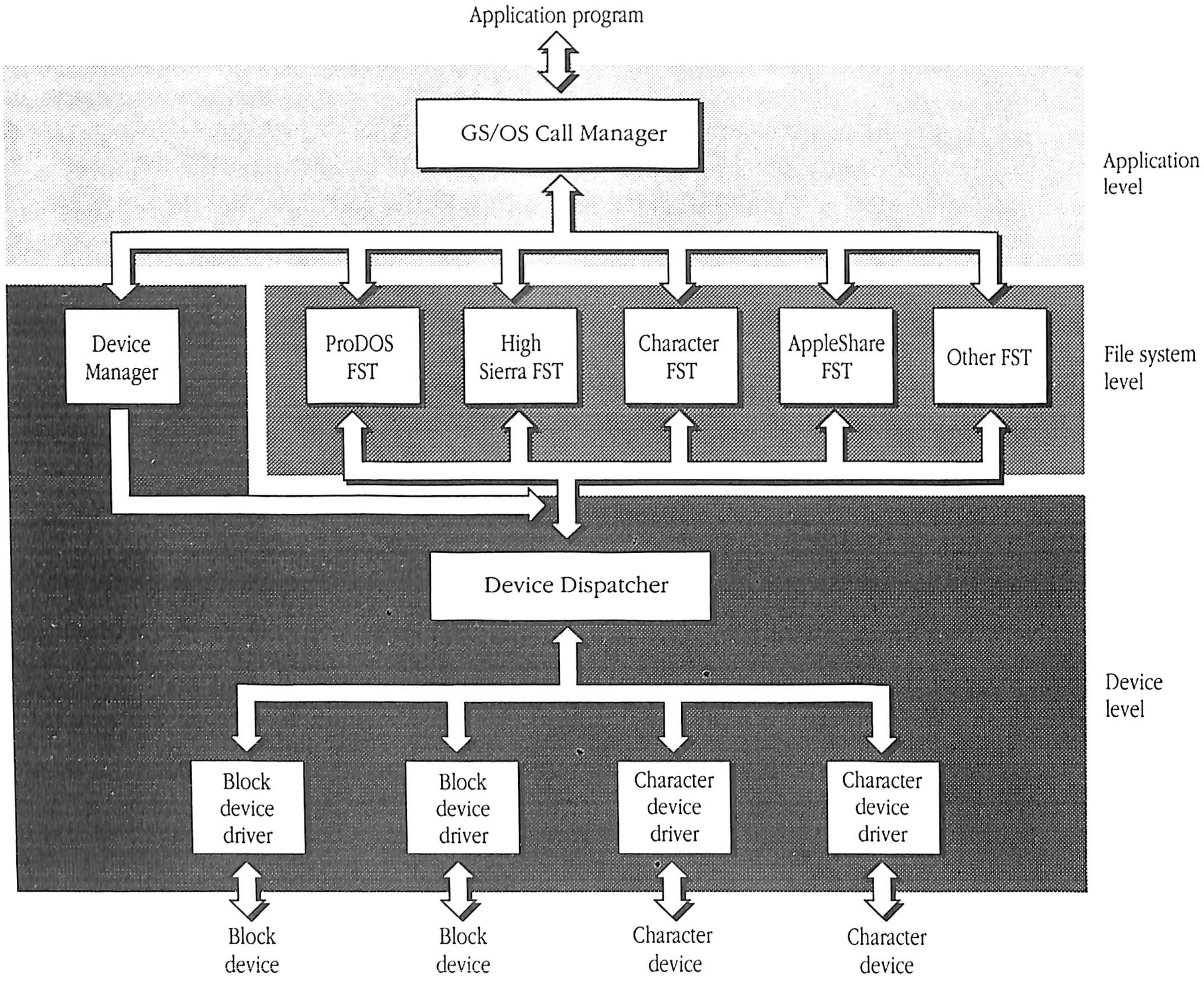
Even more important, GS/OS is file system–independent: by making GS/OS calls, your application can read and write files transparently among many different and normally incompatible file systems. GS/OS accomplishes this by defining a generic GS/OS file interface, the abstract file system. Your application makes calls to that interface, and then GS/OS uses file system translators to convert those calls and data into formats consistent with individual file systems.

This introduction gives an overview of the structure and capabilities of GS/OS, followed by a brief history of the evolution in Apple® II operating systems from DOS to GS/OS.

The components of GS/OS

GS/OS is more complex and integrated than previous Apple II operating systems. As Figure I-1 shows, you can think of it in terms of three levels of interface: the application level, the file system level, and the device level. A typical GS/OS call passes through the three levels in order, from the application at the top to the device hardware at the bottom.

■ **Figure I-1** Interface levels in GS/OS



- **Application level:** At this level, the GS/OS Call Manager processes GS/OS calls that allow an application to access files or devices, or to get or set specific system information.

  In handling a typical GS/OS call, the Call Manager mediates between an individual application and the file system level. The application-interface level is described in Part I of this book.

- **File system level:** The file system level consists of **file system translators** (FSTs), which receive application calls, convert them to a specific file system format, and send them on to device drivers. FSTs allow applications to use the same calls to read and write files for any number of file systems. FSTs also allow applications to access character devices (like display screens or printers) as if they were files.

  The file system level is completely internal to GS/OS. Although your applications don't interact with the file system level directly, you may want to know how calls are translated by different file system translators. For example, CD-ROM files are read-only, so write calls cannot be translated meaningfully by an FST that accesses files on compact discs.

  In handling a typical GS/OS call, the file system translators mediate between the application level and the device level. The file system–interface level is described in Part II of this book.

- **Device level:** The device level consists of the Device Manager, the Device Dispatcher, and all device drivers connected to the system. In handling a typical GS/OS call, the Device Manager and the Device Dispatcher mediate between the file system level and an individual device driver.

  The device level of GS/OS has two other types of communication. Your application can bypass the file system level entirely by making **device calls,** which are calls that directly access devices. Finally, device drivers communicate with the device level by accepting **driver calls,** which are mostly low-level translations of device calls.

  Devices are normally accessed through application-level file calls, described in Part I of this book. The lower-level device calls are described in the *GS/OS Device Driver Reference;* if you want to give your application direct access to devices, look there to find out how to do it. Driver calls are also described in the *GS/OS Device Driver Reference;* if you are writing a device driver, look there for details.

Another part of system software that is described in this manual is the Apple IIGS **System Loader.** The System Loader loads programs into memory and prepares them for execution. Although not strictly part of GS/OS, the System Loader occupies the same disk file as GS/OS, and works very closely with GS/OS. For most applications, however, its functioning is totally automatic; in special situations, some applications need to make loader calls.

GS/OS features

This section describes some of the principal GS/OS features of interest to application writers.

File system independence

Because it uses file system translators, GS/OS accesses non-ProDOS file systems as easily as it accesses the more familiar (to Apple II applications) ProDOS files. It is possible to gain access to any file system for which an FST has been written. Several FSTs currently exist; as Apple Computer creates new FSTs, they can be added very easily to existing systems.

The GS/OS abstract file system supports both flat and hierarchical file systems and systems with specific file types and access permissions. GS/OS recognizes standard files, directory files, and extended files (two-fork files such as those used by the Macintosh®). Certain GS/OS calls make it easy to retrieve and use directory information for any file system.

The abstract file system is described in Chapter 1 of this book. FSTs are described in Part II of this book.

Enhanced device support

All GS/OS device drivers provide a uniform interface to character and block devices. GS/OS supports both ROM-based and RAM-based device drivers, making it easier to integrate new peripheral devices into GS/OS.

GS/OS provides a uniform input/output model for both block and character devices. Devices such as printers and the console are accessed in the same way as sequential files on block devices. This can greatly simplify I/O for your application.

Unlike ProDOS 8 and ProDOS 16, GS/OS recognizes disk-switched and duplicate-volume situations, to help your application avoid writing data to the wrong disk.

Devices are normally accessed through application-level file calls, described in Part I of this book. Device drivers are described in the *GS/OS Device Driver Reference.*

Speed enhancements

GS/OS transfers data much faster than ProDOS 8 or ProDOS 16 because it uses caching, allows multiple-block reads and writes, eliminates the duplicate levels of buffering used by ProDOS 16, and is written entirely in 65816 native-mode assembly language.

Elimination of ProDOS restrictions

GS/OS allows any number of open files (rather than only 8) up to the amount of available RAM, any number of devices on line (rather than only 14), and any number of devices per slot (rather than only 2). GS/OS allows volumes and files to be as large as 4096 megabytes (MB), rather than only 16 MB for files and 32 MB for volumes.

The GS/OS file interface is described in Chapter 1 of this book.

ProDOS 16 compatibility

GS/OS includes a complete set of ProDOS 16 calls and implements them just as ProDOS 16 does. All well-designed ProDOS 16 applications can run without modification under GS/OS. Further, existing ProDOS 16 applications running under GS/OS can now automatically access files on non-ProDOS disks, and can also access character devices as files.

Where to find call descriptions

As already noted, a program can make several types of calls to GS/OS. Broadly, calls can be divided into **application-level calls** (made from application programs to GS/OS) and low-level calls (made between GS/OS and low-level software such as device drivers). Most application-level calls are described in this book; most low-level calls are described in the *GS/OS Device Driver Reference.* Within these broad divisions, there are several subcategories of calls and call-related descriptions; each subcategory is described in one of the two books.

The following call descriptions are found in this book:

- **Standard GS/OS calls:** Also called *class 1 calls* or just *GS/OS calls,* these are the primary calls an application makes to access files or system information. They are application-level calls. This category covers all operating-system calls that a typical GS/OS application makes.
- **System Loader calls:** These are calls a program makes to load other programs or program segments into memory. Although you usually don't make System Loader calls, they are described in this book in case you need them.
- **FST-specific information on GS/OS calls:** Because different file systems have different characteristics, they do not all respond identically to GS/OS calls. In addition, each FST can support the GS/OS call `FSTSpecific`, an application-level call whose function is defined individually for each FST. Therefore, this book includes descriptions of how each FST handles certain GS/OS calls, including `FSTSpecific`.
- **ProDOS 16 calls:** Also called *class 0 calls,* these are application-level calls that are identical to the calls described in the *Apple IIGS ProDOS 16 Reference.* GS/OS supports these calls so that existing ProDOS 16 applications can run without modification under GS/OS.
- **FST-specific information on ProDOS 16 calls:** Because different file systems have different characteristics, they do not all respond identically to ProDOS 16 calls. Therefore, this book includes descriptions of how each FST handles ProDOS 16 calls. There is no `FSTSpecific` ProDOS 16 call as there is for GS/OS calls.

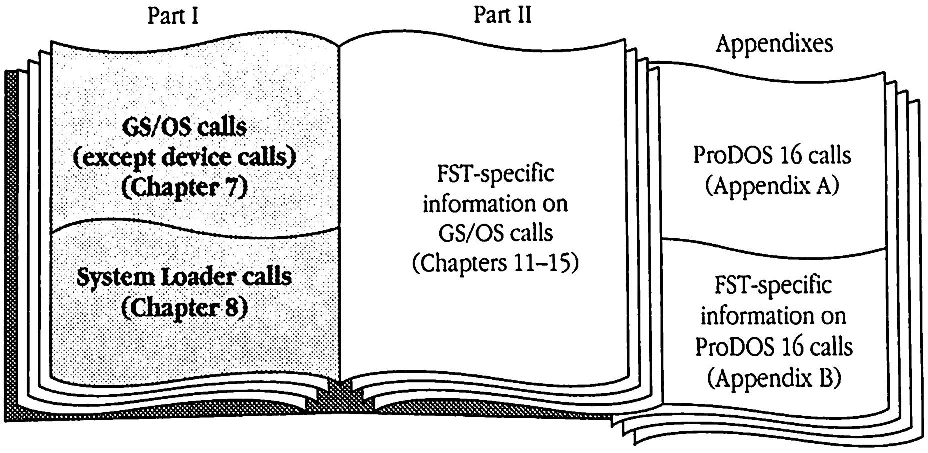
The following call descriptions are found in the *GS/OS Device Driver Reference:*

- **GS/OS device calls:** These are a subset of the standard, application-level GS/OS device calls described in the *GS/OS Reference.* The lower-level device calls are special because they bypass the file system level altogether and access devices directly.
- **Driver-specific information on GS/OS device calls:** Because different devices have different characteristics, device drivers do not all respond identically to GS/OS calls. Therefore, this book includes descriptions of how each GS/OS driver handles certain GS/OS device calls.
- **Driver calls:** These are calls that GS/OS makes to individual device drivers. They are low-level calls, of interest mainly to device-driver writers.
- **System service calls:** System service calls give low-level components of GS/OS (such as FSTs and device drivers) a uniform method for accessing system information and executing standard routines. This book describes the system service calls that GS/OS device drivers can make.

Figure I-2 shows you where to look in each book for the principal descriptions of each call category. For example, the descriptions of all standard GS/OS calls (except those that access devices) are in Chapter 7, Part I, of this book. Most applications make only the calls described in Part I (shaded area).

◆ *Note:* Figure I-2 is reproduced in each part opening in this book, highlighted to show the calls described in that part.

■ **Figure I-2** Where to find call descriptions


Part I
Part II
Appendixes
GS/OS calls (except device calls) (Chapter 7)
System Loader calls (Chapter 8)
FST-specific information on GS/OS calls (Chapters 11–15)
ProDOS 16 calls (Appendix A)
FST-specific information on ProDOS 16 calls (Appendix B)

GS/OS system requirements

GS/OS will not run on a standard Apple II computer. It requires an Apple IIGS with a ROM version of 01 or greater, at least 512 KB of RAM, and a disk drive with at least 800 KB capacity. A second 800 KB drive or a hard disk is strongly recommended.

The development of GS/OS

The material in this section is a brief discussion of how GS/OS evolved from previous Apple II operating systems.

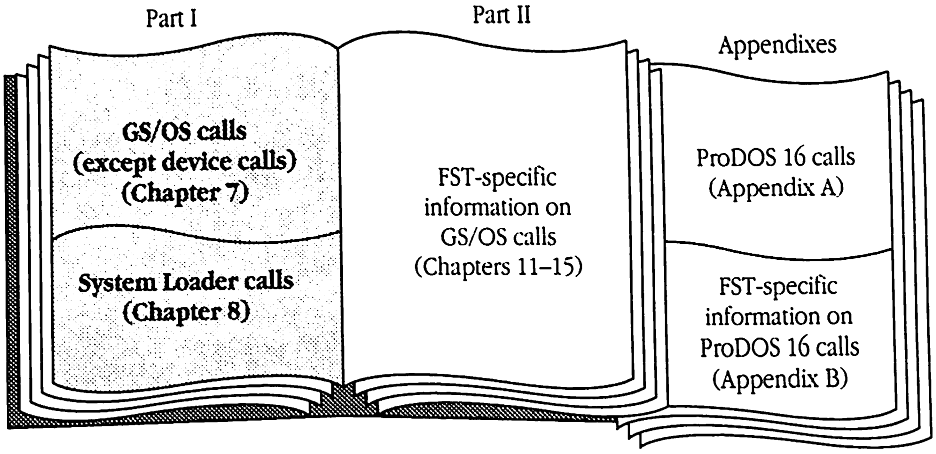
Apple Computer has created several operating systems for the Apple II family of computers. GS/OS is the latest in that line; it is related to several earlier systems, but has far greater capabilities than any of them. Here are thumbnail sketches of the other systems:

- **DOS:** DOS (for *Disk Operating System*) was Apple's first operating system. It provided the Apple II computer with its first capability to store and retrieve disk files. DOS has relatively slow data transfer rates by modern standards, supports a flat (rather than hierarchical) file system, can read 140 KB disks only, has no uniform interrupt support, includes no memory management, and is not extensible.
- **Pascal:** Apple II Pascal is Apple Computer's implementation and enhancement of the University of California, San Diego Pascal System. Its lineage is completely separate from the other Apple operating systems. Apple II Pascal supports only a flat file system, is characterized by slow, interpretive execution, provides no uniform support for interrupts, has no memory management, and is difficult to extend.
- **SOS:** SOS (for *Sophisticated Operating System*) was developed for the Apple III, but its most important feature, the file system, is the heart of the ProDOS family of operating systems (described next). SOS gives much faster data transfer than DOS, represents Apple Computer's first hierarchical file system, supports block devices up to 32 MB, provides a uniform sequential I/O model for both block devices and character devices, and includes interrupt handling, memory management, device handling, and extensibility via device drivers and interrupt handlers. The major deficiency of SOS is that it requires at least 256 KB of RAM for effective operation.
- **ProDOS 8:** ProDOS 8 (originally called ProDOS, for *Professional Disk Operating System*) brought some of the advanced features of SOS to 8-bit Apple II computers (Apple II Plus, Apple IIe, Apple IIc). It requires no more than 64 KB of RAM, and in fact can directly access only 64 KB of memory space. ProDOS supports exactly the same hierarchical file system as SOS, but it does not have the uniform I/O model for character devices and files, memory management, or uniform treatment of device drivers and interrupt handlers.
- **ProDOS 16:** ProDOS 16 (ProDOS for the 16-bit Apple IIGS) is the first step toward an operating system designed specifically for the Apple IIGS computer. It is an extension of ProDOS 8; with a few important additions, it has essentially the same features as ProDOS 8 and supports exactly the same hierarchical file system. The main advantage of ProDOS 16 is that it allows applications to interact with the operating system from anywhere in the 16 MB Apple IIGS address space.

- **GS/OS:** GS/OS fully exploits the capabilities of the Apple IIGS. It is a fast, modular, and extensible operating system that provides a file system–independent and device-independent environment for applications. While upwardly compatible from ProDOS 16, it corrects deficiencies in the I/O performance of ProDOS 16 and eliminates its restrictions on the number and size of open files, volumes, and devices. GS/OS supports character devices as files, handles devices uniformly, and supports RAM-based device drivers. GS/OS can create, read, and write files among a potentially unlimited number of different file systems (including ProDOS).

Although it is an extension of the ProDOS line, GS/OS is really a completely new operating system. As its name suggests, it is designed specifically for the Apple IIGS computer, and it is intended to be the principal Apple IIGS operating system.

Part I **The Application Level**



Chapter 1 The GS/OS Abstract File System

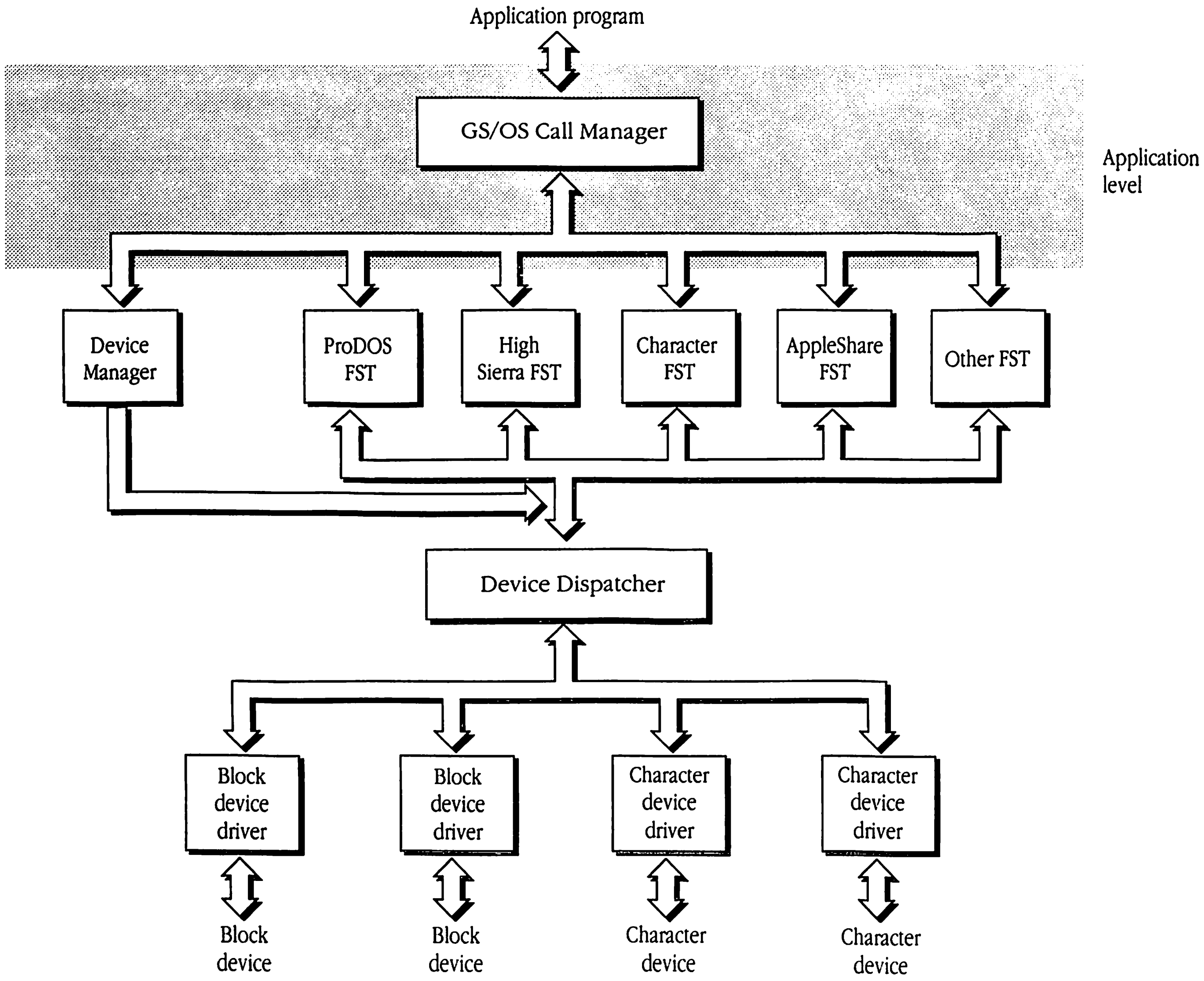
A key feature of GS/OS is its ability to insulate applications from the details of the hardware devices connected to the system and the file systems used to store applications and their data. This chapter shows how GS/OS implements this feature. It also lists, by category, the GS/OS calls that an application can make.

A high-level file system interface

GS/OS has been designed to insulate you, the application programmer, from the details of file systems and hardware devices. Normally, you simply make a GS/OS call, and GS/OS routes the call to the correct file system and device. You can think of GS/OS as looking like the illustration shown in Figure 1-1.

GS/OS can keep your application from dealing with FSTs and devices at all, and thus allow you to take a higher-level approach, by supporting files in a **hierarchical file system.** A hierarchical file system contains both normal files that contain data or applications, and

- **Figure 1-1** The application level of GS/OS



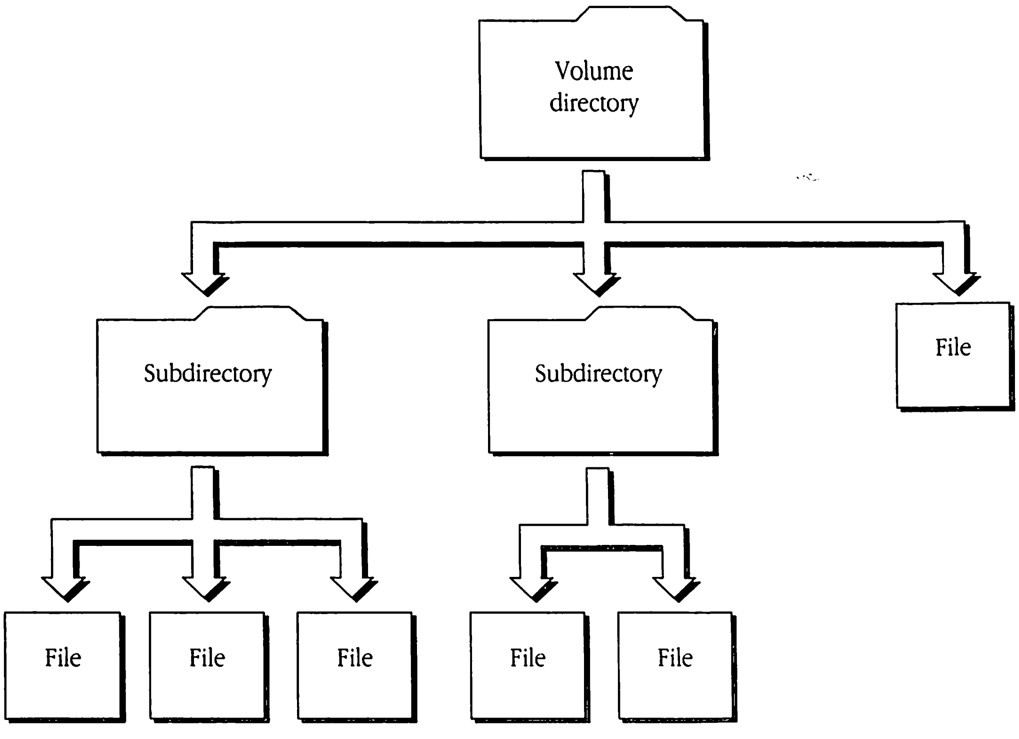
special files called **directories.** A directory file can contain the names of either normal files or other directories. Figure 1-2 shows the relationships among files in a hierarchical file system.

In GS/OS, the highest-level directory is called a **volume directory.** A volume is a logical entity that allows you to access the files contained on a physical storage medium such as a diskette, hard disk, or CD-ROM. Only block devices can be identified by **volume name,** and then only if the named volume is mounted. For example, an entire disk is identified by its volume name, which is the filename of its volume directory. GS/OS also makes certain assumptions about what each file in this hierarchical file system looks like. The assumptions are as follows:

- Each file can be classified as a directory, standard, or extended file (defined in the next section).
- Each file has a filename in a certain format.
- The logical location of each file can be uniquely identified by a pathname, which is an ordered collection of the filenames that lead to it.
- Each file has access privileges.
- Each file has a file type and an auxiliary type.
- Each file has a creation and modification date and time.

The following sections define these assumptions.

- **Figure 1-2** A hierarchical file system



Classes of GS/OS files

Every GS/OS file is a collection of bytes on a device. The classes of files are as follows:

- **Directory files** store information about other files.
- **Standard files** contain a single sequence of data.
- **Extended files** contain two sequences of data.

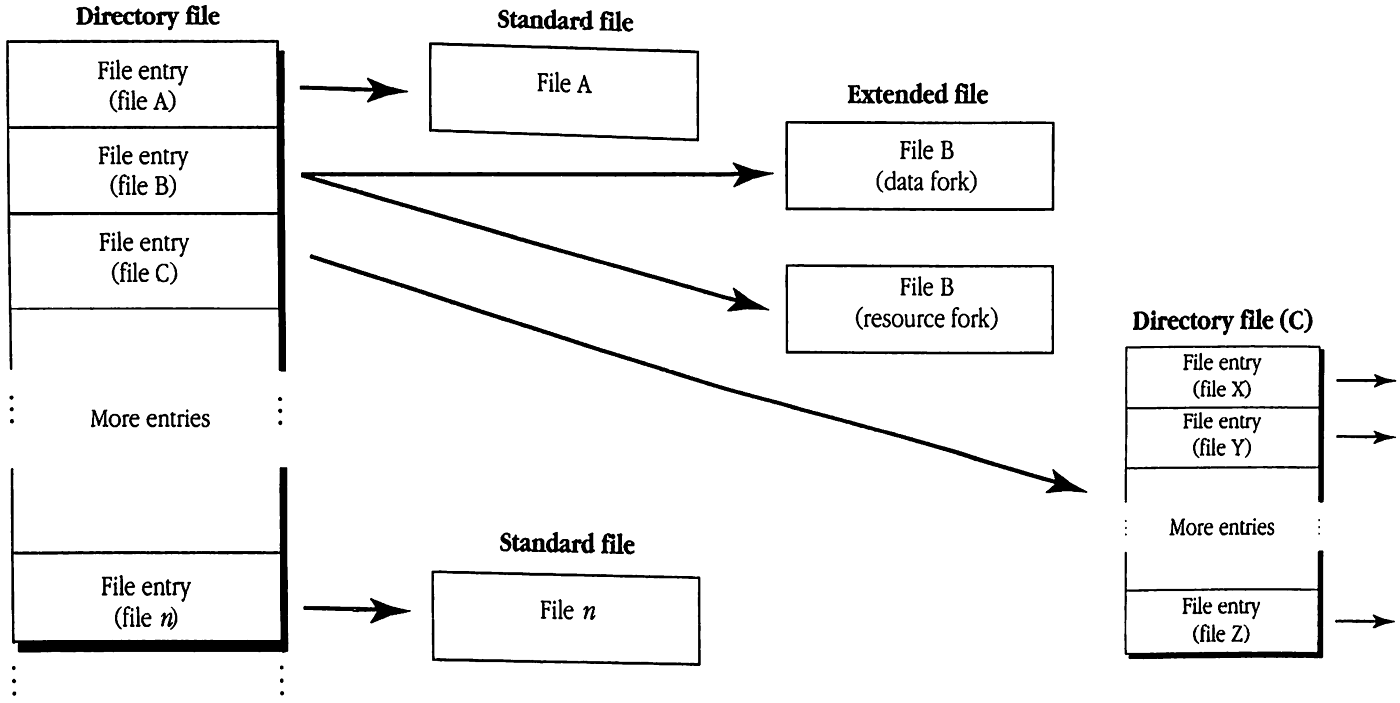
◆ *Note:* These classes of files are for block devices. GS/OS also allows you to treat character devices as if they contained files. See Chapter 14, "The Character FST."

Directory files

A directory file contains informational entries about other directories and files. Each entry in the directory file describes and points to another directory file, standard file, or extended file, as shown in Figure 1-3.

Directory files can be read from, but not written to (except by GS/OS).

- **Figure 1-3** Directory file format



A directory can, but need not, have associated file information, such as access controls, file type, creation and modification times and dates, and so on.

Usually, you need to examine directory files only when you are creating an application that catalogs files; more information about directory files is given in the section "Examining Directory Entries" in Chapter 4.

Standard files

Standard files are named collections of data consisting of a sequence of bytes and associated file information, such as access controls, file type, creation and modification times and dates, and so on. They can be read from and written to, and have no predefined internal format, because the arrangement of the data depends on the specific file type and auxiliary type.

Extended files

Extended files are named collections of data consisting of two sequences of bytes and a single set of file information, such as access controls, file type, creation and modification times and dates, and so on. The two different byte sequences of an extended file are called the **data fork** and the **resource fork.** They can be read from and written to, and have no predefined internal format; the formats depend on the specific file types.

Filenames

Every GS/OS file is identified by a filename. A GS/OS filename can be any number of characters long and can include spaces. Your application must encode filenames as sequences of 8-bit ASCII codes. All 256 extended ASCII values are legal except the colon (ASCII $3A), although most file system translators (FSTs) support much smaller legal character sets.

△ **Important** Because the colon is the pathname separator character, it must never appear in a filename. See the next section, "Pathnames," for more details about separators and pathnames. △

If an FST does not support a character that the user attempts to use in a filename, GS/OS returns error $40 (badPathSyntax). FSTs are also responsible for indicating whether filenames should be case-sensitive, and whether the high-order bit of each character is turned off. See Part II of this book for more information about FSTs.

A filename must be unique within its directory. Some examples of legal filenames are the following:

```
file-1
January Sales
long file name with spaces and special characters !@#$%
```

Pathnames

In a hierarchical file system, a file is identified by its **pathname,** a sequence of filenames starting with the name of the volume directory and ending with the name of the file. Pathnames specify the access paths to devices, volumes, directories, subdirectories, and files within flat or hierarchical file systems.

A GS/OS pathname is either a full pathname or a partial pathname, as described in the following sections. All calls that require you to name a file will accept either a full pathname or a partial pathname.

Full pathnames

A **full pathname** is one of the following names:

- a volume name followed by a series of zero or more filenames, each preceded by the same separator, and ending with the name of a directory file, standard file, or extended file
- a device name followed by a series of zero or more filenames, each preceded by the same separator, and ending with the name of a directory file, standard file, or extended file

A separator is a character that separates filenames in a pathname. Both of the following separators are valid:

- a colon : (ASCII code $3A).
- a slash character / (ASCII code $2F)

The first separator in the input string determines which separator will be used throughout. When the colon is the separator, the constituent filenames must not contain colons, but they may contain slashes. When the slash is the separator, the constituent filenames must not contain slashes or colons. Thus, colons are never allowed in filenames. These are examples of legal full pathnames:

```
:aloysius:beelzebub:cat

:a:b:c

/x

:x
```

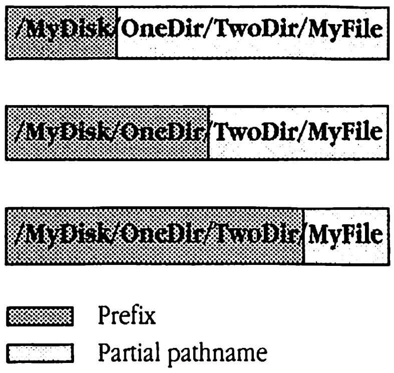
Examples of illegal full pathnames are as follows:

`/:::/::/:`	A `:` must not appear in a filename.
`:a:b:c:`	A separator must not appear after the last filename.
`a:b:c:`	A full pathname must start with a volume or device name.

Prefixes and partial pathnames

A full pathname can be broken down into a prefix and a partial pathname. The **prefix** starts at the volume or device name and can continue down the path through the last directory name, that is, down to but not including the filename. In contrast, the **partial pathname** always contains the filename and can trace back up the path to, but cannot include, the volume name or device name. Thus, when the prefix and partial pathname are combined, they yield the full pathname. Figure 1-4 illustrates the possible divisions of a single full pathname into a prefix and a partial pathname.

- **Figure 1-4** Prefixes and partial pathnames


/MyDisk/OneDir/TwoDir/MyFile
/MyDisk/OneDir/TwoDir/MyFile
/MyDisk/OneDir/TwoDir/MyFile
Prefix
Partial pathname

Prefixes are convenient when you want to access many files in the same subdirectory, because you can use a prefix designator as a substitute for the prefix, thus shortening the pathname references.

Prefix designators

A **prefix designator** takes the place of a prefix, and can be one of the following:

- A digit or sequence of digits followed by a pathname separator. The digits specify the prefix number. Thus, the prefix designators `002:` and `2/` both specify prefix number `2`.
- The asterisk character `*` followed by a pathname separator. This special prefix designator is one of the predefined prefix designators, as described later in this section.
- The *at* character `@` followed by a pathname separator. This special prefix designator is one of the predefined prefix designators, as described later in this section.

If you supply a partial pathname that doesn't contain a prefix designator to GS/OS, it takes one of the following actions:

- If prefix designator `0` is non-null, GS/OS automatically creates a full pathname by adding `0/` to the front of the partial pathname.
- If prefix designator `0` is null, GS/OS automatically creates a full pathname by adding `8/` to the front of the partial pathname.

GS/OS determines the separator for a partial pathname in the same way that it determines the separator for a full pathname, by using whichever one appeared first in the input string.

◆ *Note:* Although you may use a prefix designator as an input to the GS/OS SetPrefix call, prefixes are always stored in memory in their full pathname form (that is, they include no prefix designators themselves).

GS/OS supports two types of prefixes, as follows:

- **Short prefixes,** referred to by the prefix designators `*` and `0` through `7`, cannot be longer than 64 characters. Short prefixes are identical to the prefixes supported by ProDOS 16.
- **Long prefixes,** referred to by the prefix designators `@` and `8` through `31`, can contain up to about 8000 characters.

This means that GS/OS allows you to set 32 prefixes. You set and read prefixes using the standard GS/OS calls SetPrefix and GetPrefix. GetPrefix returns a string in which all separators are colons (ASCII $3A). Alphabetic characters are returned with the same case in which they were entered when the prefix was set.

Predefined prefix designators

For programming convenience, some prefix designators have predefined values. The asterisk (`*`) has a fixed value as the name of the volume from which GS/OS was last started up.

The *at* character `@` helps applications be AppleShare aware. Whenever an application is launched, GS/OS sets this prefix in one of two ways:

- If the application is being launched from a server, GS/OS sets the `@` prefix to the user's folder on the server.
- If the application is not being launched from a server, GS/OS sets the `@` prefix to the folder where the application resides (same as prefix `9`).

Other prefix designators have default values assigned by GS/OS at application launch (see Tables 2-5 through 2-7 in Chapter 2), but your application can change those values when it is running.

Table 1-1 shows some examples of prefix use in which prefix `0:` is set to `:VOLUME1:` and prefix `5:` is set to `:VOLUME1:TEXT.FILES:`. The pathname provided by the application is compared with the full pathname constructed by GS/OS.

- **Table 1-1** Prefixes used with full and partial pathnames

Full pathname
  as supplied: `:VOLUME1:TEXT.FILES:CHAP.3`
  as expanded by GS/OS: `:VOLUME1:TEXT.FILES:CHAP.3`

Partial pathname—implicit use of prefix `:0`
  as supplied: `GS.OS`
  as expanded by GS/OS: `:VOLUME1:GS.OS`

Partial pathname—explicit use of prefix `:0`
  as supplied: `0:SYSTEM:FINDER`
  as expanded by GS/OS: `:VOLUME1:SYSTEM:FINDER`

Partial pathname—explicit use of prefix `5:`
  as supplied: `5:CHAP.12`
  as expanded by GS/OS: `:VOLUME1:TEXT.FILES:CHAP.12`

File information

GS/OS files have several characteristics, including the following:

- Access privileges
- A file type and an auxiliary type
- File size and the current reading-writing position
- Creation and modification date and time

Your application can access and modify this information. These characteristics are introduced in the following sections and described more completely in Chapter 4, "Accessing GS/OS Files."

File access

The characteristic of file access determines what operations can be performed on the file. Several GS/OS calls read or set the access attribute for the file, which can determine the following capabilities:

- whether the file can be destroyed
- whether the file can be renamed
- whether the file is invisible, that is, whether its name is displayed by file-cataloging applications
- whether the file needs to be backed up
- whether an application can write to the file
- whether an application can read from the file

File types and auxiliary types

The file type and auxiliary type of a file do not affect the contents of a file in any way; they indicate to applications the type of information stored in the file. Apple Computer reserves the right to assign file type and auxiliary type combinations, except for the user-defined file types $F1 through $F8. The current list of file types is available on AppleLink® or from Apple Developer Technical Support.

△ **Important** If you need a new file type or auxiliary type assignment, please contact Apple Developer Technical Support. △

EOF and mark

To make reading from and writing to files easier, each open standard file and each fork of an open extended file have a byte count indicating the size of the file in bytes (the end-of-file, or **EOF**), and another defining the current position in the file (the **mark**). GS/OS moves the position of both the EOF and the mark automatically when data is added to the end of the file, but an application program must move them whenever data is deleted or added somewhere else in the file.

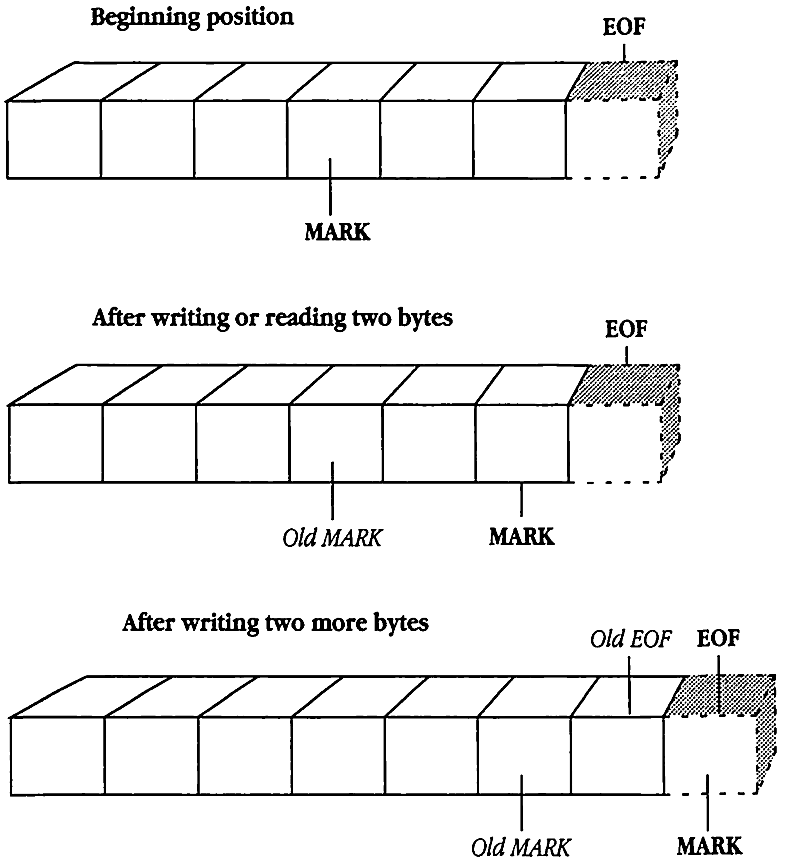
The EOF represents the number of readable bytes in the file. Since the first byte in a file has number 0, the value of the EOF indicates one position past the last character in the file.

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 new data can be written. The value of the mark cannot exceed the value of the EOF.

If the mark meets the EOF during a write operation, both the mark and 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 1-5, on the next page, illustrates the relationship between the mark and EOF.

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 SetEOF and SetMark calls. The current values of the EOF and the mark can be determined using the GetEOF and GetMark calls.

■ **Figure 1-5** How GS/OS moves the EOF and the mark


Beginning position
EOF
MARK
After writing or reading two bytes
EOF
Old MARK
MARK
After writing two more bytes
Old EOF
EOF
Old MARK
MARK

Creation and modification dates and times

All GS/OS files are marked with the date and time of their creation. When a file is first created, GS/OS stamps the file's directory entry with the current date and time from the system clock. If the file is later modified, GS/OS 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 directories—including the volume directory—must be updated.

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

Character devices as files

As part of its uniform interface, GS/OS permits applications to access character devices, like block devices, through file calls. An extension to the GS/OS abstract file system lets you make standard GS/OS calls to read to and write from character devices. This facility can be a convenience for I/O redirection.

When character devices are treated as files, only certain features are available. You can read from a character device but you cannot, for example, format it. Only the following GS/OS calls have meaning whan applied to character devices: Open, Newline, Read, Write, Close, and Flush (see brief descriptions of these calls later in this chapter).

In general, character "files" under GS/OS are much more restricted in scope than block files:

- There are no extended or directory files. Character devices are accessed as if they were standard files—single sequences of bytes. Further, it is not possible to obtain or change the current position (mark) in the sequence.
- Character devices are not hierarchical. The only legal pathname for a character "file" is a device name.
- Character devices may recognize some file-access attributes (read-enable, write-enable), but not others (rename-enable, invisibility, destroy-enable, backup-needed).
- Character "files" have no file type, auxiliary type, EOF, creation time, or other information associated with block-file directory entries.

In spite of these restrictions, it is generally quite simple and straightforward to treat character devices as files. For more information on file access to character devices, see Chapter 14, "The Character FST."

Groups of GS/OS calls

Chapters 4 through 6 list and describe the GS/OS operating system routines that are normally called by an application. They are divided into the following categories:

- File access calls (described in Chapter 4)
- Volume name and pathname calls (described in Chapter 5)
- System information and control calls (described in Chapter 6; the Quit call is described in Chapter 2)
- Interrupt calls (described in Chapter 10) and calls that directly access devices (see the *GS/OS Device Driver Reference*).

Tables 1-2 and 1-3 list the groups of GS/OS calls.

The following sections give you an overview of the capabilities of the calls in these groups. Each call is discussed in much greater detail in Chapter 7, "GS/OS Call Reference."

File access calls

The most common use of GS/OS is to make calls that access files. Your application places a file on disk by issuing a GS/OS Create call. This call specifies the file's pathname and storage type (standard file, extended file, or directory) and possibly other information about the state of the file, such as access attributes, file type, creation and modification dates and times, and so on.

Your program must make the GS/OS Open call in order to access a file's contents. For an extended file, individual Open calls are required for the data fork and resource fork, which are then read and written independently. When your application opens a file, the application must establish the access privileges.

A file can be simultaneously opened any number of times with read access. However, a single Open call with write access precludes any other Open calls on the given file.

While a file is open, your application can perform any of the following tasks:

- Read data from the file by using the Read call, or write data to the file by using the Write call
- Set or get the mark by using the SetMark or GetMark call, and set or get the end of the file by using the SetEOF or GetEOF call
- Enable or disable newline mode by using the Newline call

- **Table 1-2** GS/OS file access calls

Call	Call	Call	Call
Create ($2001)	Read ($2012)	SetEOF ($2018)	BeginSession ($201D)
Destroy ($2002)	Write ($2013)	GetEOF ($2019)	EndSession ($201E)
SetFileInfo ($2005)	Close ($2014)	SetLevel ($201A)	SessionStatus ($201F)
GetFileInfo ($2006)	Flush ($2015)	GetLevel ($201B)	ResetCache ($2026)
ClearBackupBit ($200B)	SetMark ($2016)	GetDirEntry ($201C)	
Open ($2010)	GetMark ($2017)		
Newline ($2011)			

- **Table 1-3** Other GS/OS call groups

Volume name–pathname calls	System information calls	System control calls	Interrupt and Device calls
ChangePath ($2004)	SetSysPrefs ($200C)	Quit ($2029)	BindInt ($2031)
Volume ($2008)	GetSysPrefs ($200F)	AddNotifyProc ($2034)	UnbindInt ($2032)
SetPrefix ($2009)	GetName ($2027)	DelNotifyProc ($2035)	DControl ($202E)
GetPrefix ($200A)	GetVersion ($202A)	Null ($200D)	DInfo ($202C)
ExpandPath ($200E)	GetFSTInfo ($202B)	OSShutdown ($2003)	DRead ($202F)
Format ($2024)	FSTSpecific ($2033)		DStatus ($202D)
EraseDisk ($2025)	GetStdRefNum ($2037)		DWrite ($2030)
GetBootVol ($2028)	GetRefNum ($2038)		DRename ($2036)
	GetRefInfo ($2039)		GetDevNumber ($2020)

- If the open file is a directory file, get the entries held in the file by using the GetDirEntry call
- Write changes to the disk for one or more open files by using the Flush, GetLevel, and SetLevel calls

When you are through working with an open file, you issue a GS/OS Close call to close the file and release any memory that it was using back to the Memory Manager.

After the file has been closed, you can use other GS/OS calls to work with it. One of these calls, ClearBackupBit, clears a bit so that the file appears to GS/OS as if it does not need backing up; another GS/OS call, Destroy, can be used to delete a file. Other GS/OS calls that work on closed files are described in Chapter 4.

The GS/OS calls SetFileInfo and GetFileInfo allow you to access the information in the file's directory entry. These calls are particularly useful when you are copying files because they allow you to change the creation and modification dates for a file.

The GS/OS call ResetCache allows you to resize the GS/OS cache and be able to use that resized cache immediately.

A final group of GS/OS calls—BeginSession, EndSession, and SessionStatus—are useful when you want your application to defer writing files to disk.

The background information on the file access calls is described in Chapters 1 and 4, and each individual call is listed alphabetically by name and described in detail in Chapter 7.

Volume and pathname calls

GS/OS provides a whole set of calls to deal with those situations where you want to work directly with volumes and pathnames. These calls allow you to do the following tasks:

- get information about a currently mounted volume by using the Volume call
- build a list of all mounted volumes by using the DInfo, Volume, Open, and GetDirEntry calls
- get the name of the current boot volume by using the GetBootVol call
- format a volume by using the Format call
- quickly empty a volume by using the EraseDisk call
- set or get pathname prefixes by using the SetPrefix or GetPrefix call
- change the pathname of a file by using the ChangePath call
- expand a partial pathname of a file to its full pathname by using the ExpandPath call

The background information on the volume and pathname calls is described in Chapter 5, and each individual call is listed alphabetically by name and described in detail in Chapter 7.

System information calls

The system information calls allow you to do the following tasks:

- set or get system preferences by using the SetSysPrefs and GetSysPrefs calls, which allow you to customize some GS/OS features
- get information about a specified FST by using the GetFSTInfo call
- use any special capabilities of an FST by using the FSTSpecific call
- find out the version of the operating system by using the GetVersion call
- get the filename of the currently executing application by using the GetName call

- get the reference number of the last Open call to any of the three standard prefixes by using the GetStdRefNum call
- get the reference number and access attributes for an open file by using the GetRefNum call
- get the access attributes and full pathname for an open file by using the GetRefInfo call

The background information on the system information calls is described in Chapter 6, and each individual call is listed alphabetically by name and described in detail in Chapter 7.

System control calls

The system control calls allow you to do the following tasks:

- terminate your application by using the Quit call
- add a procedure to the notification queue by using the AddNotifyProc call, or delete a procedure from the notification queue by using the DelNotifyProc call
- execute any pending events without doing anything else by using the Null call
- shut down GS/OS by using the OSShutdown call

The background information on the system control calls is described in Chapter 6, and each individual call is listed alphabetically by name and described in detail in Chapter 7.

Interrupt and device calls

GS/OS has two calls that allow you to work with interrupts. You can add an interrupt handler by using the BindInt call, or delete the interrupt handler by using the UnbindInt call. The calls are briefly summarized in Chapter 7, "GS/OS Call Reference." The mechanism for handling interrupts and signals is described in Chapter 10, "Handling Interrupts and Signals."

GS/OS offers a set of calls that allow you to access devices directly, rather than going through any file system. Most applications will not need to use any of these calls, except perhaps DInfo and GetDevNumber (their use is described in Chapter 5). The GS/OS device calls allow you to perform the following tasks:

- get general information about a device by using the DInfo call
- read information directly from a device by using the DRead call
- write information directly to a device by using the DWrite call

- get status information about a device by using the DStatus call
- send commands to a device by using the DControl call
- rename a device by using the DRename call
- get the device number of a device by using the GetDevNumber call

The individual device calls are listed alphabetically by name and briefly summarized in Chapter 7, "GS/OS Call Reference." The device calls are completely described in the *GS/OS Device Driver Reference.*

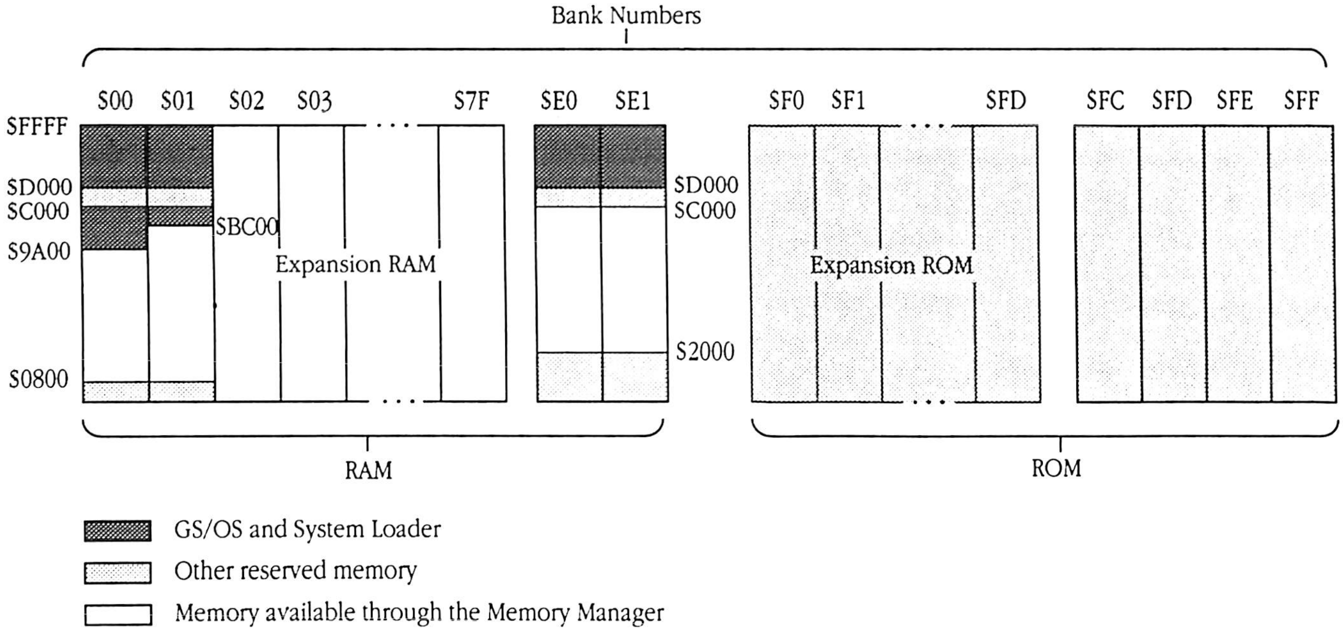
Chapter 2 GS/OS and Its Environment

GS/OS 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 GS/OS functions in that environment and how it relates to the other components.

Apple IIGS memory

The Apple IIGS microprocessor can directly address 16 megabytes (16 MB) of memory. The minimum memory configuration for GS/OS on the Apple IIGS is 512 kilobytes (512 KB) of RAM and 128 KB of ROM. As shown in Figure 2-1, the total memory space is divided into 256 banks of 64 KB each.

■ **Figure 2-1** Apple IIGS memory map



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.

Other reserved memory includes the ROM space in banks $FC–FF; they contain firmware and ROM-based tools. In addition, banks $F0–FB are reserved for future ROM expansion.

Memory allocatable through the Memory Manager is in bank $00 at locations $0800–$9A00, bank $01 at $0800–$BC000, banks $E0–$E1 at $2000–$C000, and banks $02–$7F at locations $0000–$FFFF (all 64 KB) in each bank. For example, a 1 MB Apple IIGS Memory Expansion Card makes available 16 additional banks of memory.

Under most circumstances, you should simply request memory from the Memory Manager, rather than using fixed locations. The Memory Manager is described in the *Apple IIGS Toolbox Reference*. The only fixed locations you need to use are listed in the next section.

△ **Important** Don't use all of the available memory. To process pathnames and such, GS/OS allocates memory through the Memory Manager. If you've allocated all of the available memory, GS/OS returns error $54 (`outOfMem`). If the condition is so severe that GS/OS can no longer function, it returns a fatal GS/OS error with an ID = 2, and the user will be asked to restart the system. △

For more detailed pictures of Apple IIGS memory, see the *Technical Introduction to the Apple IIGS,* the *Apple IIGS Hardware Reference,* and the *Apple IIGS Firmware Reference.*

Entry points and fixed locations

Because most Apple IIGS memory blocks are movable and under the control of the Memory Manager (see the next section, "Managing Application Memory"), 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 "Accessing Data in a Movable Memory Block" later in this chapter) should take the place of access to fixed entry points.

The supported GS/OS entry points are $E100A8 and $E100B0. These locations are the entry points for all GS/OS calls. The Tool Dispatcher entry point is $E10000, which is the entry point for all Apple IIGS tool calls, including the System Loader (described in Chapter 8).

◆ *Note:* How to use the entry points to make GS/OS calls is described in Chapter 3, "Making GS/OS Calls."

The GS/OS entry points, and the other fixed locations in bank $E1 that GS/OS supports, are shown in Table 2-1.

■ **Table 2-1** GS/OS vector space

Address	Description
$E10000	Entry vector for all Apple IIGS tool calls.
$E100A8–$E100AB	Entry vector for in-line GS/OS system calls
$E100AC–$E100AF	Reserved for internal use
$E100B0–$E100B3	Entry vector for stack-based GS/OS system calls
$E100B4–$E100B9	Reserved for internal use
$E100BA–$E100BB	Two NULL bytes (guaranteed to be zeros)
$E100BC	OS_KIND byte—indicates currently running operating system, as follows: $00—ProDOS 8 $01—GS/OS $FF—none; operating system is being loaded or switched
$E100BD	OS_BOOT byte—indicates the operating system that was initially booted, as follows: $00—ProDOS 8 $01—GS/OS
$E100BE–$E100BF	Bit 15 = 0—GS/OS is not busy Bit 15 = 1—GS/OS is busy processing a system call

Managing application memory

The Memory Manager, a ROM-resident Apple IIGS tool set, controls the allocation, deallocation, and repositioning of memory blocks in the Apple IIGS. It works closely with GS/OS and the System Loader to provide the needed memory spaces for loading programs and data and for providing buffers for input and output. All Apple IIGS software, including the System Loader and GS/OS, 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 can modify it (such as by moving it or deleting it) and how it must be placed in memory (for example, at a fixed address). See the chapter on the Memory Manager in the *Apple IIGS Toolbox Reference* for more information.

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.

Obtaining application memory

Any memory that an application needs for its own purposes must be requested directly from the Memory Manager. Figure 2-1 at the beginning of this chapter shows which parts of the Apple IIGS memory applications can allocate through requests to the Memory Manager. Applications for the Apple IIGS should avoid requesting absolute (fixed-address) blocks. See also the *Programmer's Introduction to the Apple IIGS* and the *Apple IIGS Toolbox Reference.*

Accessing data in a movable memory block

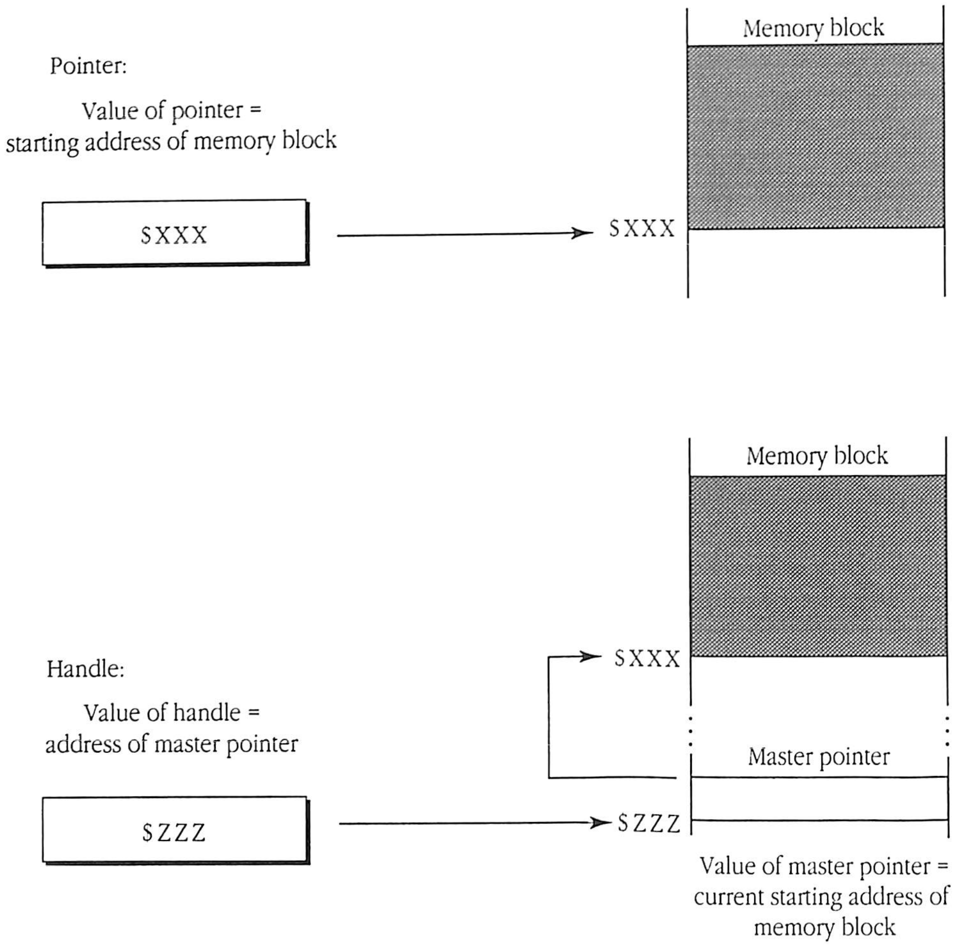
To access data in a movable block, an application cannot use a simple pointer, because the Memory Manager may move the block and change the data's address. Instead, each time the Memory Manager allocates a memory block, it returns to the requesting application a handle referencing that block.

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 2-2 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.

GS/OS 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 GS/OS are four bytes long, for ease of manipulation in the 16-bit registers of the 65C816 microprocessor.

■ **Figure 2-2** Pointers and handles



Allocating stack and direct-page space

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

The **direct page** is the Apple IIGS equivalent to the standard Apple II zero page. The difference is that it need not be absolute page zero in memory. Like the stack, the direct page can 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 addresses between $800 and $9A00 in 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 4 KB in most cases.

Automatic allocation of stack and direct-page space

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 files. If you specify at that time the total amount of space needed, GS/OS 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) when you assemble or compile your program. 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 GS/OS default (see the section "GS/OS Default Stack and Direct-Page Space" later in this chapter) is sufficient, 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.

Allocation at load 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 GS/OS. GS/OS sets the accumulator (A), direct (D), and stack pointer (S) 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** GS/OS 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.

◆ *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.

GS/OS default stack and direct-page space

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 GS/OS. However, the loader does not call the Memory Manager to allocate a direct-page/stack space, and it returns 0's as the base address and size of the space. GS/OS then calls the Memory Manager itself, and allocates a 4 KB direct-page/stack segment.

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

GS/OS sets the A, D, and S registers before handing control to the program, as follows:

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

When your application terminates with a Quit call, GS/OS disposes of the direct-page/stack segment.

GS/OS and interrupts

Do not leave interrupts disabled any longer than absolutely necessary. There are some times when interrupts must be disabled, such as in a critical timing loop for a disk driver. In particular, be aware of the following conditions:

- Do not make operating system calls with interrupts disabled. These calls could potentially take long periods of time to complete (for example, a large file read).
- If interrupts are disabled inside a loop, the effect is multiplied by the number of iterations.
- Interrupt handlers (like heartbeat tasks) execute with interrupts off; therefore, keep their run time as short as possible (such as setting a flag for a foreground task to check).

AppleShare needs to have interrupts enabled to function correctly. When interrupts are off, packets cannot be received from or sent to other computers, thus causing network services to stop functioning.

Interrupts must be on for an incoming packet to be received. Therefore, repeatedly turning interrupts on and off can be just as bad as leaving them off the entire time. For example, if a section of code has interrupts disabled 80 percent of the time and enabled 20 percent of the time, you will miss approximately 80 percent of all incoming packets.

A new state of awareness

ProDOS has traditionally been a single-user, single-computer operating system and file system. With the addition of AppleShare support to GS/OS, many computers (and many types of computers) can share the same files (on the file server) at the same time.

An **AppleShare-aware program** is a program that can be successfully run from an AppleShare file server. Such a program should be able to do the following tasks:

- load and save files on a file server
- handle error conditions in a reasonable manner (such as putting up a dialog box instead of crashing the machine)
- allow the user to quit from the program and return to a calling program (instead of having to reboot or switch off the machine)

More information on AppleShare is available in Chapter 4 and Chapter 15.

System startup considerations

The startup sequence for the Apple IIGS is invisible to applications and relatively complex, so this section describes only the general requirements for a system disk. For more detailed information, see GS/OS Technical Note #1.

Table 2-2 shows the files that must be in place for a disk to be a startup disk.

At startup, GS/OS sets prefix 0 to the boot volume name and prefix 2 to `*:SYSTEM:LIBS`.

GS/OS selects the application to run at startup by taking the following steps:

1. It first looks for a type $B3 file named `*:SYSTEM:START`. Typically, that file is the Finder, but it can be any Apple IIGS application. If `START` is found, it is selected.
2. If there is no `START` file, GS/OS searches the boot volume directory for a file that is either one of the following types:
   - □ a ProDOS 8 system program (type $FF) with the filename extension `.SYSTEM`
   - □ a GS/OS application (type $B3) with the filename extension `.SYS16`

Whichever is found first is selected.

■ **Table 2-2** General requirements for a system disk

File	Description
`*:SYSTEM:START.GS.OS.`	This file is divided into GLoader and GQuit. GLoader is the operating system loader. It's temporary and is used only during system startup. GQuit is the program dispatcher. It contains the code used for starting and quitting ProDOS 8 and GS/OS applications.
`*:SYSTEM:ERROR.MSG`	This file contains the system error messages.
`*:SYSTEM:FSTS:`*startFST*	The start FST must reside in the subdirectory, must have a file type of $BD, and must have the high bit of its auxiliary type set to 0. Any other FSTs to be loaded at startup must reside in the `*:SYSTEM:FSTS` subdirectory. The files must be Apple IIGS load files of type $BD. If bit 15 of a file's auxiliary type is 1, the FST is not loaded.
`*:SYSTEM:SYSTEM.SETUP:TOOL.SETUP.`	The `TOOL.SETUP` file must have file type $B6; it executes, in turn, every file (other than `TOOL.SETUP`) that it finds in the `*:SYSTEM:SYSTEM.SETUP` subdirectory. The files must be Apple IIGS load files of type $B6 or $B7. If bit 15 of a file's auxiliary type is 1, the setup file is not executed.
`*:SYSTEM:DESK.ACCS`	GS/OS installs all desk accessories it finds in this subdirectory. The files must be Apple IIGS load files of type $B8 or B9. If bit 15 of a file's auxiliary type is 1, the desk accessory is not loaded.

◆ *Note:* If a ProDOS 8 system program is found first, but the ProDOS 8 operating system (file `*:SYSTEM:P8`) is not on the boot volume, GS/OS then searches for and selects the first ProDOS 16 application.

The Apple IIGS startup sequence ends when control is passed to the GS/OS program dispatcher. This routine is entered both at boot time and whenever an application terminates with a GS/OS, ProDOS 16, or ProDOS 8 Quit call. The GS/OS program dispatcher determines which program is to be run next, and runs it. After startup, the program dispatcher is permanently resident in memory.

Quitting and launching applications

When you want your application to quit, you issue a GS/OS Quit call. GS/OS performs all necessary functions to shut down the current application, determines which application should be executed next, and then launches that application. When you issue the Quit call, you can take the following actions:

- indicate to GS/OS whether your application can be restarted from memory
- specify the next application to be launched
- specify whether your application should be placed on the quit return stack so that it will be restarted when the other program quits
- specify whether prefixes `10` through `12` should be set to .CONSOLE or left as is

The following sections further explain your options when quitting.

Specifying whether an application can be restarted from memory

When your application sets the `restart-from-memory` flag in the Quit call to TRUE (bit 14 of the flags word = 1), the application can be restarted from a dormant state in the computer's memory. If your application sets the `restart-from-memory` flag to FALSE (bit 14 = 0), the program must be reloaded from disk the next time it is run.

If you set the `restart-from-memory` flag to TRUE, remember that the next time the application is run, its code and data will be exactly as they were when the application quit. Thus, you may need to reinitialize certain data locations.

Specifying standard prefixes

To support I/O redirection, prefixes `10`, `11`, and `12` are defined to be the Standard I/O prefixes, as follows:

prefix `10` = `StdIn`

prefix `11`= `StdOut`

prefix `12` = `StdError`

If your application sets the `skip-std-prefixes` flag to FALSE (bit 13 = 0) in its Quit call, GS/OS sets the standard I/O prefixes to `.CONSOLE` before launching the next application. When your application sets the `skip-std-prefixes` flag in the Quit call to TRUE (bit 13 of the flags word = 1), the standard I/O prefixes remain unchanged.

When a GS/OS application is launched at startup or after a ProDOS 8 application has quit, the standard I/O prefixes are always set to `.CONSOLE`.

Specifying the next application to launch

When you are quitting your application, and want to pass control to another application, you supply the pathname of that application in the Quit call.

◆ *Note:* GS/OS loads only programs that have a file type $B3, $B5, or $FF.

Specifying a GS/OS application to launch

You should not specify a device name if you are specifying the pathname of a GS/OS application; GS/OS returns a fatal error if the device does not contain a disk. GS/OS does not handle volume names or filenames longer than 32 characters.

Specifying a ProDOS 8 application to launch

If you are quitting to a ProDOS 8 application, the pathname specified in the Quit call must be a legal ProDOS 8 pathname. In particular, device names must not be used when specifying the pathname of a ProDOS 8 application; ProDOS 8 will return a fatal error.

The GS/OS Program Dispatcher then takes the following steps:

1. Shuts down GS/OS and the System Loader.
2. Allocates segments in nonspecial memory and copies parts of GS/OS into them.
3. Allocates all special memory for the application.
4. Loads and starts up ProDOS 8.

When the ProDOS 8 application quits, the next action depends on whether the ProDOS 8 application uses a standard ProDOS 8 `QUIT` call, or an enhanced ProDOS 8 `QUIT` call.

- If the ProDOS 8 application executes a standard ProDOS 8 `QUIT` call, the GS/OS Program Dispatcher restarts GS/OS and the System Loader and launches the next application on the quit return stack.
- If the ProDOS 8 application executes an enhanced ProDOS 8 `QUIT` call, which contains a pathname to an application to be launched, control is passed to the specified application. The specified application can be a ProDOS 8 application or a GS/OS application. If it is a GS/OS application, the Program Dispatcher will restart GS/OS and the System Loader and then launch the application.

Specifying whether control should return to your application

The **quit return stack** is a stack of user IDs used to restart applications that have previously quit. If an application specifies a TRUE `quit return` flag (bit 15 of the flags word = 1) in its Quit call, GS/OS pushes the user ID of the quitting program onto the quit return stack and saves information needed to restart the program. As subsequent programs run and quit, several user IDs may be pushed onto the stack. With this mechanism, multiple levels of shells can execute subprograms and subshells, while ensuring that they eventually regain control when their subprograms quit.

For example, the `START` file might pass control to a software development system shell, using the Quit call to specify the pathname of the shell and placing its own ID on the stack. The shell in turn could hand control to a debugger, likewise placing its own ID on the stack. If the debugger quits without specifying a pathname, control passes automatically back to the shell; if the shell then quits without specifying a pathname, control passes automatically back to the `START` file.

This automatic return mechanism is specific to the GS/OS 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.

Quitting without specifying the next application to launch

If you want to quit your application and do not want to specify the next application to be launched, supply the following parameters in the Quit call:

- no pathname
- a FALSE `quit return` flag

GS/OS then attempts to pull a user ID off the quit return stack and relaunch that application. If the quit return stack is empty, GS/OS will attempt to relaunch the `START` program.

Launching another application without returning

When you are quitting your application and want to pass control to another application, but do not want control to eventually return to your application, supply the following parameters in the Quit call:

- the pathname of the application to be launched
- a FALSE `quit return` flag

GS/OS will attempt to launch the specified application.

Launching another application and returning

If you want to pass control to another application, and want control to return to your application when the next application is finished, set the `quit return` flag to TRUE in the Quit call. That way your program can function as a shell—whenever it quits to another specified program, it knows that it will eventually be reexecuted. Supply the following parameters in the Quit call:

- the pathname of the application to be launched
- a TRUE `quit return` flag

GS/OS pushes the user ID of your quitting application onto the quit return stack, and then attempts to launch the specified application.

Machine state at application launch

The GS/OS Program Dispatcher initializes certain components of the Apple IIGS and GS/OS before it passes control to an application. The initial state of those components is described in the following sections.

Machine state at GS/OS application launch

When a GS/OS program is launched, the machine state is as shown in Table 2-3.

- **Table 2-3** Machine state at GS/OS application launch

Item	State
Reserved memory	Addresses above $9A00 in bank $00 and above $BC00 in bank $01 are reserved for GS/OS, and are therefore unavailable to the application. A direct-page/stack space, of a size determined either by GS/OS or by the application itself, is reserved for the application; it is located in bank $00 at an address determined by the Memory Manager.
Hardware registers	
A register	User ID assigned to the application
X and Y registers	Zero ($0000)
e, m, and x flags in the processor status register	Set to zero; processor in full native mode
stack register	Address of the top of the direct-page/stack space
direct register	Address of the bottom of the direct-page/stack space
Standard input/output	For both $B3 and $B5 files, standard input, output, and error locations are set to Pascal 80-column character device vectors
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 GS/OS vector space. The specific values an application finds in the vector space are shown in Table 2-1 earlier in this chapter.
Pathname prefix values	Set as described in the section "Pathname Prefixes at GS/OS Application Launch" later in this chapter.

Machine state at ProDOS 8 application launch

When a ProDOS 8 program is launched, the machine state is as shown in Table 2-4.

- **Table 2-4** Machine state at ProDOS 8 application launch

Item	State	
Reserved space	All special memory is reserved for use by the program.	
Hardware registers		
A, X, and Y registers	Undefined	
e flag in processor status register	Set to 1; processor is in emulation mode	
stack register	Set to $01FB	
direct register	Set to $0000	
Shadowing	Shadow register is $08, which means	
	language card and I/O spaces:	shadowing ON
	text pages:	shadowing ON
	graphics pages:	shadowing ON
Pathname prefix values	Set as described in the section "Pathname Prefixes at ProDOS 8 Application.Launch" later in this chapter.	

Pathname prefixes at GS/OS application launch

When a GS/OS application is launched, all 32 GS/OS prefix numbers are assigned to specific pathnames (some are meaningful pathnames, whereas others are NULL strings). Because an application can change the assignment of any prefix number except the boot prefix (*/), and certain initial prefix values might be left over from the previous application, beware of assuming a value for any particular prefix.

△ **Important** Do not depend on prefix 0 to contain any specific value, such as the name of the boot volume. Instead, use prefix 1 and prefix 9, which are both set to the full pathname of the directory containing the current application. If the string is more than 64 characters long, prefix 1 is set to a NULL string and prefix 9 contains the full string. △

Tables 2-5 through 2-7 show the initial values of the prefix numbers that a GS/OS application receives under the three different launching conditions possible on the Apple IIGS.

At all times during execution, GetName returns the filename of the current application, and GetBootVol returns the boot volume name.

■ **Table 2-5** Prefix values when a GS/OS application is launched at boot time

Prefix	Description
*	boot volume name
0	boot volume name
1	full pathname of directory containing current application, or NULL string if pathname contains more than 64 characters
2	*/SYSTEM/LIBS
3-8	null strings
9	full pathname of directory containing current application
10-12	.CONSOLE
13-31	null strings
@	if current application resides on an AppleShare volume, @ is set to the pathname of the user's directory on the file server; otherwise, @ is set to the same pathname as prefix 9

■ **Table 2-6** Prefix values when a GS/OS application is launched after a previous GS/OS application quits

Prefix	Description
*	unchanged from previous application
0	unchanged from previous application
1	full pathname of directory containing current application, or NULL string if pathname contains more than 64 characters
2-8	unchanged from previous application
9	full pathname of directory containing current application
10-12	.CONSOLE if the skip-std-prefixes flag (bit 13 of flags word) is 0, or unchanged from previous application if flag is 1
13-31	NULL strings
@	if current application resides on an AppleShare volume, @ is set to the pathname of the user's directory on the file server; otherwise, @ is set to the same pathname as prefix 9

■ **Table 2-7** Prefix values when a GS/OS application is launched after a ProDOS 8 application quits

Prefix	Description
*	boot volume name
0	set to the ProDOS 8 system prefix under previous application
1	full pathname of directory containing current application, or NULL string if pathname contains more than 64 characters
2	*/SYSTEM/LIBS
3-8	NULL strings
9	full pathname of directory containing current application
10-12	.CONSOLE
13-31	NULL strings
@	if current application resides on an AppleShare volume, @ is set to the pathname of the user's directory on the file server; otherwise, @ is set to the same pathname as prefix 9

Pathname prefixes at ProDOS 8 application launch

Table 2-8 shows the initial values of the ProDOS 8 system prefix and the pathname at location $0280 in bank $00 when a ProDOS 8 application is launched from GS/OS.

■ **Table 2-8** Prefix and pathname values at ProDOS 8 application launch

Condition	System prefix	Location $0280 pathname
Application launched at boot time	Boot volume name	Filename of current application
Application launched through enhanced ProDOS 8 QUIT call	Unchanged from previous application	Full or partial pathname given in QUIT call
Application launched after a GS/OS application has quit (if Quit call specified a full pathname)	Previous application's prefix 0/	Full pathname given in Quit call
Application launched after a GS/OS application has quit (if Quit call specified a prefix and a partial pathname)	Prefix specified in the Quit call	Partial pathname given in Quit call

Chapter 3 **Making GS/OS Calls**

This chapter describes the methods your application must use to make GS/OS calls. The current application, a desk accessory, and an interrupt handler are examples of applications that can make GS/OS calls.

Chapter 3 **Making GS/OS Calls**

This chapter describes the methods your application must use to make GS/OS calls. The current application, a desk accessory, and an interrupt handler are examples of applications that can make GS/OS calls.

GS/OS call methods

When an application makes a GS/OS call, the processor can be in emulation mode or full native mode, or any state in between (see the *Technical Introduction to the Apple IIGS*). There are no register requirements on entry to GS/OS. GS/OS saves and restores all registers except the accumulator (A) and the processor status register (P); these two registers store information on the success or failure of the call.

Calling in a high-level language

To make a GS/OS call from a high-level language, such as C, you supply the name of the call and a pointer to the parameter block.

Calling in assembly language

You can make GS/OS calls in assembly language using any of the following techniques:

- Macros. This technique uses macros defined by Apple Computer to generate in-line calls. Macro calls are the simplest to make and the easiest to read.
- In-line calls. This technique is similar to ProDOS 8.
- Stack calls. This technique is consistent with the way compilers generate code.

There is virtually no difference in the run-time performance of these three techniques; essentially, which one you use is a matter of personal preference. Each of these techniques is detailed separately in the following sections.

To make a GS/OS assembly-language call, your application must provide

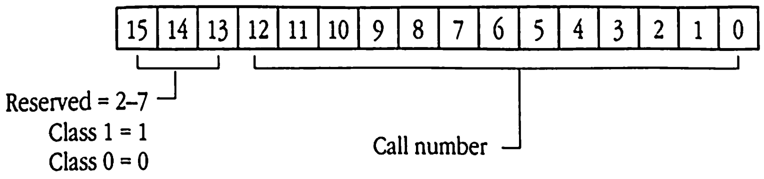
- a Jump to Subroutine Long (JSL) instruction (if you don't use the macro name) to the appropriate GS/OS entry point
- a 2-byte call number or the macro name of the call
- a 4-byte pointer to the standard GS/OS parameter block for the call; the parameter block passes information between the caller and the called function

The macro name or call number specifies the type of GS/OS call, as follows:

- Standard GS/OS calls: These calls allow you to access the full power of GS/OS; you should use them if you are writing a new application. Most of the description in this manual is devoted solely to these calls.

- ProDOS 16 calls: These calls, described in Appendix A of this book, are provided only for compatibility with ProDOS 16. (ProDOS 16 is described in the *Apple IIGS ProDOS 16 Reference*.)

Every GS/OS call that doesn't use the macro technique must specify the system call number and class in a parameter referred to in the next sections as `callnum`. The `callnum` parameter has the following format:



The primary call number is given in each call description. For example, the call number for the Open call is $10.

Thus, to make a standard GS/OS (class 1) Open call, your application uses the macro name or a `callnum` value of $2010 and a pointer to the standard GS/OS parameter block. To make a ProDOS 16–compatible (class 0) OPEN call, the caller uses a `callnum` value of $0010 and a pointer to the ProDOS 16–compatible parameter block.

Making a GS/OS call using macros

To make a standard GS/OS call using the macro technique, follow these steps:

1. Provide the name of the standard GS/OS call.
2. Follow the name with a pointer to the parameter block for the call.

GS/OS performs the function and returns control to the instruction that immediately follows the macro.

The following code fragment illustrates a macro call:

```
              _CallName_C1 parmblock  ;name of call
              bcs   error             ;handle error if carry set on return
              .
              .
              .
error                                 ;code to handle error return
              .
              .
              .
parmblock                             ;parameter block
```

Making an in-line GS/OS call

To make a standard GS/OS call using the in-line method, perform the following steps:

1. Perform a JSL to $E100A8, the GS/OS in-line entry point.
2. Follow the JSL with the call number.
3. Follow the call number with a pointer to the parameter block.

GS/OS performs the function and returns control to the instruction that immediately follows the parameter block pointer.

The following code fragment illustrates an in-line call:

```
inline_entry  gequ $E100A8       ;address of GS/OS in-line entry point
;
              jsl  inline_entry  ;long jump to GS/OS in-line entry point
              dc   i2'callnum'   ;call number
              dc   i4'parmblock' ;parameter block pointer
              bcs  error         ;handle error if carry set on return
                   .
                   .
                   .
error                            ;code to handle error return
                   .
                   .
                   .
parmblock                        ;parameter block
```

Making a stack call

To make a standard GS/OS call using the stack method, perform the following steps:

1. Push the parameter block pointer onto the stack (high-order word first, low-order word second).
2. Push the call number of the call onto the stack.
3. Perform a JSL to $E100B0, the GS/OS stack entry point.

GS/OS performs the GS/OS command and returns control to the instruction that immediately follows the JSL.

The following code fragment illustrates a stack call:

```
stack_entry gequ $E100B0       ;address of GS/OS stack entry point
;
            pea  parmblock|-16 ;push high word of parameter block pointer
            pea  parmblock     ;push low word of parameter block pointer
            pea  callnum       ;push call number
            jsl  stack_entry   ;long jump to GS/OS stack entry point
            bcs  error         ;handle error if carry set on return
                 .
                 .
                 .
error                          ;code to handle error return
                 .
                 .
                 .
parmblock                      ;parameter block
```

Including the appropriate files

If you are writing your application in assembly language, include the following files, as appropriate:

`E16.GSOS` and `M16.GSOS`	If you are making standard GS/OS calls
`E16.ProDOS` and `M16.ProDOS`	If you are making ProDOS 16–compatible calls

If you are writing your application in C, include one or both of the following files:

`GSOS.h`	If you are making standard GS/OS calls
`ProDOS.h`	If you are making ProDOS 16–compatible calls

△ **Important** In either language, if you include files to make both standard GS/OS and ProDOS 16–compatible calls, you must append the suffix `GS` to the standard GS/OS call names and parameter block type identifiers. △

GS/OS parameter blocks

A GS/OS parameter block is a formatted table that occupies a set of contiguous bytes in memory. The block 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 GS/OS call requires a valid parameter block (`parmblock` in the preceding examples), referenced by a 4-byte pointer. The application is responsible for constructing the parameter block for each call that it makes; the block can be anywhere in memory.

The formats of the fields for individual parameter blocks are presented in the detailed system call descriptions in Chapter 7.

Types of parameters

Each field in a GS/OS parameter block contains a single parameter, one or more words in length. Each parameter is an input from the application to GS/OS, a result that GS/OS returns to the application, or both an input and a result.

- An input can be either a numerical value or a pointer to a string or other data structure.
- A result is a numerical value that GS/OS places into the parameter block for the caller to use.

A pointer is the 4-byte address of a location containing data or the starting address of a buffer space in which GS/OS can receive or place data; that is, the pointer may point to a location that contains an input, a location that receives a result, or a location that both contains an input and receives a result.

Parameter block format

All standard GS/OS parameter blocks begin with a **parameter count,** which is a word-length input value that specifies the total number of parameters in the block. This allows you to vary the number of parameters in a call as needed, and also makes possible future parameter block expansion.

All parameter fields that contain block numbers, block counts, file offsets, byte counts, and other file or volume dimensions are 4 bytes long. Using 4-byte fields ensures that GS/OS can accommodate large devices using file system translators.

All parameter fields contain an even number of bytes, for ease of manipulation by the 16-bit 65C816 processor. 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 GS/OS 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.

△ **Important** The range of theoretically possible values defined by the length of a parameter 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 each file system can define its own permissible values for a field. △

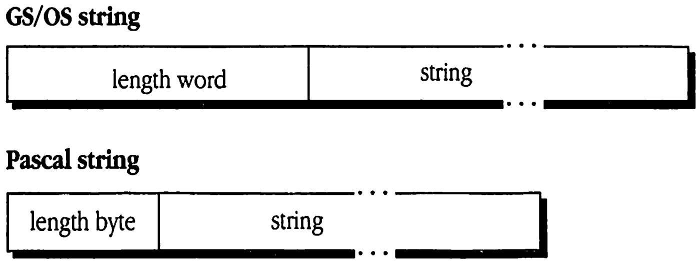
GS/OS string format

GS/OS strings resemble Pascal-style strings. A **Pascal string** begins with a **length byte** that defines the length of the string in bytes, followed by the string itself, with each character equal to one byte. A GS/OS string is very similar, except that it begins with a **length word** (two bytes) instead of a length byte. See Figure 3-1.

String parameters consist of a pointer parameter in the call's parameter block that points to a data structure containing the string. For standard GS/OS calls, that data structure varies depending on whether the string parameter is an input to or output from the call.

ProDOS 16–compatible calls use Pascal-style strings, with the exception of the GET_DIR_ENTRY call, which uses GS/OS strings.

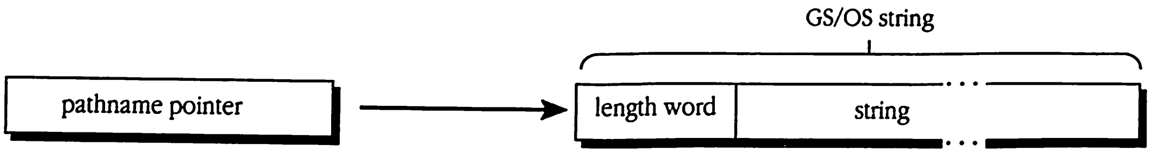
■ **Figure 3-1** GS/OS and Pascal strings



GS/OS input string structures

When a string is used as an input from an application to GS/OS, a pointer in the call's parameter block points to the low-order byte of the length word of the string, as shown in Figure 3-2.

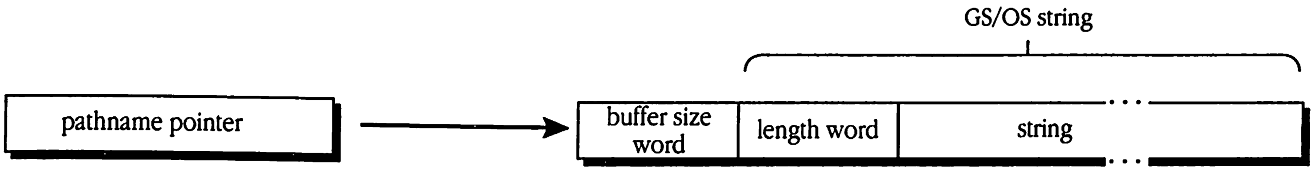
■ **Figure 3-2** GS/OS input string structure



GS/OS result buffer

When a string is returned as a result from a GS/OS call to an application, a pointer in the parameter block points to a buffer reserved for the result. This buffer starts with a buffer size word that specifies the total length of the buffer, including the buffer size word, as shown in Figure 3-3.

■ **Figure 3-3** GS/OS result buffer



How GS/OS returns the result depends on whether or not there is enough space in the buffer (excluding the buffer size word) to hold the output string. If there is enough space, the result is placed in the buffer just after the buffer size word.

If there is not enough space, GS/OS returns only the length word of the string, placing it immediately after the buffer size word. This gives the caller the opportunity to resize the buffer and reissue the call. The proper size is the value in the length word plus four (to account for the buffer size and string length words).

If the area is too small to contain the string, GS/OS returns error $4F (`buffTooSmall`) and sets the length word to the actual string length. In this case, the string field is undefined. The caller must add 4 to the returned string length to determine the total area needed to hold the buffer size word, the length word, and the string field.

The GetDirEntry call is an exception to the preceding rules. For this call only, if the result does not fit in the buffer, GS/OS copies as much of the string into the buffer as possible. The length word of the string will be set to the actual string length, not the size of the string placed in the buffer. Thus, the application may choose to use a partial string—for example, in a directory listing with a limited number of columns for the filename—or reissue the call to get a complete string.

Setting up a parameter block in memory

Each GS/OS call uses a 4-byte pointer to point to its parameter block, which can 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 a parameter block will be.

You can set up a GS/OS parameter block in memory in one of two ways:

- 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.
- Use Memory Manager and System Loader calls to place the block in memory, as follows:
    1. Request a memory block of the proper size from the Memory Manager. Use the procedures described in the *Apple IIGS Toolbox Reference.* The block should be either fixed or locked.
    2. 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).
    3. Set up your parameter block, starting at the address pointed to by the pointer obtained in step 2.

Conditions upon return from a GS/OS call

When control returns to the caller, the registers have the values shown in Table 3-1.

■ **Table 3-1** Registers on return from GS/OS

Register	Description
A	0 if call successful, error code if call unsuccessful
X	Unchanged
Y	Unchanged
S	Unchanged
D	Unchanged
P	Shown in Table 3-2
DB	Unchanged
PB	Unchanged
PC	Address of next instruction

"Unchanged" means that GS/OS initially saves, and then restores when finished, the value that the register had just before the call.

When control returns to the caller, the processor status and control bits have the values shown in Table 3-2.

■ **Table 3-2** Status and control bits on return from GS/OS

Register	Description
n	Undefined
v	Undefined
m	Unchanged
x	Unchanged
d	Unchanged
i	Unchanged
z	0 if call unsuccessful, 1 if call successful
c	0 if call successful, 1 if call unsuccessful
e	Unchanged

The n flag is undefined here; under ProDOS 8, it is set according to the value in the A register.

Checking for errors

When control returns to your application, the carry bit will be set to 1 if an error occurred, and the error code (if any) will be in register A. You can thus use a Branch if Carry Set (BCS) instruction to branch to an error-handling routine, and then pick up the error code from register A.

Fatal GS/OS errors are handled by the GS/OS Error Manager. When a fatal error occurs, the GS/OS Error Manager displays a failure message on the screen and halts execution of GS/OS.

The errors that specifically apply to a particular call are listed as part of the call description in Chapter 7, "GS/OS Call Reference." The general GS/OS errors shown in Table 3-3 can occur for almost any of the calls. You might want to invoke special error handlers to handle these conditions.

■ **Table 3-3** General GS/OS errors

Number	Error	Description
$01	`badSystemCall`	bad GS/OS call number
$04	`invalidPcount`	parameter count out of range
$07	`gsosActive`	GS/OS is busy
$53	`paramRangeErr`	parameter out of range
$54	`outOfMem`	out of memory

Chapter 4 **Accessing GS/OS Files**

The most common use of GS/OS is to access files that contain data on a storage medium such as a floppy disk, a hard disk, or a CD-ROM. A file is an ordered collection of bytes that has several attributes, including a name and a file type.

GS/OS tries to free you, the application programmer, from knowing more about files and file systems than you want to. GS/OS has been built on the theory that, in most cases, you only want to assign the attributes that are critical to the function of the file, and that you're not really interested in where the user chooses to store the file.

Thus, this chapter assumes that you want to access files using the simplest possible method. Using this method, you call the Apple IIGS Toolbox routines SFPutFile2 or SFGetFile2 (from the Standard File Operations Tool Set) to construct the name of the file the user wishes to create or open. With this method, you don't have to worry about the pathname to the file, since GS/OS is able to construct the full pathname to the file automatically.

If you want to build the pathname yourself, GS/OS also gives you that capability; see Chapter 5, "Working With Volumes and Pathnames."

An overview of simple file access

This section summarizes the simplest method you can use to access files. Each step is described in more detail in the rest of this chapter.

To use this method, perform the following steps:

1. If the user is creating a new file, call the tool set routine SFPutFile2 to get a pointer to the pathname of the file that the user wishes to create. Save the pointer, and use it in a GS/OS Create call to place the file on the disk. For more information, see the section "Creating a File" later in this chapter.
2. If the user is opening an existing file, call the tool set routine SFGetFile2 or SFMultiGet2 to get a pointer to the pathname of the file or files that the user wishes to open. Save the pointer for each file, and use it in a GS/OS Open call to open each file. For more information, see the section "Opening a File" later in this chapter.
3. When you make the Open call, you request the access your application wants to the file. At this point, you determine what access other users can have to the file. With the advent of system software version 5.0 and its ability to support AppleShare, you should carefully consider how best to implement file access. For more information, see the section "Sharing Open Files in an AppleShare Environment" later in this chapter.

   In the Open call, GS/OS returns a reference number that you must save to refer to the file and the actual access you obtained to the file.
4. After a file has been opened, you can do the following tasks:
   - □ read and write data to the file by making Read and Write calls
   - □ move or get the current reading and writing position in the file by making SetMark and GetMark calls
   - □ move or get the current EOF by making SetEOF and GetEOF calls
   - □ enable newline mode, which terminates a read if the read encounters one of the specified newline characters, or disable that mode
   - □ write all buffered information to storage to ensure data integrity by making a Flush call
5. When you have finished working with the file, close it by making a Close call.

This chapter provides you with some information on how to use the file access calls. For more details on each individual call, see Chapter 7, "GS/OS Call Reference."

Creating a file

When you want your application to create a file, issue a GS/OS Create call. When you issue that call, you assign some important characteristics to the file:

- A pathname, which must place the file within an existing directory. As already mentioned, if you use the toolbox routine SFPutFile2, you only have to save the pathname pointer it returns and supply that pointer in your Create call. If you want to build the pathname yourself, see Chapter 5.
- File access privileges, which determine whether or not the file can be written to, read from, destroyed, or renamed, and whether the file is invisible.
- A file type and an auxiliary type, which indicate to other applications the type of information to be stored in the file. They do not affect, in any way, the contents of the file.
- A storage type, which determines the physical format of the file on the disk. There are four different formats: one is used for directory files, the other three for non-directory files.
- The size of the file and the size of the resource of the file, which are used to preallocate disk storage for the file. Under most circumstances, you can leave these parameters set to their default of 0.

When GS/OS creates the file, it places the properties listed above on disk, along with the current system date and time (called creation date and creation time). Once created, a file remains on disk until it is deleted (using the Destroy call).

Opening a file

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 a pathname to a previously created file; the file must be on a disk mounted in a disk drive or on an AppleShare volume that has been mounted, or GS/OS returns an error. As already mentioned, you can query the user for the filename by using the SFGetFile2 routine in the Standard File Operations Tool Set of the Apple IIGS Toolbox.

The Open call returns a reference number that your application must save; any other calls you make affecting the open file must use the reference number. The file remains open until you use the Close call.

Multiple Open calls can be made to files on block devices for read-only access; in that situation, the file remains open until you make a Close call for each file you opened.

GS/OS allows any number of files to be open at a time. The only limit is imposed by the amount of total available memory and the number of available reference numbers. However, each open file requires some system overhead, so in cases where memory is in short supply, your application might want to keep as few files open as possible.

Be aware of the differences between a file on disk and portions of an open file 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 GS/OS 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.

A final consideration when opening a file is the access that users are allowed to the file, as discussed in the next section. Since GS/OS works in an AppleShare environment, that access is critical in determining whether more than one user can access a file at once.

Sharing open files in an AppleShare environment

ProDOS has traditionally been a single-user, single-computer operating system and file system. With the addition of AppleShare support to GS/OS, many users (using many types of computers) can share files on the file server simultaneously. Thus, if your program opens files, you must decide whether another user trying to open the same file should be allowed access to the file, and open the file in an appropriate manner. The standard GS/OS Open call allows you to specify the access you require to the open file in the `requestAccess` parameter, as shown in Table 4-1.

△ **Important** You should normally specify a nonzero value for the `requestAccess` parameter. This way, files can be shared if possible. Further, if the Open call does not return an error, you know that you have the access to the file that you requested. △

◆ *Note:* Your application can exercise greater control over the access that other users are permitted on AppleShare volumes by using the SpecialOpenFork FSTSpecific call, described in Chapter 15, "The AppleShare FST."

- **Table 4-1** Access attributes and their implications

`requestAccess`	Access	Description
$0001	Read only, deny write	This setting allows your application to read the file, and allows other users to read the file, but doesn't allow them to write to the file, so that the data you are reading doesn't change.
$0002	Write only, deny read/write	This setting allows your application to write to the file, but doesn't allow other users to read or write to the file.
$0003	Read/write, deny read/write	This setting allows your application to read and write to the file, but doesn't allow other users to read or write to the file.
$0000	As permitted	This setting first tries to open the file for read/write; if that fails, it tries read-only; if that fails, it tries write-only. There is no way of knowing what access you have to the file.

Thus, your application can do one of the following:

- allow a single user to open a file for reading and writing
- allow multiple users to open a file only for reading
- allow multiple users to open a file for reading and writing as necessary

Each of these options is described in the following sections.

Allowing single users to open the file for reading and writing

If you set the `requestAccess` parameter to 0, or don't even supply the field in the Open call (it is optional), only the first user can open the file; the rest get an error when they try to open the file (error $4E, `invalidAccess`). This situation represents the pre–AppleShare aware state of GS/OS, and prohibits your application from allowing more than one user access to the file on the network. Thus, under normal circumstances, you should try to allow the options described in the next section.

Allowing multiple users to open a file for reading

You can easily allow users to open files for reading only; you don't even have to take any special AppleShare actions as long as you follow safe programming practices.

△ **Important** The most important safe programming practice in an AppleShare environment is to not disable interrupts. For more information, see the section "GS/OS and Interrupts" in Chapter 2. △

When GS/OS opens and loads your application, it makes sure the application file itself can be shared by several computers. Thus, to ensure that your application can be executed by more than one computer at a time, you need only be careful about how you open other files. The guidelines you should use are as follows:

- Because other users use the same copy of the application, do not write to the application files—for example, to save configuration information.
- If your application always opens certain files—such as error message files—be certain that the application opens them read-only, so that other users can open the same files.
- Be careful to assign only the minimum access required; that is, if your application only needs read access to a file, do not assign read/write access.
- If your application can handle several different kinds of access to the file, try those different access modes individually until you get one of the desired kinds of access. For example, if you can handle either read/write or read-only access but prefer read/write, try opening the file with `requestAccess` parameter = 3 (read/write). If this fails, try opening with `requestAccess` parameter = 1 (read-only).
- Don't assume that `requestAccess` = 0 allows your application read and write access; other users may have opened the file, or access privilege settings set by other applications may restrict access.

For example, an adventure game might want to load a map of rooms in a dungeon. In this example, the application really only needs to read the contents of the file, and does not need to modify the contents. Since that is true, the application should open the file read-only (`requestAccess` = 1). Several users can then run the program at the same time and open the dungeon file successfully, since the read-only open allows others to open the file read-only.

As a second example, consider a file copying program (like the Finder). It opens the source file read-only, so that other users can copy it or use it. It opens the destination file write-only (`requestAccess` = 2), since it only needs to write to the file; thus, no other user is allowed to read or write to the copy while it is being written. Note that opening the destination file for reading and writing causes the open to fail if access privileges to the file prevent read access (such as if the file is in a drop box).

As a third example, consider a word-processing program. It must be able to read from the file so that it can be displayed or printed, and write to the file so that it can be edited and saved. In this case, the application opens the file with `requestAccess` = 3 (read and write). Also, the file must be kept open the entire time it is being edited. If it isn't, another user can open the file for editing after it has been closed. In that case, that user's version is saved, and the first version is overwritten.

Allowing multiple users to open a file for reading and writing

To allow multiple users the ability to access and possibly change data in the same file at the same time, your application must make AppleShare calls. A typical example is a database program that lets several users view and edit records at the same time. In this case, the read/write protections are applied to individual records instead of the entire file. To do this, you use commands specific to the AppleShare FST; see Chapter 15, "The AppleShare FST," for more information.

Using the `@` prefix

The `@` prefix is a system prefix defined when your application is launched. If the application was launched from an AppleShare volume, this prefix is set to the name of the user's directory on the file server. If the application was launched from a non-AppleShare volume, it will be set to the name of the directory containing the application.

You can use the `@` prefix as part of the pathname to save configuration information in a separate place for each user. For example, if your program were called *Fred,* you might use the pathname `@:Fred.Config` for storing preferences and configuration data for each user.

◆ *Note:* Your application still must handle the case where another workstation may be accessing the same configuration file (for example, when two users at different workstations log in using the same user name).

Working on open files

When you open a file, some of the file's characteristics are placed into a region of memory. Several of these characteristics are accessible to calling applications by way of GS/OS calls, and can be changed while the file is open.

This section describes the GS/OS calls that work with open files.

Reading from and writing to files

Read and write calls to GS/OS transfer data between memory and a file. For both calls, the application must specify the following information:

- the reference number of the file (assigned when the file was opened)
- the location in memory of a buffer that contains, or is to contain, the transferred data
- the number of bytes to be transferred
- the cache priority, which determines whether or not the blocks involved in the call are saved in RAM for later reading or writing

When the request has been carried out, GS/OS 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 EOF has been reached). Read requests can also terminate when a specified character is read.

Setting and reading the EOF and mark

Your application can place the EOF anywhere in the file, 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 SetEOF and SetMark calls. The current values of the EOF and the mark can be determined using the GetEOF and GetMark calls.

Enabling or disabling newline mode

Your application can use the Newline call to indicate that read requests terminate on a specified character or one of a set of specified characters. For example, you can use this capability to read lines of text that are terminated by carriage returns.

Newline mode is disabled by default when a file is opened.

Examining directory entries

Your application does not need to know the details of directory format to access files with known names. You need to examine a directory's entries only when your application is performing operations on unknown files (such as listing the files in a directory). The GS/OS call you use to examine a directory's entries is GetDirEntry; for more details, see the GetDirEntry call in Chapter 7, "GS/OS Call Reference."

Flushing open files

The GS/OS Flush call writes any unwritten data from an open file's I/O buffer to the file, and updates the file's size in the directory. However, it keeps the reference number (returned from the Open call) and the file's buffer space active, and thus allows continued access to the file.

When used with a reference number of 0, Flush normally causes all open files to be flushed. Specific groups of files can be flushed using the system file level (see "Setting and Getting File Levels" later in this chapter).

Closing 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 that was assigned when the file was opened.

The Close call writes any unwritten data from memory to the file and updates the file's size in the directory, if necessary. Then it frees the file's buffer space for other uses and releases the file's reference number and file control block. To access the file again, you must 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. You can prevent this situation from occurring by using the Flush call.

Setting and getting file levels

When a file is opened, it is assigned a file level equal to the current value of the **system file level.** The system file level determines which files are closed or flushed whenever a Close or Flush call is made with a reference number of 0. GS/OS closes or flushes only those files whose levels are greater than the current system level.

The system file level feature can be used, for example, by a controlling program such as a development system shell to implement an EXEC process:

1. The shell opens an EXEC program file when the level is $00.
2. The shell then sets the level to, for example, $07.
3. The EXEC program opens whatever files it needs.
4. The EXEC program executes a GS/OS Close command with a reference number of $0000 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 SetLevel call; you obtain the current value by making a GetLevel call.

Working on closed files

This section describes some of the functions of the GS/OS calls that work with closed files. Some of the calls that work with pathnames are performed on closed files; see Chapter 5, "Working With Volumes and Pathnames," for more information.

Clearing backup status

Whenever a file is altered, GS/OS automatically changes the information about the file's state to indicate that it has been changed but not backed up. Thus, an application that performs backups can check the backup status to determine whether or not to back up the file.

If you want to change the information about the backup status to indicate to GS/OS that the file does not need to be backed up, use the ClearBackupBit call. This resets the backup status so that it looks to GS/OS as if the file had not been altered. For example, you can use this technique in a word-processing application when the user deletes something from the file but then decides to undo the change; issuing the ClearBackupBit call prevents the file from being backed up.

Deleting files

If you want your application to delete a file on disk, you can use the GS/OS Destroy call to delete the file. You can use this technique only on subdirectories, standard files, and extended files; you can't use the technique to delete volume directories or character-device files.

◆ *Note:* Character-device files are treated somewhat differently; see Chapter 14 for more information.

Setting and getting file characteristics

Certain characteristics of an open or closed file can be retrieved or modified by the standard GS/OS calls SetFileInfo and GetFileInfo.

△ **Important** Although SetFileInfo and GetFileInfo calls can be performed on open files, you might not get back the information you want. It's safer to perform these calls only on closed files. △

These characteristics include

- access to the file
- file type and auxiliary type
- creation time and date
- modification time and date
- a pointer to an option list for FST-specific information (see Part II of this book, "The File System Level," for more information about FSTs)

An example of how you can use SetFileInfo and GetFileInfo is given in the section "Copying Files" later in this chapter.

Changing the creation and modification dates and times

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. Each field contains the time and date information in the format shown in Table 4-2.

- **Table 4-2** Date and time format

Item	Byte position
Seconds	Byte 1
Minutes	Byte 2
Hour	Byte 3
Year	Byte 4
Day	Byte 5
Month	Byte 6
Null	Byte 7
Weekday	Byte 8

This date and time format is the same as that used by the `ReadTimeHex` IIGS Toolbox call in the Miscellaneous Tool Set.

Since data in the file's directory entry itself is 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 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 directories—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 dates and times as the original file, and not the date and time at which the copy was created. See the following section for more information.

Copying files

To copy single files to a disk, or to copy files to and from file servers where you have full access, perform the following steps:

1. Make a GetFileInfo call on the source file (the file to be copied) to get its creation and modification dates and times.
2. Make a Create call to create the destination file (the file to be copied to).
3. Open both forks of the source and destination files. Use Read and Write calls to copy the source to the destination.
4. Make a Flush call on the destination fork or forks.
5. Make a SetFileInfo call on the destination file, using all the information returned from GetFileInfo in step 1. This sets the modification date and time values to those of the source file.
6. Close both files.

To copy multiple files, you may want to use the GS/OS cache mechanism, described in the next section.

If you are writing an application similar to the Finder™, you will also need to deal with drop folders. To copy files into a drop folder on a server where you have access to make changes but not to see files or to see folders, perform the following steps:

1. Copy everything into the drop folder.
2. Just before closing, change the name of the owner of the copies to become the name of the owner of the drop folder.

Caching files

All blocks on a disk that can be read by GS/OS can be classified into one of two categories. Application blocks are all blocks on the disk that are contained in any file (except a directory file), while system blocks are other blocks on the disk. System blocks belong to the file system and include directory blocks, bitmap blocks, and other housekeeping blocks specific to the file system.

GS/OS always maintains at least a 16 KB cache, even if the user has used the Control Panel to set the disk cache size to 0 KB. When the system (usually an FST) reads a system block, the block is identified as a candidate for caching and is cached if possible. Applications define blocks as candidates for caching by using the `cachePriority` field of many standard (class 1) GS/OS calls. Note that ProDOS 16–compatible (class 0) calls do not have this field; thus, applications using exclusively class 0 calls cannot cache any application blocks.

GS/OS usually makes the best use of the cache automatically, freeing your application from the duty of caching system blocks. However, if your application normally reads the same portion of a document file over and over again, it can benefit from the caching mechanism.

For example, an application that does not limit document size to available memory often keeps only a portion of a document in memory at any given time. The beginning of the application's document file might contain a header that points to various parts of the document file (for example, chapters for a word processor, report formats for a database manager, or individual pictures for an animation program). This document header is probably not very long, but the application usually needs to read it quite often to quickly access various portions of the document file. This header is a prime candidate for caching, since it will be read many times during the life of the application.

The best general rule is to not cache complete files, but instead cache only those portions of your document files that will be read from disk many times.

For example, if you are attempting to cache a 40K file (80 512-byte blocks), and the cache is set to less than 40K, the entire cache will be written through, kicking out all system blocks currently cached. A cache of this size slows system performance for little gain, since the entire file cannot be cached anyway. Even if the cache is large enough to hold the entire file, you are needlessly storing a duplicate copy of a single file (by reading it into memory that you obtain from the Memory Manager, as well as asking GS/OS to keep a copy in the cache).

To further improve disk-caching performance, you can use the write-deferral mechanism, described in the next section.

Using the write-deferral mechanism

GS/OS provides a write-deferral mechanism that allows you to cache disk writes in order to increase performance.

To use this technique, perform the following steps:

1. Start the write-deferral session by making a GS/OS BeginSession call. Deferred blocks that are written to the cache are locked and cannot be purged until the EndSession call.

△ **Important** If you have started a write-deferral session, and have written files to the disk, do not allow the user to eject the disk until you end the write-deferral session; otherwise, you could damage the disk files. Make sure that you place an EndSession call in the flow of both a normal and an abnormal exit. △

2. Copy the files.
3. If the cache fills up with deferred blocks, the GS/OS Autoflush feature automatically issues an EndSession call, which immediately writes all deferred blocks to disk. GS/OS then automatically issues a BeginSession call to restart the write-deferral session.
4. End the write-deferral session by making a GS/OS EndSession call.

The SessionStatus call also allows you to check whether a write-deferral session is currently in force.

△ **Important** The price of the increased performance is increased caution. Do not allow your application to exit while a write-deferral mechanism is in force; you could harm the data integrity of any open disk files. Make sure that you place an EndSession call in the flow of both a normal and an abnormal exit. △

Chapter 5 **Working With Volumes and Pathnames**

You can usually avoid working with volumes, pathnames, and devices in detail; GS/OS can free you from keeping track of exactly where files reside. As discussed in Chapter 4, if you use the Apple IIGS Standard File Operations Tool Set routines SFPutFile2 and SFGetFile2, you don't need to know where a file is, since these routines tell GS/OS where the file is located.

In some situations, however, you may not be able to or may not want to use SFPutFile2 and SFGetFile2. For example, you might need or want more control if your application has any of the following characteristics:

- It is text-based (and thus unable to access SFPutFile2 and SFGetFile2).
- It needs to check whether particular files are in the appropriate directories (for example, when the data files for an application need to be in the same directory as the application).
- It builds its own pathnames (for example, if you want to present a list of all mounted volumes to the user).

In any of these cases, you have to understand more about pathnames and volumes, and just a little bit more about devices. This chapter discusses the concepts that you need to understand about entities, and the GS/OS calls that allow you to work with them.

◆ *Note:* This chapter doesn't discuss direct access to devices; for that information, see the *GS/OS Device Driver Reference.*

Chapter 5 Working With Volumes and Pathnames

You can usually avoid working with volumes, pathnames, and devices in detail; GS/OS can free you from keeping track of exactly where files reside. As discussed in Chapter 4, if you use the Apple IIGS Standard File Operations Tool Set routines SFPutFile2 and SFGetFile2, you don't need to know where a file is, since these routines tell GS/OS where the file is located.

In some situations, however, you may not be able to or may not want to use SFPutFile2 and SFGetFile2. For example, you might need or want more control if your application has any of the following characteristics:

- It is text-based (and thus unable to access SFPutFile2 and SFGetFile2).
- It needs to check whether particular files are in the appropriate directories (for example, when the data files for an application need to be in the same directory as the application).
- It builds its own pathnames (for example, if you want to present a list of all mounted volumes to the user).

In any of these cases, you have to understand more about pathnames and volumes, and just a little bit more about devices. This chapter discusses the concepts that you need to understand about entities, and the GS/OS calls that allow you to work with them.

◆ *Note:* This chapter doesn't discuss direct access to devices; for that information, see the *GS/OS Device Driver Reference.*

Volumes

Some GS/OS calls are designed to allow you to work directly with volumes; they are described in the following sections.

Getting volume information

GS/OS provides the Volume call to retrieve information about the volume currently mounted in a specified device. You can retrieve the following information:

- the name of the volume
- the total number of blocks on the volume
- the number of free blocks on the volume
- the file system contained on the volume
- the size, in bytes, of a block on the volume

An example of the use of the Volume call is given in the next section.

Building a list of mounted volumes

If you want your application to build a list of all the mounted volumes, you need to use the GS/OS calls DInfo and Volume, as follows:

1. To determine the names of the current devices, make DInfo calls for device 1, device 2, and so on until GS/OS returns error $11 (invalid device number). DInfo returns the name of the device associated with each device number (see Chapter 7 for details on the DInfo call).
2. Once you have the device name, you can use the GS/OS Volume call to obtain the name of the volume currently mounted on the device.

You can also continue from this point to examine directory entries and build the pathname to a file. See the section "Building Your Own Pathnames," later in this chapter, for more information.

Getting the name of the boot volume

If you need to determine the name of the volume from which GS/OS was booted, use the standard GS/OS call GetBootVol to retrieve a pointer to the volume name. That name is equivalent to the prefix specified by `*:`. For example, an application can start up QuickDraw™ II and the Event Manager and then use the GetBootVol call to check if the boot volume is on line. This allows the application to use the toolbox call `TLMountVol` to put up a custom dialog box if the boot volume is off line.

Formatting a volume

GS/OS provides two format options to applications.

- The GS/OS Format call attempts to format the medium; this method is necessary when your application can't read the existing volume.
- The GS/OS EraseDisk call assumes that a formatted medium already exists in the appropriate device, and writes new boot blocks, directory, and bitmaps to the medium. EraseDisk is usually faster than Format, but it requires that the medium already be formatted. You can use this call, for example, to quickly make all of the space reusable on a disk that can already be read by your application.

You have to provide a device name to either call, so you need to use the GS/OS DInfo call at some point to find out the device name.

After you issue the EraseDisk or Format call, GS/OS takes control, and presents a graphic or text interface that allows the user to choose the file system to be used to format the volume.

◆ *Note:* If you don't want to give the user the option of selecting the file system to be placed on the volume, you can specify the file system as a parameter to the EraseDisk or the Format call.

For GS/OS to present the graphic user interface, your application has to meet the following requirements:

- The Apple IIGS Toolbox Desk Manager must be active; by implication, all of the tool sets on which the Desk Manager depends must also be active (see the *Apple IIGS Toolbox Reference*).
- The List Manager must be active.
- For the graphics tools to run, 64 KB of RAM must be available.
- The Super Hi-Res screen must be currently displayed.

If all of these requirements are met, GS/OS presents the graphic interface to the user; if any one of the requirements is not met, GS/OS presents the text interface to the user.

Pathnames

If you need to, you can work directly with the pathname of a file. The following sections describe the pathname capabilities of GS/OS.

Setting and getting prefixes

You can use standard GS/OS calls to set and retrieve the prefix assignments manually. The SetPrefix call explicitly sets one of the numbered prefixes to the prefix you want, and the GetPrefix call returns the current value of any of the numbered prefixes.

△ **Important** SetPrefix and GetPrefix cannot be used to change or retrieve the boot volume prefix or the `@` prefix. To retrieve the name of the boot volume prefix, use the GS/OS GetBootVol call, described earlier in this chapter and detailed in Chapter 7. Your application cannot change the prefix of the boot volume at all. However, if the user renames the boot volume, GS/OS automatically adjusts all pathnames to reflect the changed prefix. To retrieve the name of the `@` prefix, use the GS/OS ExpandPath call. △

Changing the path to a file

GS/OS allows you to change the path to a specified file. From the user's point of view, this "moves" the file from the old directory to the new directory, even though the physical location of the file does not change. In addition, if you change the path to a directory, all files and directories that are in that directory also have their paths changed.

To change the pathname, use the standard GS/OS call ChangePath. For detailed information about how to change the path, see the discussion of the ChangePath call in Chapter 7.

Expanding a pathname

GS/OS allows you to expand a partial pathname into its corresponding full pathname. To expand a pathname, use the standard GS/OS call ExpandPath. For detailed information about how to expand a path, see the discussion of the ExpandPath call in Chapter 7.

Building your own pathnames

If you want your application to build a pathname by itself, you need to use several GS/OS calls.

1. To determine the names of the current devices, make DInfo calls for device 1, device 2, and so on until GS/OS returns error $11 (invalid device number). The DInfo call returns the name of the device associated with each device number (see Chapter 7 for details on DInfo).
2. Once you have the device name, use the GS/OS Volume call to obtain the name of the volume currently mounted on the device.
3. Open the volume directory by using the GS/OS Open call.
4. Get the directory entries for the files by using successive GetDirEntry calls.

Devices

A **device** is a piece of equipment that transfers information to or from the Apple IIGS. Disk drives, printers, mouse devices, and joysticks are external devices. The keyboard and screen are also considered 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.

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

Device names

GS/OS identifies devices by device names. A GS/OS device name is a sequence of 2 to 32 characters beginning with a period (`.`).

Your application must encode device names as sequences of 7-bit ASCII codes, with the device name in all uppercase letters and with the most significant bit off. The slash character (`/`; ASCII 2F) and the colon (`:`; ASCII 3A) are always illegal in device names.

Block devices

A **block device** reads and writes information in multiples of one block of characters 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; for example, disk drives are block devices.

◆ *Note:* GS/OS supports any RAM disk that behaves like a block device in all respects just as if it were a block device.

GS/OS supports two different kinds of access to block devices:

- File access, where you make a GS/OS Read or Write call, and GS/OS does the work of finding and accessing the device. This process is described in Chapter 4.
- Direct access, which you can use if your application needs to access blocks directly. The calls that directly access devices are briefly summarized in Chapter 7 and discussed in detail in the *GS/OS Device Driver Reference.*

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.

△ **Important** Be aware of character devices. When prompted for a filename or pathname, a user might enter a pathname to a character device. Error $58 (`notBlockDev`) can protect you against this on many calls, including Create, but you must still take precautions. The DInfo call tells you if a device is a character device or block device; bit 7 of the `characteristics` word is set if the device is a block device. △

GS/OS supports character devices through both direct access and file access. For more information, see Chapter 14.

Direct access to devices

Generally, you don't need to do the work of accessing devices directly. For some special applications and devices, however, you may want to take over that work; if you do, you'll have to know a lot more about devices. See the *GS/OS Device Driver Reference* for that information.

Device drivers

Block devices generally require device drivers to translate a file system's logical block device model into the tracks and sectors by which information is actually stored on the physical device. Character devices also require drivers.

There are two types of GS/OS drivers: loaded drivers, which are RAM-based, and generated drivers, which are constructed by GS/OS. Device drivers are discussed in the *GS/OS Device Driver Reference.*

Chapter 6 **Working With System Information**

Several GS/OS calls provide access to information about GS/OS. This chapter introduces you to them.

Chapter 6 **Working With System Information**

Several GS/OS calls provide access to information about GS/OS. This chapter introduces you to them.

Setting and getting system preferences

GS/OS provides a preferences word that allows your application to make the following choices:

- If your application is using pathname calls, it can determine whether it will handle error $45 (`volNotFound`) itself, or whether it will have GS/OS handle those errors.
- It can display either a standard Volume Mount dialog box or a Volume Mount dialog box without a Cancel button.
- It can either suppress error dialog boxes that contain only one button or allow them to be displayed.

For a more detailed discussion of how to set up the preferences word and any other options available in that word, see the description of SetSysPrefs and GetSysPrefs in Chapter 7, "GS/OS Call Reference."

Checking FST information

If you want to check the information for a specific FST, you can use the standard GS/OS call GetFSTInfo. That call returns the following information about the FST:

- name and version number of the FST
- some general attributes of the FST, such as whether GS/OS will change the case of pathnames to uppercase before passing them to the FST, whether GS/OS will strip the high-order bit before passing on a filename, and whether it is a block or character FST
- block size of blocks handled by the FST
- maximum size of volumes handled by the FST
- maximum size of files handled by the FST

For a more detailed discussion of how to retrieve the information, see the GetFSTInfo call in Chapter 7, "GS/OS Call Reference." For more information about FSTs, see Part II.

Another call that is useful for a specific FST is, not surprisingly, FSTSpecific. FSTSpecific is a call that can be defined individually for any file system translator. For a more detailed discussion, see the FSTSpecific call in Chapter 7, "GS/OS Call Reference." For more information about how the ProDOS FST, for example, uses the FSTSpecific Call, see Chapter 12, "The ProDOS FST."

Finding out the version of the operating system

If your application depends on a feature of GS/OS that was implemented in a version later than 2.0, you can use the standard GS/OS call GetVersion to retrieve the version number of GS/OS. For more detailed information about how to retrieve the version number, see the GetVersion call in Chapter 7, "GS/OS Call Reference."

Getting the name of the current application

To get the filename of the application that is currently executing, you can use the standard GS/OS call GetName. For example, if you want your application to display its own name to the user, you can use GetName to get its current name (remember, the user can rename application files).

For more detailed information, see the GetName call in Chapter 7, "GS/OS Call Reference."

Getting reference numbers and information

You can use the following GS/OS calls to get information about the reference number, access attributes, and full pathname of an open file.

- To get the reference number of the last Open call to any of the three standard prefixes (`10`, `11`, and `12`), use the GetStdRefNum call.
- To get the reference number and access attributes for an open file, use the GetRefNum call.
- To get the access attributes and full pathname for an open file, use the GetRefInfo call.

For more detailed information about how to use those calls, see their descriptions in Chapter 7, "GS/OS Call Reference."

Getting the current device number

You can use the GetDevNumber call to get the device number of a device identified by device name or volume name. Only block devices may be identified by volume name, and then only if the named volume is mounted. Most other device calls refer to devices by device number.

Working with the notification queue

The notification queue allows applications to be notified when certain operating-system events occur. Two standard GS/OS calls, AddNotifyProc and DelNotifyProc, add and delete notification procedures from the notification queue. For more detailed information about how to use those calls, see the AddNotifyProc and DelNotifyProc calls in Chapter 7, "GS/OS Call Reference."

A notification procedure starts with the header shown in Table 6-1.

The code in `Proc_Entry` must set the direct-page and data bank registers as needed; however, it does not need to save and restore the entry values.

In general, GS/OS calls cannot be made by the notification procedure (because GS/OS is busy). The exceptions are the "switch GS/OS to ProDOS 8" and "switch ProDOS 8 to GS/OS" events; GS/OS calls are allowed during these events so that files may be opened or closed.

△ **Important** Do not make a Quit call from a notification procedure under any circumstances. △

Notification procedures may be called during an interrupt; therefore, keep the code short (set a flag or set up a signal).

Parameters are passed to notification procedures in the A, X, and Y registers. The parameters for each event are shown in Table 6-2.

■ **Table 6-1** Notification procedure header

Name	Size	Description
Reserved	Long	Reserved (link to next task in queue)
Reserved	Word	Reserved
`Signature`	Word	$A55A signature
`Event_flags`	Long	Bit flags indicating which events to call the procedure for, as follows: **Bit** **Event** 0 Reserved 1 Switch GS/OS to ProDOS 8 2 Switch ProDOS 8 to GS/OS 3 Disk insert 4 Disk eject 5 Shutdown 6 Volume change (writing occurred) 31–37 Reserved
`Event_code`	Long	The current event code will be put here by GS/OS. This will be equivalent to the value of `Event_flags` with the appropriate bit set (for example, disk insert = $00000008).
`Proc_Entry`	As needed	The code for the notification procedure. GS/OS will perform a `JSL` to the code in full native mode.

■ **Table 6-2** Notification procedure parameters

Event	A register	X register	Y register
Switch GS/OS to ProDOS 8	Undefined	Undefined	Undefined
Switch ProDOS 8 to GS/OS	Undefined	Undefined	Undefined
Disk insert	Device number	Undefined	Undefined
Disk eject	Device number	Undefined	Undefined
Shutdown	Undefined	Undefined	Undefined
Volume change	GS/OS call number*	Device number	Undefined

*The call number will be 0 when the AppleShare FST detects that a server volume has been modified.

Using the optionList parameter

Many GS/OS calls have an output longword pointer parameter called optionList. The parameter allows you to point to additional data that your application requires. The structure of the buffer pointed to by the optionList parameter must be as follows:

Offset		Size and type
$00	totalSize	Word input value
$02	reqSize	Word output value
$04	fileSysID	Word output value
$06	outputData	Output; length determined by call

totalSize Word input value: Total size of buffer, including this size word. If the total size is less than 4, error $53 (paramRangeErr) will be returned.

reqSize Word output value: Required size of buffer minus 4 (that is, not including the totalSize and reqSize parameters). If the data will not fit in the space provided, error $4F (buffTooSmall) will be returned; in this case, a new buffer should be allocated using the reqSize plus 4.

fileSysID Word output value: File system ID. The file system IDs are as follows:

$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E–$FFFF	Reserved
$0007	Lisa®		

outputData Output value: Data specific to call and file system.

If the optionList parameter is NIL, no error will be returned.

Chapter 7 **GS/OS Call Reference**

This chapter provides the detailed description for all GS/OS calls, arranged in alphabetical order by call name. Each description includes these elements:

- the call'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 and description

The diagram accompanying each call description is a simplified representation of the call's parameter block in memory. The width of the parameter block diagram represents one byte; successive tick marks down the side of the block represent successive bytes in memory. Each diagram also includes these features:

- **Offset:** Hexadecimal numbers down the left side of the parameter block represent byte offsets from the base address of the block.
- **Name:** The name of each parameter appears at the parameter's location within the block.
- **No.:** Each parameter in the block has a number, identifying its position within the block. The total number of parameters in the block is called the parameter count (`pCount`); `pCount` is the initial (zeroth) parameter in each call. The `pCount` parameter is needed because in some calls parameter count is not fixed; see minimum parameter count, below.
- **Size and type:** Each parameter is also identified by size (word, longword, or double longword) and type (input or result, and value or pointer). A word is 2 bytes; a longword is 4 bytes; a double longword is 8 bytes. An input is a parameter passed from the caller to GS/OS; a result is a parameter returned to the caller from GS/OS. A value is numeric or character data to be used directly; a pointer is the address of a buffer containing data (whether input or result) to be used.
- **Minimum parameter count:** To the right of each diagram, across from the `pCount` parameter, the minimum permitted value for `pCount` appears in parentheses. The maximum permitted value for `pCount` is the total number of parameters shown in the parameter block diagram.

Each parameter is described in detail after the diagram.

$2034 AddNotifyProc

Description

This call adds a notification procedure to the notification queue. After the call succeeds, whenever the specified event occurs, GS/OS will call the notification procedure. For the details of the notification procedure, see the section “Working With the Notification Queue” in Chapter 6. To delete a procedure, see the DelNotifyProc call later in this chapter.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	procPointer	1	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`procPointer` Longword input pointer: Pointer to the notification procedure to add to the notification queue.

Errors

(none except general GS/OS errors)

$201D BeginSession

Description

This call tells GS/OS to begin deferring block writes to disk. Normally GS/OS writes blocks to disk immediately whenever part of the system issues a block write request. However, when a write-deferral session is in progress, GS/OS caches blocks that are to be written until it receives an EndSession call.

This technique speeds up multiple file copying operations because it avoids physically writing file-system overhead blocks (such as directory blocks) to disk for every file. To do a fast multiple file copy, the application should execute a BeginSession call, copy the files, then execute an EndSession call.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 0)

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 0; maximum = 0.

Errors

(none except general GS/OS errors)

$2031 BindInt

Description

This function places the address of an interrupt handler into GS/OS's interrupt vector table.

For a complete description of GS/OS's interrupt handling subsystem, see the *GS/OS Device Driver Reference*. See also the UnbindInt call in this chapter.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 3)
$02	`intNum`	1	Word result value
$04	`vrn`	2	Word input value
$06	`intCode`	3	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 3; maximum = 3.

`intNum` Word result value: An identifying number assigned by GS/OS to the binding between the interrupt source and the interrupt handler. Its only use is as an input to the GS/OS call UnbindInt.

`vrn` Word input value: Vector reference number of the firmware vector for the interrupt source to be bound to the interrupt handler specified by `intCode`.

`intCode` Longword input pointer: Points to the first instruction of the interrupt handler routine.

Errors

$25	`irqTableFull`	interrupt vector table full

$2004 ChangePath

Description

This call changes a file's pathname to another pathname on the same volume, or changes the name of a volume. ChangePath cannot be used to change a device name; use the DRename call for that purpose.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 2)
$02	`pathname`	1	Longword input pointer
$06	`newPathname`	2	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 2.

`pathname` Longword input pointer: Points to a GS/OS string representing the name of the file whose pathname is to be changed.

`newPathname` Longword input pointer: Points to a GS/OS string representing the new pathname of the file whose name is to be changed.

Comments

A file may not be renamed while it is open.

A file may not be renamed if rename access is disabled for the file.

A subdirectory *s* may not be moved into another subdirectory *t* if $s = t$ or if *t* is contained in the directory hierarchy starting at *s*. For example, "rename `/v` to `/v/w`" is illegal, as is "rename `/v/w` to `/v/w/x`".

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$47	`dupPathname`	duplicate pathname
$4A	`badFileFormat`	version error
$4B	`badStoreType`	unsupported storage type
$4E	`invalidAccess`	file not destroy-enabled
$50	`fileBusy`	file open
$52	`unknownVol`	unsupported volume type
$57	`dupVolume`	duplicate volume
$58	`notBlockDev`	not a block device
$5A	`damagedBitMap`	block number out of range

$200B ClearBackupBit

Description

This call sets a file's state information to indicate that the file has been backed up and not altered since the backup. Whenever a file is altered, GS/OS sets the file's state information to indicate that the file has been altered.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	pathname	1	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`pathname` Longword input pointer: Points to a GS/OS string that gives the pathname of the file or directory whose backup status is to be cleared.

Errors

$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2B	`drvrWrtProt`	write-protected disk
$2E	`drvrDiskSwitch`	disk switched
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4A	`badFileFormat`	version error
$52	`unknownVol`	unsupported volume type
$58	`notBlockDev`	not a block device

$2014 Close

Description

This call closes the access path to the specified file, releasing all resources used by the file and terminating further access to it. Any file-related information that has not been written to the disk is written, and memory-resident data structures associated with the file are released.

If the specified value of the refNum parameter is $0000, all files at or above the current system file level are closed. This not only closes the resource fork of your application, but also closes the resource forks of any open desk accessory.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	refNum	1	Word input value

pCount Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

refNum Word input value: Identifying number assigned to the file by the Open call. A value of $0000 indicates that all files at or above the current system file level are to be closed.

Errors

$27	drvrIOError	I/O error
$2B	drvrWrtProt	write-protected disk
$2E	drvrDiskSwitch	disk switched
$43	invalidRefNum	invalid reference number
$48	volumeFull	volume full
$5A	damagedBitMap	block number out of range

$2001 Create

Description

This call creates either a standard file, an extended file, or a subdirectory on a volume mounted in a block device. A standard file contains a single sequence of bytes; an extended file contains a data fork and a resource fork, each of which is an independent sequence of bytes; a subdirectory is a data structure that contains information about other files and subdirectories. This call sets up file system state information for the new file and initializes the file to the empty state.

This call cannot be used to create a volume directory; the Format call performs that function. Similarly, it cannot be used to create a character-device file; the Character FST creates that special kind of file (see Chapter 14).

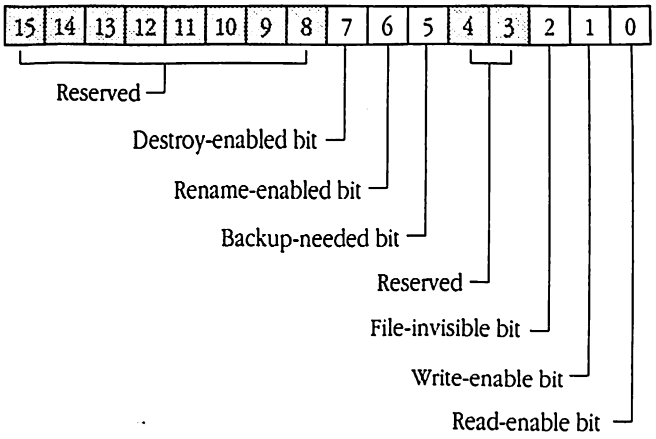
Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`pathname`	1	Longword input pointer
$06	`access`	2	Word input value
$08	`fileType`	3	Word input value
$0A	`auxType`	4	Longword input value
$0E	`storageType`	5	Word input value
$10	`eof`	6	Longword input value
$14	`resourceEOF`	7	Longword input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 7.

`pathname` Longword input pointer: Points to a GS/OS string representing the pathname of the file to be created. This is the only required parameter.

`access` Word input value: Specifies how the file may be accessed after it is created and whether or not the file has changed since the last backup, as shown in the following bit flag:



The most common setting for the access word is $00C3.

Software that supports file hiding (invisibility) should use bit 2 of the flag to determine whether or not to display a file or subdirectory.

`fileType` Word input value: Categorizes the file's contents. The value of this parameter has no effect on GS/OS's handling of the file, except that only certain file types may be executed directly by GS/OS. The file type values are assigned by Apple Computer.

`auxType` Longword input value: Categorizes additional information about the file. The value of this parameter has no effect on GS/OS's handling of the file. By convention, the interpretation of values in this parameter depends on the value in the `fileType` parameter. The auxiliary type values are assigned by Apple Computer.

`storageType` Word input value: Determines whether the file being created is a standard file, an extended file, or a subdirectory file. In addition, if bit 15 (msb) is set to 1, a resource fork is added to the file if the file already exists. The following values are valid:

$0000–$0003*	create a standard file
$0005	create an extended file
$8005	convert an existing standard file to contain a resource fork
$000D	create a subdirectory file

*If this parameter is set to $0000, $0002, or $0003, GS/OS interprets it as $0001 and actually changes it to $0001 on output.

`eof` Longword input value: Specifies an amount of storage to be preallocated during the Create call for the file that is being created. The type of entity is specified by the `storageType` parameter.

For a standard file, the `eof` parameter specifies the file size, in bytes, for which space is to be preallocated. GS/OS preallocates enough space to hold a standard file of the given size.

For an extended file, the `eof` parameter specifies the size, in bytes, of the data fork. GS/OS preallocates enough space to hold a data fork of the specified size.

For a subdirectory, the `eof` parameter specifies the number of entries the caller intends to place in the subdirectory. GS/OS preallocates enough space for the subdirectory to hold the specified number of entries.

`resourceEOF` Longword input value: Specifies the amount of space to preallocate for the resource fork for an extended file. GS/OS preallocates enough space to hold a resource fork of the specified size. This parameter is meaningful only if the `storageType` parameter value is $0005, indicating that an extended file is to be created.

Comments The Create call applies only to files on block devices.

All FSTs implement standard files, but they are not required to implement extended files.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$2F	`drvrOffLine`	specified volume not on line
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$47	`dupPathname`	duplicate pathname
$48	`volumeFull`	volume full
$49	`volDirFull`	volume directory full
$4B	`badStoreType`	unsupported (or incorrect) storage type
$52	`unknownVol`	unsupported volume type
$58	`notBlockDev`	not a block device
$5A	`damagedBitMap`	block number out of range
$70	`resExistsErr`	cannot expand file, resource fork already exists
$71	`resAddErr`	cannot add resource fork to this type of file

$202E DControl

Description

This call sends control information to a specified device. Dcontrol is really ten or more subcalls in one. Depending on the value of the control code parameter (`code`), DControl can set several classes of control information.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 5)
$02	`devNum`	1	Word input value
$04	`code`	2	Word input value
$06	`list`	3	Longword input pointer
$0A	`requestCount`	4	Longword input value
$0E	`transferCount`	5	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 5; maximum = 5.

`devNum` Word input value: Device number of the device to which the control information is being sent.

`code` Word input value: Number indicating the type of control request being made. The standard control codes are as follows:

$0000	`ResetDevice`
$0001	`FormatDevice`
$0002	`Eject`
$0003	`SetConfigParameters`
$0004	`SetWaitStatus`

$0005	`SetFormatOptions`
$0006	`AssignPartitionOwner`
$0007	`ArmSignal`
$0008	`DisarmSignal`
$0009	`SetPartitionMap`
$000A–$7FFF	(reserved)
$8000–$FFFF	(device-specific subcalls)

$0000–$7FFF are standard DControl subcalls that must be supported by the device driver; device-specific control calls may be supported by a particular device. Each subcall is described later in this section.

`list` Longword input pointer: Points to a buffer containing the device control information. The format of the data returned in the control buffer depends on the control subcall; see the individual subcall descriptions later in this section.

`requestCount` Longword input value: For control codes that have a control list, this parameter gives the size of the control list. For control subcalls that do not use the control list, this parameter is not used.

`transferCount` Longword result value: For control codes that have a control list, this parameter indicates the number of bytes of information actually transferred to the device. For control subcalls that do not use the control list, this parameter is not used.

Errors

$11	`invalidDevNum`	invalid device number
$22	`drvrBadParm`	bad call parameter

Control-list buffer

On a control call, the caller supplies a pointer (`list`) to a buffer, whose size must be at least `requestCount` bytes. In some cases, the first 2 bytes of the buffer are a length word, specifying the number of bytes of data in the buffer. In those cases, `requestCount` (which describes the amount of data supplied to the driver in the buffer) must be at least 2 bytes greater than the amount of data the driver needs, to account for the length word. The value returned in `transferCount` is the number of bytes used by the driver. If not enough data is supplied for the requested function, this call may return error $22 (`drvrBadParm`).

For those subcalls that pass no information in the control list, the driver does not access the control list and verify that its length word is zero; the driver ignores the control list entirely.

Subcalls

DControl is several control subcalls rather than a single call. Each value for the parameter `controlCode` corresponds to a particular subcall. Control codes of $0000 through $7FFF are standard control subcalls that are supported (if not actually acted upon) by every device driver. Device-specific control subcalls, which may be defined for individual devices, use control codes $8000 through $FFFF.

ResetDevice (DControl subcall)

Control code = $0000.

The Reset Device subcall sets a device's configuration parameters back to their default values. Many GS/OS device drivers contain default configuration settings for each device it controls; see the *GS/OS Device Driver Reference* for more information.

ResetDevice also sets a device's format options back to their default values, if the device supports media variables. See the SetFormatOptions subcall described later in this section.

If successful, the transfer count for this call is zero. The request count is ignored, and the control list is not used. However, for future compatibility, the `requestCount` parameter should be set to $0.

FormatDevice (DControl subcall)

Control code = $0001.

The FormatDevice subcall is used to format the medium, usually a disk drive, used by a block device. This call is not linked to any particular file system, in that no directory information is written to disk. FormatDevice simply prepares all blocks on the media for reading and writing.

After formatting, FormatDevice resets the device's format options back to their default values, if the device supports media variables. See the DControl subcall SetFormatOptions described later in this section.

Character devices do not implement this function but return with no error.

If successful, the transfer count for this call is zero. Request count is ignored; the control list is not used.

EjectMedium (DControl subcall)

Control code = $0002.

The EjectMedium subcall physically or logically ejects the recording medium, usually a disk, from a block device. In the case of linked devices (separate partitions on a single physical disk), physical ejection occurs only if, as a result of this call, all the linked devices become off line. If any devices linked to the device being ejected are still on line, the device being ejected is marked as off line but is not actually ejected.

Character devices do not implement this function but return with no error.

If successful, the transfer count for this call is zero. The `requestCount` parameter is ignored; the control list is not used.

SetConfigParameters (DControl subcall)

Control code = $0003.

The Set ConfigParameters subcall is used to send device-specific configuration parameters to a device. The configuration parameters are contained in the control list. The first word in the control list (`length`) indicates the length of the configuration list, in bytes. The configuration parameters follow the length word. Here is what the control list looks like:

Offset		Size	Description
$00	`length`	Word	The length of the list (in bytes)
$02	`configParamList`	Longword	The configuration list

The structure of the configuration list is device-dependent. See the *GS/OS Device Driver Reference* for more information.

This subcall is most typically used in conjunction with the status subcall GetConfigParameters. The application first uses the status subcall to get the list of configuration parameters for the device; it then modifies parameters as needed and makes this control subcall to send the new parameters to the device driver.

The request count for this subcall must be equal to `lengthWord` + 2. Furthermore, the length word of the new configuration list must equal the length word of the existing configuration list (the list returned from GetConfigParameters). If this call is made with an improper configuration list length, the call returns error $22 (`drvrBadParm`).

SetWaitStatus (DControl subcall)

Control code = $0004.

The SetWaitStatus subcall is used to set a character device to wait mode or no-wait mode.

◆ *Note:* Block devices cannot be set to no-wait mode. For block devices, the driver should return error $53 (`paramRangeErr`) on a no-wait mode request.

When a device is in wait mode, it does not terminate a Read call until it has read the number of characters specified in the request count, or if a newline character is encountered during the read and newline mode is enabled. In no-wait mode, a read call returns immediately after reading the available characters, with a transfer count indicating the number of characters returned. If one or more characters was available, the transfer count has a nonzero value; if no character was available, the transfer count is zero.

The control list for this subcall contains $0000 (to set wait mode) or $8000 (to set no-wait mode). The request count must be $0000 0002. The control list looks like this:

Offset		Size	Description
$00	`waitMode`	Word	The wait/no-wait status of the device

This subcall has no meaning for block devices; they operate in wait mode only. SetWaitStatus should return from block devices with no error (if wait mode is requested) or with error $ 22 (`drvrBadParm`) if no-wait mode is requested.

SetFormatOptions (DControl subcall)

Control code = $0005.

Some block devices can be formatted in more than one way. Formatting parameters can include such variables as file system group, number of blocks, block size, and interleave. Each driver that supports media variables (multiple formatting options) contains a list of the formatting options for its devices.

The SetFormatOptions subcall is used to set these media-specific formatting parameters prior to executing a FormatDevice subcall. SetFormatOptions does not itself cause or require a formatting operation. The control list for SetFormatOptions is as follows:

Offset		Size	Description
$00	`formatOption`	Word	The number of the format option
$0C	`interleave`	Word	The override interleave factor (if nonzero)

The format option number (`formatOptionNum`) specifies a particular format option entry from the driver's list of formatting options (returned from the DStatus subcall GetFormatOptions), in the following format:

Offset		Size	Description
$00	`formatOptionNum`	Word	The number of this option
$02	`linkRefNum`	Word	Number of linked option
$04	`flags`	Word	Flags word; see the following definition
$06	`blockCount`	Longword	Number of blocks supported by device
$0A	`blockSize`	Word	Flags word; see the following definition
$0C	`interleave`	Word	Interleave factor (in ratio to 1)
$0E	`mediaSize`	Word	Media size; see description of flags

See the description of the DStatus subcall GetFormatOptions, later in this chapter, for a more detailed description of the format option entry.

The `interleave` parameter in the control list, if nonzero, overrides `interleave` in the format option list. If the control list interleave factor is zero, the interleave specified in the format option list is used.

To carry out a formatting process with this subcall, do this:

1. Issue a (DStatus) GetFormatOptions subcall to the device. The call returns a list of all the device's format option entries and their corresponding values of `formatOptionNum`.
2. Issue a (DControl) SetFormatOptions subcall, specifying the desired format option.
3. Issue a (DControl) FormatDevice subcall.

△ **Important** SetFormatOptions sets the parameters for *one* subsequent formatting operation only. You must call SetFormatOptions each time you format a disk with anything other than the recommended (default) option. △

The SetFormatOptions subcall applies to block devices only; character devices return error $20 (`drvrBadReq`) if they receive this call.

AssignPartitionOwner (DControl subcall)

Control code = $0006.

The AssignPartitionOwner subcall provides support for partitioned media on block devices. Each partition on a disk has an owner, identified by a string stored on disk. The owner name is used to identify the file system to which the partition belongs.

This subcall is executed by an FST when an application makes the call EraseDisk, to allow the driver to reassign the partition to the new owner.

Partition owner names are assigned by Apple Developer Technical Support, and can be up to 32 bytes in length—uppercase and lowercase characters are considered equivalent.

The control list for this call consists of a GS/OS string naming the partition owner:

Offset		Size	Description
$00	`length`	Word	The length of the name (in bytes)
$02	`ownerName`		The partition owner name

Block devices with non-partitioned media and character devices do nothing with this call and return no error.

ArmSignal (DControl subcall)

Control code = $0007.

The ArmSignal subcall provides a means for an application to bind its own software interrupt handler to the hardware interrupt handler controlled by the device. This is the control list for the subcall:

Offset		Size	Description
$00	`signalCode`	Word	An ID for this handler and its signals
$02	`priority`	Word	The priority for this handler's signals
$04	`handlerAddress`	Longword	A pointer to the signal handler's entry

The `signalCode` parameter is an arbitrary number assigned by the caller to match the signals that the signal source generates with the proper handler; its only subsequent use is as an input to the DControl subcall DisarmSignal. The `priority` parameter is the signal priority the caller wishes to assign, with $0000 being the lowest priority and $FFFF being the highest priority. The `handlerAddress` parameter is the entry address of the signal handler for that signal code.

DisarmSignal (DControl subcall)

Control code = $0008.

The Disarm Signal subcall provides a means for an application to unbind its own software interrupt handler from the hardware interrupt handler controlled by the device. The `signalCode` parameter is the identification number assigned to that handler when the signal was armed.

Offset		Size	Description
$00	`signalCode`	Word	The signal handler's ID

SetPartitionMap (DControl subcall)

Status code = $0009.

This call passes to a device, in the control list, the partion map for a partitioned disk or other medium. The structure of the partition information is device-dependent.

Device-specific DControl subcalls

Device-specific DControl subcalls are provided to allow device-driver writers to implement control calls specific to individual device drivers' needs. DControl subcalls with `code` values of $8000 to $FFFF are passed by the Device Manager directly to the device dispatcher for interpretation by the device driver.

The content and format of information passed by this subcall can be defined individually for each type of device. The only requirements are that the parameter block must be the regular DControl parameter block, and the control code must be in the range $8000–$FFFF.

$2035 DelNotifyProc

Description

This call removes a notification procedure from the notification queue. After this call succeeds, GS/OS no longer calls the specified notification procedure. For the details of the notification procedure, see the section “Working With the Notification Queue” in Chapter 6. To add a procedure, see the AddNotifyProc call earlier in this chapter.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	procPointer	1	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`procPointer` Longword input pointer: Pointer to the notification procedure to delete from the notification queue.

Errors

(none except general GS/OS errors)

$2002 Destroy

Description

This call deletes a specified standard file, extended file (both the data fork and resource fork), or subdirectory, and updates the state of the file system to reflect the deletion. After a file is destroyed, no other operations on the file are possible.

This call cannot be used to delete a volume directory; the Format call reinitializes volume directories.

It is not possible to delete only the data fork or only the resource fork of an extended file.

Before deleting a subdirectory file, you must empty it by deleting all the files it contains.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	pathname	1	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`pathname` Longword input pointer: Points to a GS/OS string representing the pathname of the file to be deleted.

Comments

A file cannot be destroyed if it is currently open or if the access attributes do not permit destroy access.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4B	`badStoreType`	unsupported storage type
$4E	`invalidAccess`	file not destroy-enabled
$50	`fileBusy`	file open
$52	`unknownVol`	unsupported volume type
$58	`notBlockDev`	not a block device
$5A	`damagedBitMap`	block number out of range

$202C DInfo

Description This call returns general information about a device attached to the system.

Parameters

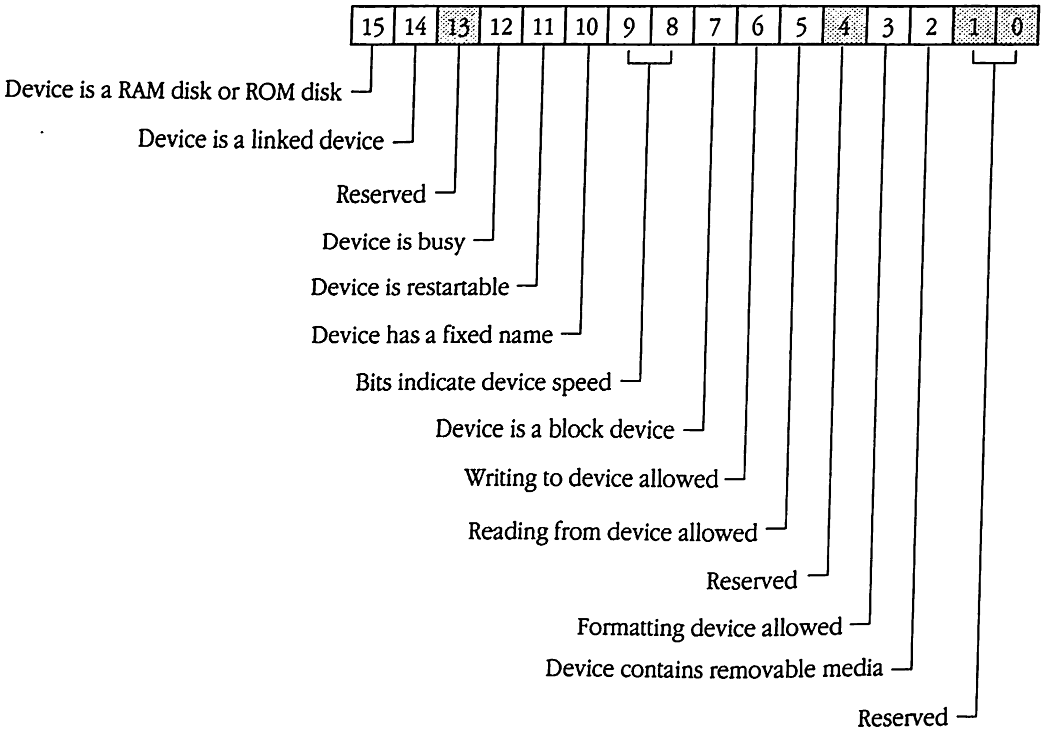
Offset		No.	Size and type
$00	pCount		Word input value (minimum = 2)
$02	devNum	1	Word input value
$04	devName	2	Longword input pointer
$08	characteristics	3	Word result value
$0A	totalBlocks	4	Longword result value
$0E	slotNum	5	Word result value
$10	unitNum	6	Word result value
$12	version	7	Word result value
$14	deviceID	8	Word result value
$16	headLink	9	Word result value
$18	forwardLink	10	Word result value
$1A	extendedDIBptr	11	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 11.

`devNum` Word input value: Device number. GS/OS assigns device numbers in sequence 1, 2, 3, and so on as it loads or creates the device drivers. There is no fixed correspondence between devices and device numbers. To get information about every device in the system, make repeated calls to DInfo with devNum values of 1, 2, 3, and so on until GS/OS returns error $11 (invalid device number).

devName Longword input pointer: Points to a result buffer in which GS/OS returns the device name of the device specified by device number. The maximum size of the string is 31 bytes, so the maximum size of the returned value is 33 bytes. Thus the buffer size should be 35 bytes.

characteristics Word result value: Individual bits in this word give the general characteristics of the device, as shown in the following bit flag:

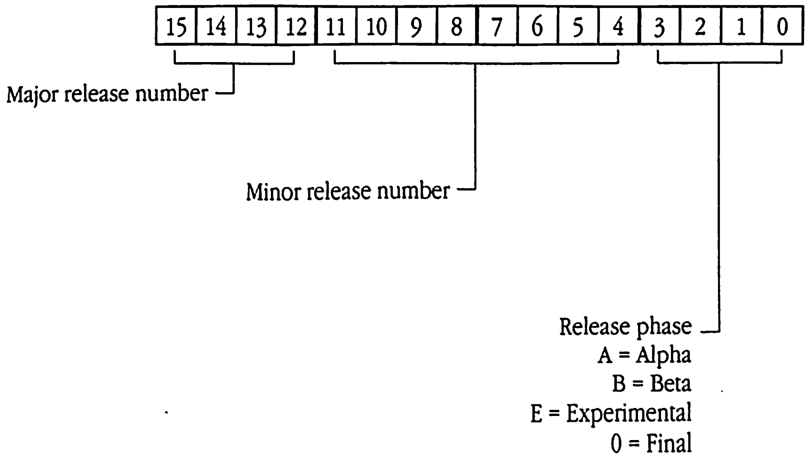

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Device is a RAM disk or ROM disk
Device is a linked device
Reserved
Device is busy
Device is restartable
Device has a fixed name
Bits indicate device speed
Device is a block device
Writing to device allowed
Reading from device allowed
Reserved
Formatting device allowed
Device contains removable media
Reserved

totalBlocks Longword result value: If the device is a block device, this parameter gives the maximum number of blocks on volumes handled by the device. For character devices, this parameter returns zero.

`slotNum` Word result value: Slot number corresponding to the resident firmware associated with the device or slot number of the slot containing the device. Valid values are $0000–000F.

`unitNum` Word result value: Unit number of the device within the given slot. This parameter has no correlation with device number.

`version` Word result value: Version number of the device driver. This parameter has the same format as the SmartPort version, as shown in the following bit flag:



For example, a version of 2.00 in this format would be entered as $2000; a version of 0.18 Beta would be entered as $018B.

`deviceID` Word result value: Identifying number associated with a particular type of device. This parameter may be useful for Finder-like applications when determining what type of icon to display for a particular device.

Current definitions of device ID numbers include

$0000	Apple 5.25 Drive (includes UniDisk™, DuoDisk®, Disk IIc, and Disk II®)
$0001	ProFile™ 5 MB
$0002	ProFile10 MB
$0003	Apple 3.5 Drive (includes UniDisk 3.5 Drive)
$0004	SCSI (generic)
$0005	SCSI hard disk
$0006	SCSI tape drive
$0007	SCSI CD-ROM
$0008	SCSI printer
$0009	Serial modem
$000A	Console driver
$000B	Serial printer
$000C	Serial LaserWriter®
$000D	AppleTalk® LaserWriter
$000E	RAM disk
$000F	ROM disk
$0010	File Server
$0011	Reserved
$0012	Apple Desktop Bus™
$0013	Hard disk (generic)
$0014	Floppy disk (generic)
$0015	Tape drive (generic)
$0016	Character device driver (generic)
$0017	MFM-encoded disk drive
$0018	AppleTalk network (generic)
$0019	Sequential access device
$001A	SCSI scanner
$001B	Other scanner
$001C	LaserWriter SC
$001D	AppleTalk main driver
$001E	AppleTalk file service driver
$001F	AppleTalk RPM driver

`headLink` Word result value: Device number that describes a link to another device. It is the device number of the first device in a linked list of devices that a represent distinct partitions on a single medium. A value of 0 indicates that no link exists.

`forwardLink` Word result value: Device number that describes a link to another device, that is, it is the device number of the next device in a linked list of devices that represent distinct partitions on a single medium. A value of 0 indicates that no link exists.

`extendedDIBptr` Longword input pointer: Points to a buffer in which GS/OS returns information about the extended device information block.

Errors

$11	`invalidDevNum`	invalid device number

$202F DRead

Description

This call performs a device-level read on a specified device.

This description only provides general information about the parameter block; for more information, see the *GS/OS Device Driver Reference.*

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 6)
$02	devNum	1	Word input value
$04	buffer	2	Longword input pointer
$08	requestCount	3	Longword input value
$0C	startingBlock	4	Longword input value
$10	blockSize	5	Word input value
$12	transferCount	6	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 6; maximum = 6.

`devNum` Word input value: Device number of the device from which data is to be read.

`buffer` Longword input pointer: Points to a buffer into which the data is to be read. The buffer must be big enough to hold the data.

`requestCount` Longword input value: Specifies the number of bytes to be read.

startingBlock Longword input value: For a block device, this parameter specifies the logical block number of the block where the read starts. For a character device, this parameter is unused.

blockSize Word input value: Size, in bytes, of a block on the specified block device. For character devices, the parameter must be set to zero.

transferCount Longword result value: Number of bytes actually transferred by the call.

Errors			
	$11	invalidDevNum	invalid device number

$2036 DRename

Description

This call replaces a device name as specified in a device information block.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 2)
$02	devNum	1	Word input value
$04	strPtr	2	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 2.

`devNum` Word input value: Device number of the device to be renamed.

`strPtr` Longword input pointer: Points to a GS/OS string with a maximum length of 31 ASCII characters. The string must be uppercase with the most significant bit off.

Errors

$11	`invalidDevNum`	invalid device number
$53	`paramRangeErr`	string length greater than 31 characters or less than 1 character
$67	`devNameErr`	device exists with same name as replacement name

$202D DStatus

Description

Returns status information about a specified device.

This description provides only general information about the call; for more information, see the *GS/OS Device Driver Reference.*

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 5)
$02	`devNum`	1	Word input value
$04	`code`	2	Word input value
$06	`list`	3	Longword input pointer
$0A	`requestCount`	4	Longword input value
$0E	`transferCount`	5	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 5; maximum = 5.

`devNum` Word input value: Device number of the device whose status is to be returned.

`code` Word input value: Number indicating the type of status request being made. The status requests are described completely in the *GS/OS Device Driver Reference.* Status codes of $0000-$7FFF are standard status calls that must be supported by the device driver. Device-specific status calls may be supported by a particular device; they use status codes $8000–$FFFF. These are the standard status codes:

$0000	Device status
$0001	Return configuration parameters
$0002	Return wait/no-wait status

$0003	Get format options
$0004	Return partition map
$0005–$7FFF	Reserved for Apple Computer, Inc.
$8000–$FFFF	Device-specific

The individual subcalls are described later in this section.

list Longword input pointer: Points to a buffer in which the device returns its status information.

requestCount Longword input value: Specifies the number of bytes to be returned in the status list. The call will never return more than this number of bytes.

transferCount Longword result value: Specifies the number of bytes actually returned in the status list. This value will always be less than or equal to the request count.

Errors $11 invalidDevNum invalid device number

Buffer size On a status call, the caller supplies a pointer (list) to a buffer, whose size must be at least requestCount bytes. In some cases, the first 2 bytes of the buffer are a length word, specifying the number of bytes of data in the buffer. In those cases, requestCount must be at least 2 bytes greater than the maximum amount of data than the call can return, to account for the length word.

If requestCount is not big enough for the requested data, the driver either fills the buffer with as much data as can fit and returns with no error, or does not fill the buffer and returns error $22 (drvrBadParm). See the individual DStatus subcall descriptions for details.

DStatus subcalls DStatus is several status subcalls rather than a single call. Each value for the parameter statusCode corresponds to a particular subcall. Status codes of $0000 through $7FFF are standard status subcalls that are supported (if not actually acted upon) by every device driver. Device-specific status subcalls, which may be defined for individual devices, use status codes $8000 through $FFFF. Each of the status subcalls is described individually in the following sections.

GetDeviceStatus (DStatus subcall)

Status code = $0000.

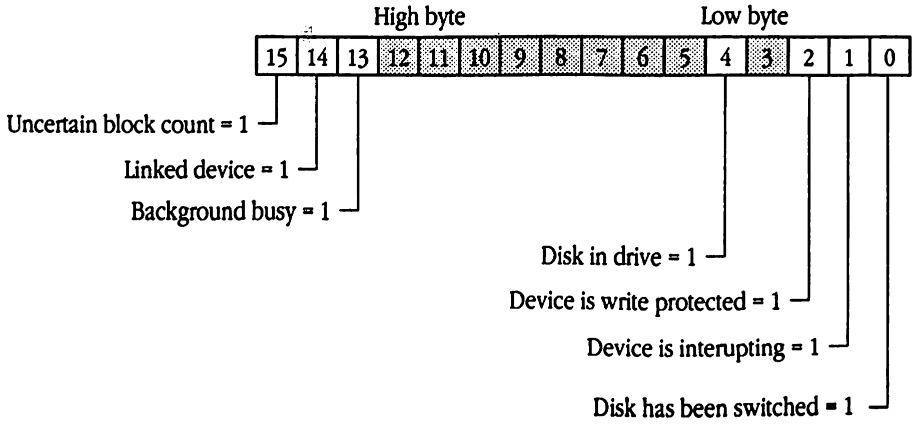
The Device Status subcall returns, in the status list, a general device status word followed by a number giving the total number of blocks on the device.

This subcall normally requires an input `requestCount` of $0000 0006, the size in bytes of the status list in this case. However, if only the status word is desired, use a request count of $0000 0002. This is the format of the status list:

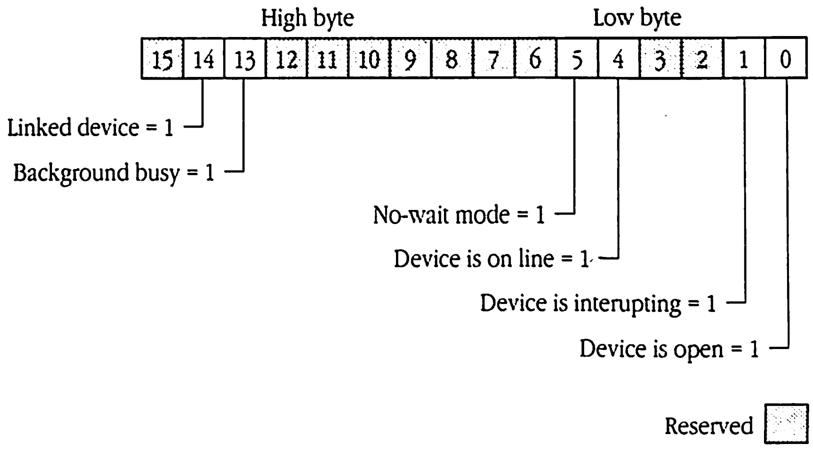
Offset		Size	Description
$00	`statusWord`	Word	Status word (see following definition)
$02	`numBlocks`	Longword	The number of blocks on the device

The device status word has two slightly different formats, depending on whether the device is a block device or a character device. This is its definition:

Block device:


High byte
Low byte
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Uncertain block count = 1
Linked device = 1
Background busy = 1
Disk in drive = 1
Device is write protected = 1
Device is interupting = 1
Disk has been switched = 1

Character device:



To maintain future compatibility, the driver must return zero in all reserved bit positions for the status word, because reserved bits may in the future be assigned new values.

GetConfigParameters (DStatus subcall)

Status code = $0001.

The GetConfigParameters subcall returns, in the status list, a length word and a list of configuration parameters. The structure of the configuration list is device-dependent.

The request count for this subcall (the length of the configuration list plus the length word) must be in the range $0000 0002 to $0000 FFFF. This is the format of the status list:

Offset		Size	Description
$00	length	Word	The length of the list (in bytes)
$02	configParamList	Longword	The configuration list

GetWaitStatus (DStatus subcall)

Status code = $0002.

The GetWaitStatus subcall is used to determine if a device is in wait mode or no-wait mode. When a device is in wait mode, it does not terminate a Read call until it has read the number of characters specified in the request count, or if a newline character is encountered during the read and newline mode is enabled. In no-wait mode, a Read call returns immediately after reading the available characters, with a transfer count indicating the number of characters returned. If one or more characters was available, the transfer count has a nonzero value; if no character was available, the transfer count is zero.

The status list for this subcall contains $0000 if the device is operating in wait mode, $8000 if it is operating in no-wait mode. The request count must be $0000 0002. This is the status list format:

Offset		**Size**	**Description**
$00	`waitMode`	Word	The wait/no-wait status of the device

◆ *Note:* Block devices always operate in wait mode. Whenever this call is made to a block device, the call returns $0000 in the status list.

GetFormatOptions (DStatus subcall)

Status code = $0003.

Some block devices can be formatted in more than one way. Formatting parameters can include such variables as file system group, number of blocks, block size, and interleave. Each driver that supports media variables (multiple formatting options) contains a list of the formatting options for its devices. The options can be used for two purposes:

- An application can select one with a SetFormatOptions subcall, prior to formatting a block device. See the description of the DControl call in this chapter.
- An FST can display one or more of the options to the user when initializing disks. See the section "Disk Initialization and FSTs," in Chapter 11.

This subcall returns the list of formatting options for a particular device. Devices that do not support media variables return a transfer count of zero and generate no error. Character devices do nothing and return no error from this call. If a device does support media variables, it returns a status list consisting of a 4-word header followed by a set of entries, each of which describes a formatting option. The status list looks like this:

Offset		Size	Description
$00	`numOptions`	Word	Number of format-option entries in list
$02	`numDisplayed`	Word	Number of options to be displayed
$04	`recommendOption`	Word	Recommended default formatting option
$06	`currentOption`	Word	Formatting option of current media
$08	`formatOption1`	16 bytes	The first format option entry
$XX	`formatOptionN`	16 bytes	The last format option entry

Of the total number of options in the list, zero or more can be displayed on the initialization dialog presented to the user when initializing a disk (see the calls Format and EraseDisk in this chapter). The options to be displayed are always the first ones in the list. (Undisplayed options are available so that drivers can provide FSTs with logically different options that are actually physically identical and therefore needn't be duplicated in the dialog.)

Each format-options entry consists of 16 bytes, containing these fields:

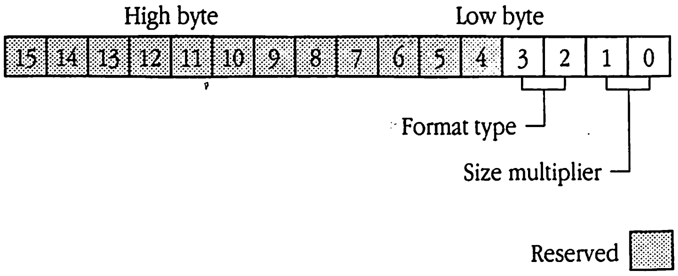
Offset		Size	Description
$00	`formatOptionNum`	Word	The number of this option
$02	`linkRefNum`	Word	Number of linked option
$04			

(Continued)

$04	`flags`	Word	Flags word; see the following definition
$06	`blockCount`	Longword	Number of blocks supported by device
$0A	`blockSize`	Word	Flags word; see the following definition
$0C	`interleave`	Word	Interleave factor (in ratio to 1)
$0E	`mediaSize`	Word	Media size; see description of flags

Linked options are options that are physically identical but which may appear different at the FST level. Linked options are in sets; one of the set is displayed, whereas all others are not, so that the user is not presented with several choices on the initialization dialog. See "Example," later in this section.

Bits within the flags word are defined as follows:



In the format options flag word, Format type defines the general file-system family for formatting. An FST might use this information to enable or disable certain options in the initialization dialog. Format type can have these binary values and meanings:

00	Universal Format	(for any file system)
01	Apple Format	(for an Apple file system)
10	NonApple Format	(for other file systems)
11	(not valid)	

Size multiplier is used, in conjunction with the parameter `mediaSize`, to calculate the total number of bytes of storage available on the device.

Size multiplier can have these binary values and meanings:

00	`mediaSize` is in bytes
01	`mediaSize` is in kilobytes (KB)
10	`mediaSize` is in megabytes (MB)
11	`mediaSize` is in gigabytes (GB)

Example

A list returned from this call for a device supporting two possible interleaves intended to support Apple's file systems (DOS 3.3, ProDOS, MFS or HFS) might be as follows. The field `transferCount` has the value $0000 0038 (56 bytes returned in list). Only two of the three options are displayed; option 2 (displayed) is linked to option 3 (not displayed), because both have exactly the same physical formatting. Both must exist, however, because the driver will provide an FST with either 512 bytes or 256 bytes per block, depending on the option chosen. At format time, each FST will choose its proper option among any set of linked options.

The entire format options list looks like this:

Value	**Explanation**
Format options list header:	
$0003	Three format options in the status list
$0002	Only two display entries
$0001	Recommended default is option 1
$0003	Current media is formatted as specified by option 3
Format Option 1:	
$0001	Option 1
$0000	LinkRef = none
$0005	Apple format/size in kilobytes
$0000 0640	Block count = 1600
$0200	Block size = 512 bytes
$0002	Interleave factor = 2:1
$0320	Media size = 800 KB
Format Option 2:	
$0002	Option 2
$0003	LinkRef = option 3
$0005	Apple format/size in kilobytes
$0000 0640	Block count = 1600
$0100	Block size = 256 bytes
$0004	Interleave factor = 4:1
$0190	Media size = 400 KB

Value	Explanation
Format Option 3:	
$0003	Option 3
$0000	LinkRef = none
$0005	Apple format/size in kilobytes
$0000 0320	Block count = 800
$0200	Block size = 512 bytes
$0004	Interleave factor = 4:1
$0190	Media size = 400 KB

GetPartitionMap (DStatus subcall)

Status code = $0004.

This call returns, in the status list, the partition map for a partitioned disk or other medium. The structure of the partition information is device-dependent.

Device-specific DStatus subcalls

Device-specific DStatus subcalls are provided to allow device-driver writers to implement Status calls specific to individual device drivers' needs. DStatus calls with `statusCode` values of $8000 to $FFFF are passed by the Device Manager directly to the device dispatcher for interpretation by the device driver.

The content and format of information returned from these subcalls can be defined individually for each type of device; the only requirements are that the parameter block must be the regular DStatus parameter block, and the status code must be in the range $8000–$FFFF.

$2030 DWrite

Description

This call performs a device-level write to a specified device.

This description only provides general information about the parameter block; for more information, see the *GS/OS Device Driver Reference*.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 6)
$02	`devNum`	1	Word input value
$04	`buffer`	2	Longword input pointer
$08	`requestCount`	3	Longword input value
$0C	`startingBlock`	4	Longword input value
$10	`blockSize`	5	Word input value
$12	`transferCount`	6	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 6; maximum = 6.

`devNum` Word input value: Device number of the device from which data is to be written.

`buffer` Longword input pointer: Points to a buffer from which the data is to be written.

`requestCount` Longword input value: Specifies the number of bytes to be written.

`startingBlock` Longword input value: For a block device, this parameter specifies the logical block number of the block where the write starts. For a character device, this parameter is unused.

`blockSize` Word input value: Size, in bytes, of a block on the specified block device. For character devices, the parameter is unused and must be set to zero.

`transferCount` Longword result value: Number of bytes actually transferred by the call.

Errors			
	$11	`invalidDevNum`	invalid device number

$201E EndSession

Description

This call tells GS/OS to flush any deferred block writes that occurred during a write-deferral session (started by a BeginSession call) and to resume normal write-through processing for all block writes.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 0)

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 0; maximum = 0.

Errors

(none except general GS/OS errors)

$2025 EraseDisk

Description

This call puts up a dialog box that allows the user to erase a specified volume and choose which file system is to be placed on the newly erased volume. The volume must have been previously physically formatted. The only difference between EraseDisk and Format is that EraseDisk does not physically format the volume. See the Format call later in this chapter.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 3)
$02	`devName`	1	Longword input pointer
$06	`volName`	2	Longword input pointer
$0A	`fileSysID`	3	Word result value
$0C	`reqFileSysID`	4	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 3; maximum = 4.

`devName` Longword input pointer: Points to a GS/OS string representing the device name of the device containing the volume to be erased.

`volName` Longword input pointer: Points to a GS/OS string representing the volume name to be assigned to the newly erased volume.

`fileSysID` Word result value: If the call is successful, this parameter identifies the file system with which the disk was formatted. If the call is unsuccessful, this parameter is undefined.

The file system IDs are as follows:

$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E–$000F	Reserved
$0007	Lisa		

`reqFileSysID` Word input value: Provides the file system ID of the file system that should be initialized on the disk. The values for this parameter are the same as those for the `fileSysID` parameter.

If you supply this parameter, it suppresses the dialog box from the Disk Initialization package that asks the user which file system to place on the newly erased disk. Normally, your application should not use this parameter; use it only if your application needs to format the disk for a specific FST.

Errors

If the carry flag is set but A is equal to 0, the user selected Cancel in the dialog box.

$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device request
$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2B	`drvrWrtProt`	write-protected disk
$40	`badPathSyntax`	invalid pathname syntax
$50	`fileBusy`	files open on the volume mounted in the target device
$58	`notBlockDev`	not a block device
$5D	`osUnsupported`	file system not available
$64	`invalidFSTID`	invalid FST ID

$200E ExpandPath

Description

This call converts the input pathname into the corresponding full pathname with colons (ASCII $3A) as separators. If the input is a full pathname, ExpandPath simply converts all of the separators to colons. If the input is a partial pathname, ExpandPath concatenates the specified prefix with the rest of the partial pathname and converts the separators to colons.

If a device name or number is used in the input path, GS/OS attempts to replace that portion of the output path with the name of the volume mounted in the selected device. If the volume name cannot be determined, this call leaves the device name and number portion of the input path unchanged in the output path.

If bit 15 (msb) of the `flags` parameter is set, the call converts all lowercase characters to uppercase (all other bits in this word must be cleared). This call also performs limited syntax checking. It returns an error if it encounters an illegal character, two adjacent separators, or any other syntax error.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 2)
$02	`inputPath`	1	Longword input pointer
$06	`outputPath`	2	Longword input pointer
$0A	`flags`	3	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 3.

`inputPath` Longword input pointer: Points to a GS/OS string that is to be expanded.

`outputPath` Longword input pointer: Points to a result buffer where the expanded pathname is returned.

`flags` Word input value: If bit 15 is set to 1, this call returns the expanded pathname all in uppercase characters. All other bits in this word must be zero.

Errors

$40	`badPathSyntax`	invalid pathname syntax
$4F	`buffTooSmall`	buffer too small

$2015 Flush

Description

This call writes to the volume all file state information that is buffered in memory but has not yet been written to the volume. The purpose of this call is to assure that the representation of the file on the volume is consistent and up to date with the latest GS/OS calls affecting the file.

Thus, if a power failure occurs immediately after the Flush call is completed, it should be possible to read all data written to the file as well as all file attributes. If such a power failure occurs, files that have not been flushed may be in inconsistent states, as may the volume as a whole. The price for this security is performance; the Flush call takes time to complete its work. Therefore, be careful how often you use the Flush call.

A value of $0000 for the `refNum` parameter indicates that all files at or above the current file level are to be flushed.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`refNum`	1	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`refNum` Word input value: Identifying number assigned to the file by the Open call. A value of $0000 indicates that all files at or above the current system file level are to be flushed.

Errors

$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$2E	`drvrDiskSwitch`	disk switched
$43	`invalidRefNum`	invalid reference number
$48	`volumeFull`	volume full
$5A	`damagedBitMap`	block number out of range

$2024 Format

Description

This call puts up a dialog box that allows the user to physically format a specified volume and choose which file system is to be placed on the newly formatted volume.

Some devices do not support physical formatting. In this case the Format call acts like the EraseDisk call and writes only the empty file system. See the EraseDisk call earlier in this chapter.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 3)
$02	devName	1	Longword input pointer
$06	volName	2	Longword input pointer
$0A	fileSysID	3	Word result value
$0C	reqFileSysID	4	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 3; maximum = 4.

`devName` Longword input pointer: Points to a GS/OS string representing the device name of the device containing the volume to be formatted.

`volName` Longword input pointer: Points to a GS/OS string representing the volume name to be assigned to the newly formatted blank volume.

`fileSysID` Word result value: If the call is successful, this parameter identifies the file system with which the disk was formatted. If the call is unsuccessful, this parameter is undefined.

The file system IDs are as follows:

$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E–$000F	Reserved
$0007	Lisa		

`reqFileSysID` Word input value: Provides the file system ID of the file system that should be initialized on the disk. The values for this parameter are the same as those for the `fileSysID` parameter.

If you supply this parameter, it suppresses the dialog box from the Disk Initialization package that asks the user how the disk should be formatted. Normally, your application should not use this parameter; use it only if your application needs to format the disk for a specific file system.

Errors

If the carry flag is set but A is equal to 0, the user selected Cancel in the dialog box.

$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device number request
$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2B	`drvrWrtProt`	write-protected disk
$40	`badPathSyntax`	invalid pathname syntax
$50	`fileBusy`	files open on the volume mounted in the target device
$58	`notBlockDev`	not a block device
$5D	`osUnsupported`	file system not available
$64	`invalidFSTID`	invalid FST ID

$2033 FSTSpecific

Description

FSTSpecific is a call that can be defined individually for any file system translator.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06		3	Subcall-specific parameter or parameters

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = number of subcall-specific parameters.

`fileSysID` Word input value: File system ID of the FST to which the call is directed.

`commandNum` Word input value: Number that specifies which particular subcall of FSTSpecific to execute.

(subcall-specific) Word or longword input or result value: Depends on the specific subcall. See the appropriate FST chapter for those subcalls.

Errors

(none except general GS/OS errors)

$2028 GetBootVol

Description Returns the volume name of the volume from which GS/OS was last loaded and executed. The volume name returned by this call is equivalent to the prefix specified by `*/`.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`dataBuffer`	1	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`dataBuffer` Longword input pointer: Points to a memory area where a GS/OS output string giving the boot volume name is to be returned.

Errors

$4F	`buffTooSmall`	buffer too small

$2020 GetDevNumber

Description

This call returns the device number of a device identified by device name or volume name. Only block devices may be identified by volume name, and then only if the named volume is mounted. Most other device calls refer to devices by device number.

GS/OS assigns device numbers at boot time. The numbers are a series of consecutive integers beginning with 1. There is no algorithm for determining the device number for a particular device.

Because a device may hold different volumes and because volumes may be moved from one device to another, the device number returned for a particular volume name may be different at different times.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 2)
$02	devName	1	Longword input pointer
$06	devNum	2	Word result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 2.

`devName` Longword input pointer: Points to a result buffer representing the device name or volume name (for a block device).

`devNum` Word result value: Device number of the specified device.

Errors

$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device request
$40	`badPathSyntax`	invalid pathname syntax
$45	`volNotFound`	volume not found

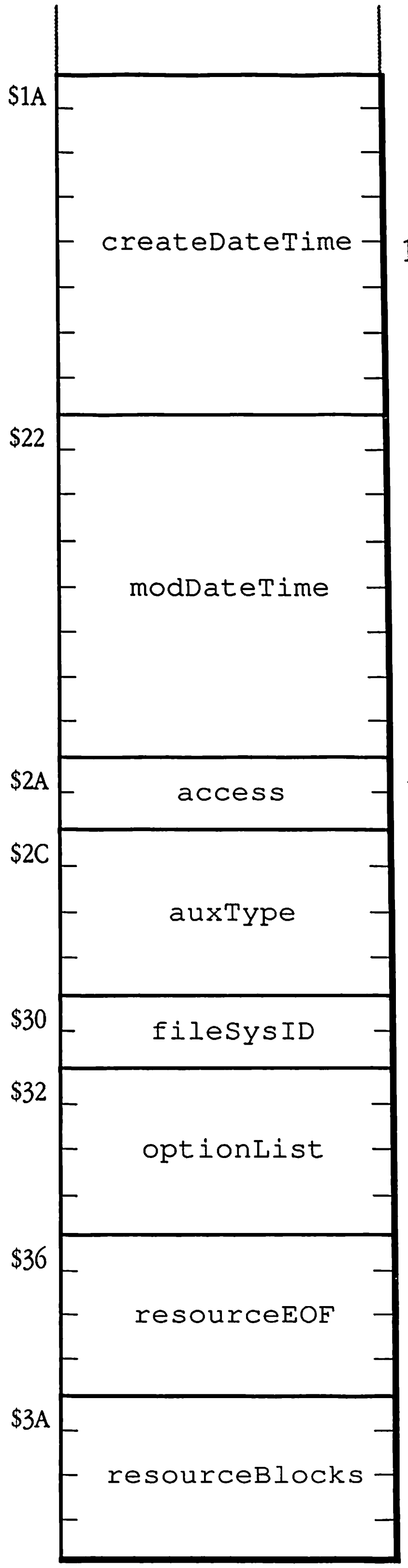
$201C GetDirEntry

Description

This call returns information about a directory entry in the volume directory or a subdirectory. Before executing this call, the application must open the directory or subdirectory. The call allows the application to step forward or backward through file entries or to specify absolute entries by entry number.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 5)
$02	refNum	1	Word input value
$04	flags	2	Word result value
$06	base	3	Word input value
$08	displacement	4	Word input value
$0A	name	5	Longword input pointer
$0E	entryNum	6	Word result value
$10	fileType	7	Word result value
$12	eof	8	Longword result value
$16	blockCount	9	Longword result value
$1A			

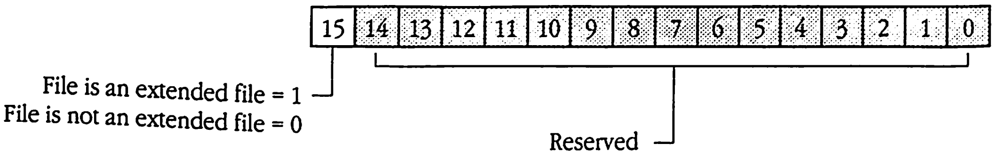


$1A	createDateTime	10	Double longword result value
$22	modDateTime	11	Double longword result value
$2A	access	12	Word result value
$2C	auxType	13	Longword result value
$30	fileSysID	14	Word result value
$32	optionList	15	Longword input pointer
$36	resourceEOF	16	Longword result value
$3A	resourceBlocks	17	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 5; maximum = 17.

`refNum` Word input value: Identifying number assigned to the directory or subdirectory by the Open call.

`flags` Word result value: Flags that indicate various attributes of the file, as follows:



15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
File is an extended file = 1
File is not an extended file = 0
Reserved

`base` Word input value: Determines how the displacement parameter should be interpreted, as follows:

$0000 displacement gives an absolute entry number

$0001 displacement is added to current displacement to get next entry number

$0002 displacement is subtracted from current displacement to get next entry number

`displacement` Word input value: In combination with the `base` parameter, the `displacement` parameter specifies the directory entry whose information is to be returned. When the directory is first opened, GS/OS sets the current displacement value to $0000. The current displacement value is updated on every GetDirEntry call.

If the `base` and `displacement` parameters are both zero, GS/OS returns a 2-byte value in the `entryNum` parameter that specifies the total number of active entries in the subdirectory. In this case, GS/OS also resets the current displacement to the first entry in the subdirectory.

To step through the directory entry by entry, you should set both the `base` and `displacement` parameters to $0001.

`name` Longword input pointer: Points to a result buffer giving the name of the file or subdirectory represented in this directory entry.

`entryNum` Word result value: Absolute entry number of the entry whose information is being returned. This parameter is provided so that a program can obtain the absolute entry number even if the `base` and `displacement` parameters specify a relative entry.

`fileType` Word result value: File type of the directory entry.

`eof` Longword result value: For a standard file, specifies the number of bytes that can be read from the file. For an extended file, this parameter gives the number of bytes that can be read from the file's data fork.

`blockCount` Longword result value: For a standard file, specifies the number of blocks used by the file. For an extended file, this parameter gives the number of blocks used by the file's data fork.

`createDateTime` Double longword result value: Creation date and time of the directory entry. The format of the date and time is shown in Table 4-2 in Chapter 4.

`modDateTime` Double longword result value: Modification date and time of the directory entry. The format of the date and time is shown in Table 4-2 in Chapter 4.

`access` Word result value: Access attribute of the directory entry.

`auxType` Longword result value: Auxiliary type of the directory entry.

`fileSysID` Word result value: File system identifier of the file system on the volume containing the file. Values of this parameter are described under the Volume call later in this chapter.

`optionList` Longword input pointer: Points to a data area where GS/OS returns FST-specific information related to the file. This is the same information returned in the option list of the Open and GetFileInfo calls.

This parameter points to a buffer that starts with a length word giving the total buffer size, including the length word. The next word is an output length value that is undefined on input. On output, this word is set to the size of the output data excluding the length word and the output length word. GS/OS will not overflow the available space specified in the input length word. If the data area is too small, the application can reissue the call after allocating a new output buffer with size adjusted to output length plus four.

`resourceEOF` Longword result value: If the specified file is an extended file, this parameter gives the number of bytes that can be read from the file's resource fork. Otherwise, the parameter is undefined.

`resourceBlocks` Longword result value: If the specified file is an extended file, this parameter gives the number of blocks used by the file's resource fork. Otherwise, the parameter is undefined.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$4A	`badFileFormat`	version error
$4B	`badStoreType`	unsupported storage type
$4F	`buffTooSmall`	buffer too small
$52	`unknownVol`	unsupported volume type
$58	`notBlockDev`	not a block device
$61	`endOfDir`	end of directory

$2019 GetEOF

Description

This function returns the current logical size of a specified file. See also the SetEOF call.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 2)
$02	`refNum`	1	Word input value
$04	`eof`	2	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 2.

`refNum` Word input value: Identifying number assigned to the file by the Open call.

`eof` Longword result value: Current logical size of the file, in bytes.

Errors

$43	`invalidRefNum`	invalid reference number

$2006 GetFileInfo

Description

This call returns certain file attributes of an existing open or closed block file. See also the SetFileInfo call.

△ **Important** A GetFileInfo call following a SetFileInfo call on an open file may not return the values set by the SetFileInfo call. To guarantee recording of the attributes specified in a SetFileInfo call, you must first close the file. △

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 2)
$02	pathname	1	Longword input pointer
$06	access	2	Word result value
$08	fileType	3	Word result value
$0A	auxType	4	Longword result value
$0E	storageType	5	Word result value
$10	createDateTime	6	Double longword result value
$18			

$18	modDateTime	7	Double longword result value
$20	optionList	8	Longword input pointer
$24	eof	9	Longword result value
$28	blocksUsed	10	Longword result value
$2C	resourceEOF	11	Longword result value
$30	resourceBlocks	12.	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 12.

`pathname` Longword input pointer: Points to a GS/OS string representing the pathname of the file whose file information is to be retrieved.

`access` Word result value: Access attribute of the file, described under the Create call.

`fileType` Word result value: File type attribute of the file.

`auxType` Longword result value: Auxiliary type attribute of the file.

`storageType` Word result value: Storage type of the file, as follows:

$01 = standard file

$05 = extended file

$0D = volume directory or subdirectory file

`createDateTime` Double longword result value: Creation date and time attributes of the file. The format of the date and time is shown in Table 4-2 in Chapter 4.

`modDateTime` Double longword result value: Modification date and time attributes of the file. The format of the date and time is shown in Table 4-2 in Chapter 4.

`optionList` Longword input pointer: Points to a result buffer. On output, GS/OS sets the output length field to a value giving the number of bytes of space required by the output data, excluding the length words. GS/OS will not overflow the available output data area.

`eof` Longword result value: For a standard file, specifies the number of bytes that can be read from the file. For an extended file, this parameter specifies the number of bytes that can be read from the file's data fork.

For a subdirectory or a volume directory file, this parameter is undefined.

`blocksUsed` Longword result value: For a standard file, specifies the total number of blocks used by the file. For an extended file, this parameter specifies the number of blocks used by the file's data fork.

For a subdirectory or a volume directory file, this parameter is undefined.

`resourceEOF` Longword result value: If the specified file is an extended file, this parameter gives the number of bytes that can be read from the file's resource fork. Otherwise, the parameter is undefined.

`resourceBlocks` Longword result value: If the specified file is an extended file, this parameter gives the number of blocks used by the file's resource fork. Otherwise, the parameter is undefined.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4A	`badFileFormat`	version error
$4B	`badStoreType`	unsupported storage type
$52	`unknownVol`	unsupported volume type
$58	`notBlockDev`	not a block device

$202B GetFSTInfo

Description

This function returns general information about a specified File System Translator (FST). See also the SetFSTInfo call, and Part II of this book.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 2)
$02	`fstNum`	1	Word input value
$04	`fileSysID`	2	Word result value
$06	`fstName`	3	Longword input pointer
$0A	`version`	4	Word result value
$0C	`attributes`	5	Word result value
$0E	`blockSize`	6	Word result value
$10	`maxVolSize`	7	Longword result value
$14	`maxFileSize`	8	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 8.

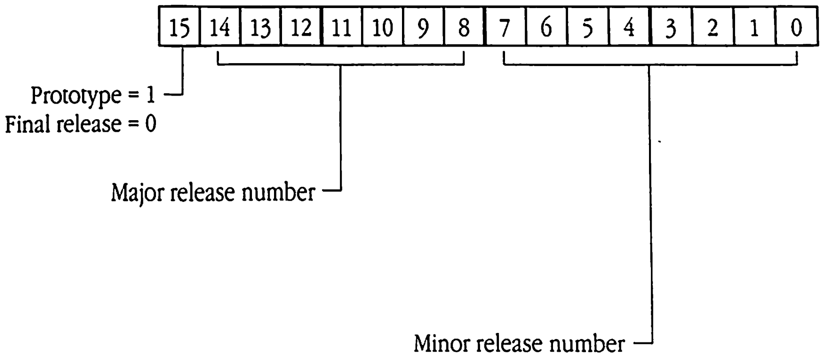
`fstNum` Word input value: FST number. GS/OS assigns FST numbers in sequence (1, 2, 3, and so on) as it loads the FSTs. There is no fixed correspondence between FSTs and FST numbers. To get information about every FST in the system, make repeated calls to GetFSTInfo with `fstNum` values of 1, 2, 3, and so on until GS/OS returns error $53 (`paramRangeErr`).

`fileSysID` Word result value: Identifies the file system as follows:

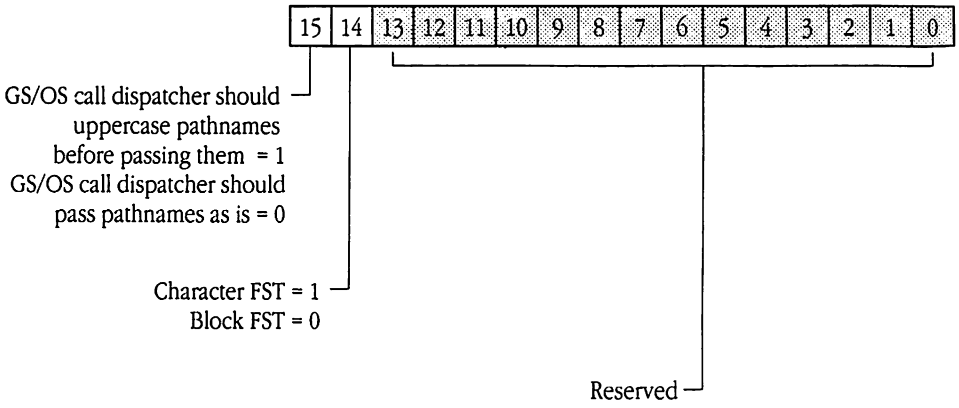
$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E–$000F	Reserved
$0007	Lisa		

`fstName` Longword input pointer: Points to a result buffer where GS/OS is to return the name of the FST.

`version` Word result value: Version number of the FST, in the following format:



`attributes` Word result value: General attributes of the FST, as follows:


15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
GS/OS call dispatcher should uppercase pathnames before passing them = 1
GS/OS call dispatcher should pass pathnames as is = 0
Character FST = 1
Block FST = 0
Reserved

`blockSize` Word result value: Block size (in bytes) of blocks handled by the FST.

`maxVolSize` Longword result value: Maximum size (in blocks) of volumes handled by the FST.

`maxFileSize` Longword result value: Maximum size (in bytes) of files handled by the FST.

Errors

\$53	`paramRangeErr`	parameter out of range

$201B GetLevel

Description This function returns the current value of the system file level. See also the SetLevel call.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	level	1	Word result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`level` Word result value: The value of the system file level.

Errors $59 `invalidLevel` invalid file level

$2017 GetMark

Description This function returns the current file mark for the specified file. See also the SetMark call.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 2)
$02	refNum	1	Word input value
$04	position	2	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 2.

`refNum` Word input value: Identifying number assigned to the file by the Open call.

`position` Longword result value: Current value of the file mark in bytes relative to the beginning of the file.

Errors $43 `invalidRefNum` invalid reference number

$2027 GetName

Description

Returns the filename (not the complete pathname) of the currently running application program.

To get the complete pathname of the current application, concatenate prefix `9/` with the filename returned by this call. Do this before making any change in prefix `9/`.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`dataBuffer`	1	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`dataBuffer` Longword input pointer: Points to a result buffer where the filename is to be returned.

Errors

$4F	`buffTooSmall`	buffer too small

$200A GetPrefix

Description

This function returns the current value of any one of the numbered prefixes. The returned prefix string will always start and end with a separator. If the requested prefix is null, it is returned as a string with the length field set to 0. This call should not be used to get the boot volume prefix (`*/`); use the GetBootVol call to do that. See also the SetPrefix call.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 2)
$02	`prefixNum`	1	Word input value
$04	`prefix`	2	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 2.

`prefixNum` Word input value: Binary value of the prefix number for the prefix to be returned.

`prefix` Longword input pointer: Pointer to a GS/OS output string where the prefix value is returned.

Errors

$4F	`buffTooSmall`	buffer too small

$2039 GetRefInfo

Description

This function returns the access attributes and full pathname for an open file when the reference number is given as input.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 2)
$02	refNum	1	Word input value
$04	access	2	Word output value
$06	pathname	3	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 3.

`refNum` Word input value: Reference number of the open file.

`access` Word output value: Access attributes of the open file, as follows:

1 = read only

2 = write only

3 = read/write

`pathname` Longword input pointer: Points to a GS/OS output string where GS/OS places the full pathname of the file selected by the `refNum` parameter.

Errors

$43	`invalidRefNum`	invalid reference number

$2038 GetRefNum

Description This function returns the reference number and access attributes for an open file when the full pathname is given as input.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 2)
$02	`pathname`	1	Longword input pointer
$06	`refNum`	2	Word output value
$08	`access`	3	Word output value
$0A	`resNum`	4	Word input value
$0C	`caseSense`	5	Word input value
$0E	`displacement`	6	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 6.

`pathname` Longword input pointer: Points to a GS/OS string representing the pathname of an open file. This name cannot contain a device name or number; if one is present, error $40 (`badPathSyntax`) is returned.

`refNum` Word output value: Reference number of open file or count of matching open files if the `displacement` parameter = 0.

`access` Word output value: Access attributes of the open file, as follows:

1 = read only

2 = write only

3 = read/write

`resNum` Word input value: Selects the data or resource fork for the file, as follows:

0 = data fork (default value)

1 = resource fork

`caseSense` Word input value: Selects case sensitivity for pathname comparison, as follows:

0 = case insensitive (default value)

1 = case sensitive

`displacement` Word input value: Selects the *n*th matching open file. Default is 1. If set to 0, the call returns the number of matching files in the `refNum` parameter.

Errors

$40	`badPathSyntax`	invalid pathname syntax
$60	`dataUnavail`	data unavailable; no matching open files

$2037 GetStdRefNum

Description

This function returns the reference number of the last open call to any of the three standard prefixes (`10`, `11`, and `12`), if available. The reference number will be available if

- a GS/OS Open call has been made with a path that is a prefix number only and is `10`, `11`, or `12`
- a GS/OS Close call has not been made with the reference number or with 0

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 2)
$02	`prefixNum`	1	Word input value
$04	`refNum`	2	Word output value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 2.

`prefixNum` Word input value: Decimal value of the prefix number of the prefix; `10`, `11`, and `12` are valid prefixes.

`refNum` Word output value: Reference number from the last Open call to the selected prefix number.

Errors

$60 `dataUnavail` data unavailable; no matching open files

$200F GetSysPrefs

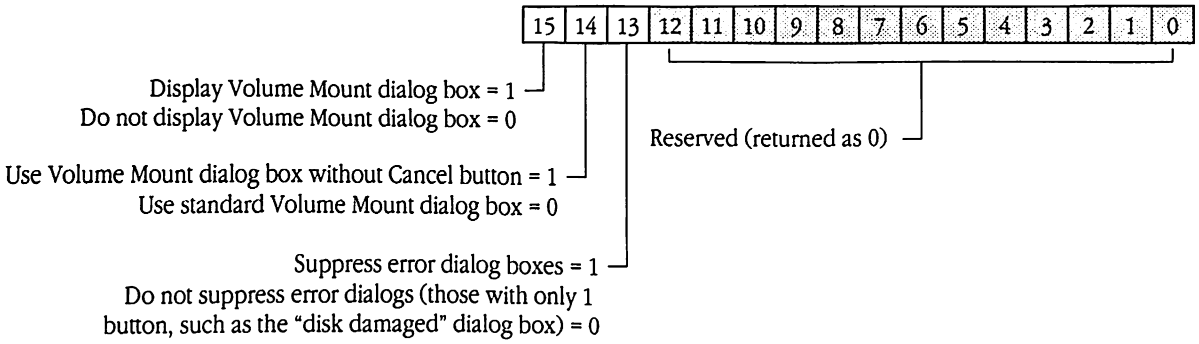
Description This call returns the value of the current global system preferences. The value of system preferences affects the behavior of some system calls. See also the SetSysPrefs call.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	preferences	1	Word result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`preferences` Word result value: Value of system preferences, as follows:



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Display Volume Mount dialog box = 1
Do not display Volume Mount dialog box = 0
Use Volume Mount dialog box without Cancel button = 1
Use standard Volume Mount dialog box = 0
Suppress error dialog boxes = 1
Do not suppress error dialogs (those with only 1 button, such as the "disk damaged" dialog box) = 0
Reserved (returned as 0)

Errors (none except general GS/OS errors)

$202A GetVersion

Description

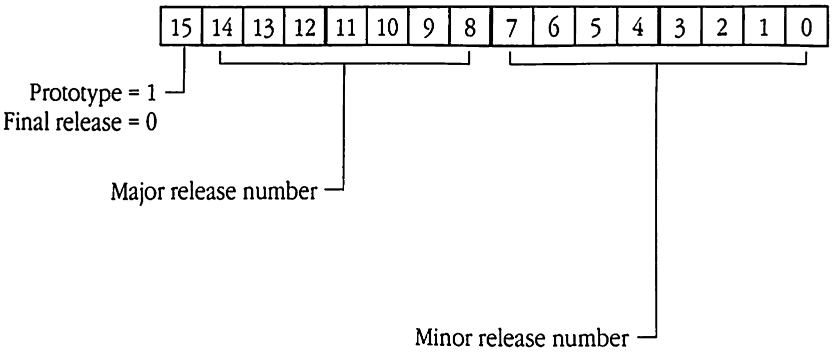
This call returns the version number of the GS/OS operating system. This value can be used by application programs to condition version-dependent operations.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	version	1	Word result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`version` Word result value: Version number of the operating system, in the following format:



Errors

(none except general GS/OS errors)

$2011 NewLine

Description

This function enables or disables the newline read mode for an open file and, when enabling newline read mode, specifies the newline enable mask and newline character or characters.

When newline mode is disabled, a Read call terminates only after it reads the requested number of characters or encounters the end of file. When newline mode is enabled, the read also terminates if it encounters one of the specified newline characters.

When a Read call is made while newline mode is enabled and a character remains in the file, GS/OS performs the following operations:

1. Transfers the next character to the user's buffer.
2. Performs a logical AND operation between the character and the low-order byte of the newline mask specified in the last NewLine call for the open file.
3. Compares the resulting byte with the newline character or characters.
4. If there is a match, terminates the read; otherwise returns to step 1.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 4)
$02	`refNum`	1	Word input value
$04	`enableMask`	2	Word input value
$06	`numChars`	3	Word input value
$08	`newlineTable`	4	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 4; maximum = 4.

`refNum` Word input value: Identifying number assigned to the file access path by the Open call.

`enableMask` Word input value: $0000 disables newline mode. If the value is greater than $0000, the low-order byte becomes the newline mask. GS/OS performs a logical AND operation of each input character with the newline mask before comparing it to the newline character or characters.

`numChars` Word input value: Number of newline characters contained in the newline character table. If the `enableMask` parameter is nonzero, this parameter must be in the range 1–256. When disabling newline mode (`enableMask` = $0000), this parameter is ignored.

`newlineTable` Longword input pointer: Points to a table of from 1 to 256 bytes that specifies the set of newline characters. Each byte holds a distinct newline character. When disabling newline mode (`enableMask` = $0000), this parameter is ignored.

Errors $43 `invalidRefNum` invalid reference number

$200D Null

Description

This call executes any pending events in the GS/OS event queue and in the Scheduler queue before returning to the calling application. Note that every GS/OS call performs these functions. This call provides a way to flush the queues without doing anything else.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 0)

pCount Word input value: Number of parameters in this parameter block. Minimum = 0; maximum = 0.

Errors

(none except general GS/OS errors)

$2010 Open

Description

This call causes GS/OS to establish an access path to a file. Once an access path is established, the user may perform Read and Write operations and other related operations on the file. This call can also return all the file information returned by the GetFileInfo call.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 2)
$02	refNum	1	Word result value
$04	pathname	2	Longword input pointer
$08	requestAccess	3	Word input value
$0A	resourceNumber	4	Word input value
$0C	access	5	Word result value
$0E	fileType	6	Word result value
$10	auxType	7	Longword result value
$14	storageType	8	Word result value
$16	createDateTime	9	Double longword result value
$1E			

(Continued)

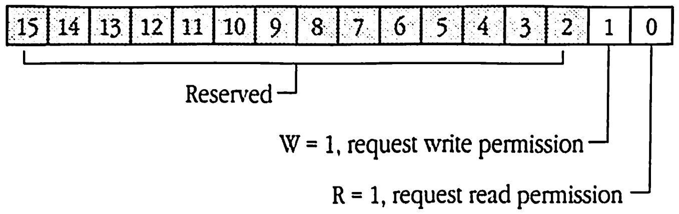
$1E	modDateTime	10	Double longword result value
$26	optionList	11	Longword input pointer
$2A	eof	12	Longword result value
$2E	blocksUsed	13	Longword result value
$32	resourceEOF	14	Longword result value
$36	resourceBlocks	15	Longword result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 15.

`refNum` Word result value: Reference number assigned by GS/OS to the access path. All other file operations (Read, Write, Close, and so on) refer to the access path by this number.

`pathname` Longword input pointer: Points to a GS/OS string representing the pathname of the file to be opened.

`requestAccess` Word input value: Specifies desired access permissions, as follows:


15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Reserved
W = 1, request write permission
R = 1, request read permission

If this parameter is not included or its value is $0000, the file is opened with access permissions determined by the file's stored access attributes.

`resourceNumber` Word input value: Meaningful only when the `pathname` parameter specifies an extended file. In that case, a value of $0000 tells GS/OS to open the data fork, and a value of $0001 tells it to open the resource fork.

`access` Word result value: Access attribute of the file, described under the Create call.

`fileType` Word result value: File type attribute.

`auxType` Longword result value: Auxiliary type attribute.

`storageType` Word result value: Storage type attribute, as follows:

$01 = standard file

$05 = extended file

$0D = volume directory or subdirectory file

`createDateTime` Double longword result value: Creation date and time attributes of the file. The format of the date and time is shown in Table 4-2 in Chapter 4.

`modDateTime` Double longword result value: Modification date and time attributes of the file. The format of the date and time is shown in Table 4-2 in Chapter 4.

`optionList` Longword input pointer: Points to a GS/OS result buffer to which FST-specific information can be returned. On output, GS/OS sets the output length field to a value giving the number of bytes of space required by the output data, excluding the length words. GS/OS will not overflow the available output data area.

`eof` Longword result value: For a standard file, indicates the number of bytes that can be read from the file. For an extended file, this parameter indicates the number of bytes that can be read from the file's data fork.

For a subdirectory or volume directory file, this parameter is undefined.

`blocksUsed` Longword result value: For a standard file, indicates the number of bytes used by the file. For an extended file, this parameter indicates the number of bytes used by the file's data fork.

For a subdirectory or volume directory file, this parameter is undefined.

`resourceEOF` Longword result value: If the specified file is an extended file, indicates the number of bytes that can be read from the file's resource fork, even when you are opening the data fork. Otherwise, the parameter is undefined.

`resourceBlocks` Longword result value: If the specified file is an extended file, this parameter indicates the number of blocks used by the file's resource fork, even if you are opening the data fork. Otherwise, the parameter is undefined.

Errors

$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2E	`drvrDiskSwitch`	disk switched
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4A	`badFileFormat`	version error
$4E	`invalidAccess`	file not destroy-enabled
$4F	`buffTooSmall`	buffer too small
$50	`fileBusy`	file open
$52	`unknownVol`	unsupported volume type
$58	`notBlockDev`	not a block device

$2003 OSShutdown

Description

This call allows an application (such as the Finder) to shut down the operating system in preparation for either powering down the machine or performing a restart. GS/OS terminates any write-deferral session in progress and shuts down all drivers and FSTs.

The action of the call is determined by the values of the `shutdownFlag` parameter. If bit 0 is set to 1, GS/OS performs the shutdown operation and restarts the machine. If bit 0 is cleared to 0, GS/OS performs the same shutdown procedure and then displays a dialog box that allows the user to either power down the computer or restart. If the user chooses to restart, GS/OS then looks at bit 1 of the `shutdownFlag` parameter.

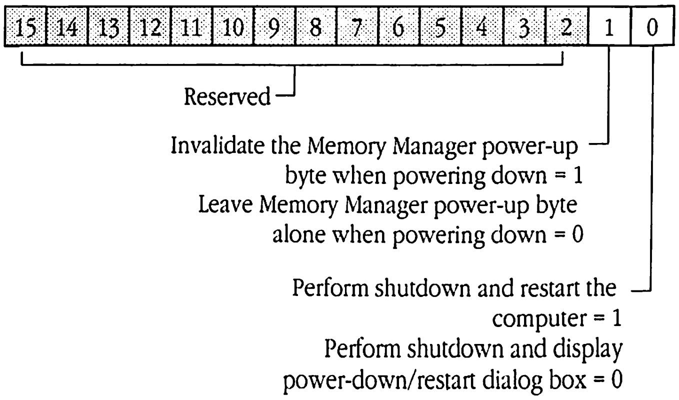
If bit 1 is cleared to 0, GS/OS leaves the Memory Manager power-up byte alone; this leaves any RAM disks intact while the machine is restarted. If bit 1 is set to 1, however, GS/OS invalidates the power-up byte, which effectively erases any RAM disk, before restarting the computer.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`shutdownFlag`	1	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`shutdownFlag` Word input value: Two Boolean flags that specify information about how to handle the shutdown, as follows:


15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Reserved
Invalidate the Memory Manager power-up byte when powering down = 1
Leave Memory Manager power-up byte alone when powering down = 0
Perform shutdown and restart the computer = 1
Perform shutdown and display power-down/restart dialog box = 0

Errors

(none except general GS/OS errors)

$2029 Quit

Description

This call terminates the running application. It also closes all open files, sets the system file level to 0, initializes certain components of the Apple IIGS and the operating system, and then launches the next application.

For more information about quitting applications, see Chapter 2, "GS/OS and Its Environment."

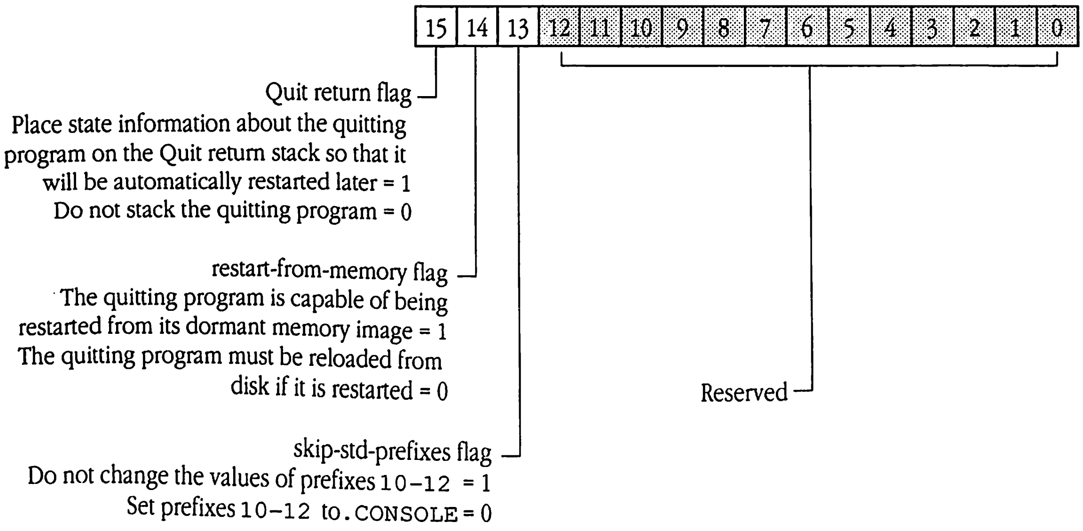
Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 0)
$02	pathname	1	Longword input pointer
$06	flags	2	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 0; maximum = 2.

`pathname` Longword input pointer: Points to a GS/OS string representing the pathname of the program to run next. If this parameter is NIL or the pathname itself has length 0, GS/OS chooses the next application, as described in Chapter 2.

`flags` Word input value: Three Boolean flags that specify information about the Quit call, as follows:



Quit return flag
Place state information about the quitting program on the Quit return stack so that it will be automatically restarted later = 1
Do not stack the quitting program = 0

restart-from-memory flag
The quitting program is capable of being restarted from its dormant memory image = 1
The quitting program must be reloaded from disk if it is restarted = 0

skip-std-prefixes flag
Do not change the values of prefixes `10-12` = 1
Set prefixes `10-12` to `.CONSOLE` = 0

Reserved

Comments Only global errors cause the Quit call to return to the calling application. All other errors are managed within the GS/OS Program Dispatcher.

Errors (none except general GS/OS errors)

$2012 Read

Description

This function attempts to transfer the number of bytes given by the `requestCount` parameter, starting at the current mark, from the file specified by the `refNum` parameter into the buffer pointed to by the `dataBuffer` parameter. The function updates the file mark to reflect the new file position after the read.

Because some situations cause the Read function to transfer fewer than the requested number of bytes, the function returns the actual number of bytes transferred in the `transferCount` parameter, as follows:

- If GS/OS reaches the end of file before transferring the number of bytes specified in `requestCount`, it stops reading and sets `transferCount` to the number of bytes actually read.
- If newline mode is enabled and a newline character is encountered before the requested number of bytes have been read, GS/OS stops the transfer and sets `transferCount` to the number of bytes actually read, including the newline character.
- If the device is a character device and no-wait mode is enabled, the call returns immediately, with `transferCount` indicating the number of characters returned.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 4)
$02	`refNum`	1	Word input value
$04	`dataBuffer`	2	Longword input pointer
$08	`requestCount`	3	Longword input value
$0C	`transferCount`	4	Longword result value
$10	`cachePriority`	5	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 4; maximum = 5.

`refNum` Word input value: Identifying number assigned to the file by the Open call.

`dataBuffer` Longword input pointer: Points to a memory area large enough to hold the requested data.

`requestCount` Longword input value: Number of bytes to be read.

`transferCount` Longword result value: Number of bytes actually read.

`cachePriority` Word input value: Specifies whether or not disk blocks handled by the read call are candidates for caching, as follows:

$0000 = do not cache blocks involved in this read

$0001 = cache blocks involved in this read if possible

Errors

$27	`drvrIOError`	I/O error
$2E	`drvrDiskSwitch`	disk switched
$43	`invalidRefNum`	invalid reference number
$4C	`eofEncountered`	end-of-file encountered
$4E	`invalidAccess`	access not allowed

$2026 ResetCache

Description

This call provides a way for a program to resize the GS/OS cache and be able to use the resized cache immediately. The call ends the current write-deferral session immediately, shuts down the cache, and then reinitializes the cache with a new size determined by the cache size field in battery RAM.

Before your application makes this call, it should use the Miscellaneous Tool Set call `WriteBParam` to set the new cache size. The battery RAM parameter number for the cache size parameter is $0081. The value of this parameter is the number of 32K blocks to use for the cache size, and the valid range for this value is $00–$FE. For more information on `WriteBParam`, see the *Apple IIGS Toolbox Reference.*

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 0)

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 0; maximum = 0.

Comments

The following sample code shows how to use this call:

```
cache_size equ   $10000    ;64K cache
.
.
.
           pea   cache_size/$8000 ;convert size into # of
                                  ;blocks
           pea   $0081     ;battery RAM parameter #
           ldx   #$0B03    ;write battery parameter
           jsl   $e10000   ;no error is possible

           _ResetCache parms  ;tell GS/OS to reset cache
.
.
.
parms      dc    i2'0'      ;parm block for ResetCache call
```

Errors

(none except general GS/OS errors)

$201F SessionStatus

Description

This call returns a value that tells whether or not a write-deferral session is in progress. See also BeginSession and EndSession in this chapter.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	status	1	Word result value

pCount Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

status Word result value: Indicates whether a write-deferral session is in progress, as follows:

$0000	no session in progress
$0001	session in progress

Errors

(none except general GS/OS errors)

$2018 SetEOF

Description

This call sets the logical size of an open file to a specified value that may be either larger or smaller than the current file size. The EOF value cannot be changed unless the file is write-enabled. If the specified EOF is less than the current EOF, the system may—but need not—free blocks that are no longer needed to represent the file. See also the GetEOF call.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 3)
$02	refNum	1	Word input value
$04	base	2	Word input value
$06	displacement	3	Longword input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 3; maximum = 3.

`refNum` Word input value: Identifying number assigned to the file by the Open call.

`base` Word input value: Specifies how to interpret the `displacement` parameter.

$0000	set EOF equal to displacement
$0001	set EOF equal to old EOF minus displacement
$0002	set EOF equal to file mark plus displacement
$0003	set EOF equal to file mark minus displacement

`displacement` Longword input value: Used to compute the new value of the EOF as described for the `base` parameter.

Errors

$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$43	`invalidRefNum`	invalid reference number
$4D	`outOfRange`	position out of range
$4E	`invalidAccess`	file not write-enabled
$5A	`damagedBitMap`	block number out of range

$2005 SetFileInfo

Description

This call sets certain file attributes of an existing open or closed block file. This call immediately modifies the file information in the file's directory entry whether the file is open or closed. It does not affect the file information seen by previously opened access paths to the same file.

> △ **Important** A GetFileInfo call following a SetFileInfo call on an open file may not return the values set by the SetFileInfo call. To guarantee recording of the attributes specified in a SetFileInfo call, you must first close the file. △

See also the GetFileInfo call.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 2)
$02	pathname	1	Longword input pointer
$06	access	2	Word input value
$08	fileType	3	Word input value
$0A	auxType	4	Longword result value
$0E	reserved	5	Word input value
$10			

(Continued)

$10	createDateTime	6	Double longword input value
$18	modDateTime	7	Double longword input value
$20	optionList	8	Longword input pointer
$24	reserved	9	Longword input value
$28	reserved	10	Longword input value
$2C	reserved	11	Longword input value
$30	reserved	12	Longword input value

pCount Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 12.

pathname Longword input pointer: Points to a GS/OS string representing the pathname of the file whose file information is to be set.

access Word input value: Access attribute of the file, described under the Create call.

fileType Word input value: File type attribute of the file.

auxType Longword result value: Auxiliary type attribute of the file.

reserved Word input value: Reserved for use by GS/OS. The value you place here is ignored.

createDateTime Double longword input value: Creation date and time attributes of the file. If the value of this parameter is zero, GS/OS does not change the creation date and time. The format of the date and time is shown in Table 4-2 in Chapter 4.

modDateTime Double longword input value: Modification date and time attributes of the file. If the value of this entire parameter is zero, GS/OS sets the modification date and time with the current system clock value. The format of the date and time is shown in Table 4-2 in Chapter 4.

optionList Longword input pointer: Points to a GS/OS result buffer to which FST-specific information can be returned.

reserved Longword input value: Reserved for use by GS/OS. The value you place here is ignored.

reserved Longword input value: Reserved for use by GS/OS. The value you place here is ignored.

reserved Longword input value: Reserved for use by GS/OS. The value you place here is ignored.

reserved Longword input value: Reserved for use by GS/OS. The value you place here is ignored.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4A	`badFileFormat`	version error
$4B	`badStoreType`	unsupported storage type
$4E	`invalidAccess`	file not destroy-enabled
$52	`unknownVol`	unsupported volume type
$58	`notBlockDev`	not a block device

$201A SetLevel

Description

This function sets the current value of the system file level.

Whenever a file is opened, GS/OS assigns it a file level equal to the current system file level. A Close call with a reference number of $0000 closes all files with file level values at or above the current system file level. Similarly, a Flush call with a reference number of $0000 flushes all files with file level values at or above the current system file level. See also the GetLevel call.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`level`	1	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`level` Word input value: New value of the system file level. Must be in the range $0000–$00FF.

Errors

$59	`invalidLevel`	invalid file level

$2016 SetMark

Description

This call sets the file mark (the position from which the next byte will be read or to which the next byte will be written) to a specified value. The value can never exceed EOF, the current size of the file. See also the GetMark call.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 3)
$02	refNum	1	Word input value
$04	base	2	Word input value
$06	displacement	3	Longword input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 3; maximum = 3.

`refNum` Word input value: Identifying number assigned to the file by the Open call.

`base` Word input value: Specifies how the `displacement` parameter should be interpreted, as follows:

$0000	set mark equal to displacement
$0001	set mark equal to EOF minus displacement
$0002	set mark equal to old mark plus displacement
$0003	set mark equal to old mark minus displacement

`displacement` Longword input value: Used to compute the new value for the file mark, as described for the `base` parameter.

Errors

$27	`drvrIOError`	I/O error
$43	`invalidRefNum`	invalid reference number
$4D	`outOfRange`	position out of range
$5A	`damagedBitMap`	block number out of range

$2009 SetPrefix

Description

This call sets one of the numbered pathname prefixes to a specified value. The input to this call can be any of the following pathnames:

- a full pathname
- a partial pathname beginning with a numeric prefix designator
- a partial pathname beginning with the special prefix designator `*/`
- a partial pathname without an initial prefix designator

The SetPrefix call is unusual in the way it treats partial pathnames without initial prefix designators. Normally, GS/OS uses the prefix `0/` in the absence of an explicit designator. However, only in the SetPrefix call, it uses the prefix *n*/ where *n* is the value of the `prefixNum` parameter. See also the GetPrefix call.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 2)
$02	`prefixNum`	1	Word input value
$04	`prefix`	2	Longword input pointer

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 2.

`prefixNum` Word input value: Prefix number that specifies the prefix to be set.

`prefix` Longword input pointer: Points to a GS/OS string representing the pathname to which the prefix is to be set.

Comments

Specifying a pathname with length 0 or whose syntax is illegal sets the designated prefix to NULL. GS/OS does not check to make sure that the designated prefix corresponds to an existing subdirectory or file.

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

Errors

$40	`badPathSyntax`	invalid pathname syntax

$200C SetSysPrefs

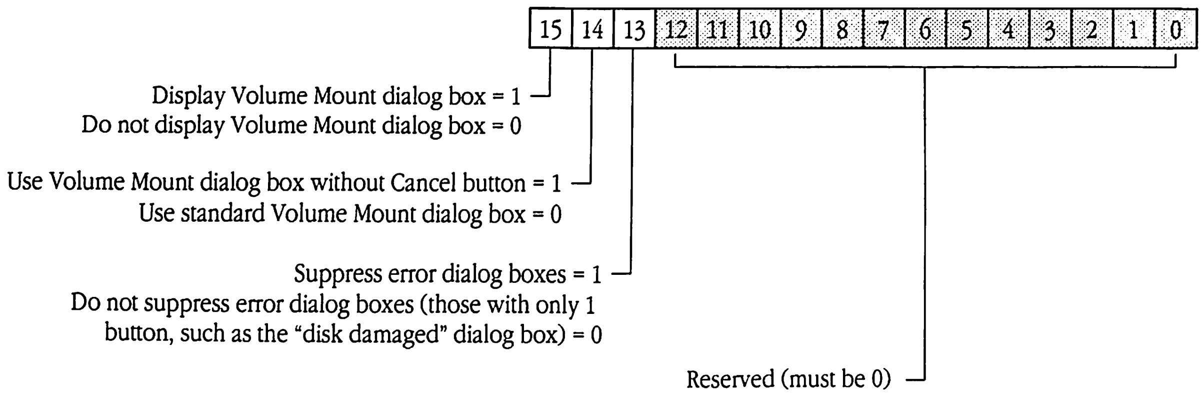
Description This call sets the value of the global system preferences. The value of system preferences affects the behavior of some system calls. See also the GetSysPrefs call.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`preferences`	1	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`preferences` Word input value: Value of system preferences, as follows:


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Display Volume Mount dialog box = 1
Do not display Volume Mount dialog box = 0
Use Volume Mount dialog box without Cancel button = 1
Use standard Volume Mount dialog box = 0
Suppress error dialog boxes = 1
Do not suppress error dialog boxes (those with only 1 button, such as the "disk damaged" dialog box) = 0
Reserved (must be 0)

◆ *Note:* If bits 14 and 13 are both set to 1, the Volume Mount dialog box is always suppressed, because it only has one button.

Comments Under certain circumstances, parts of the system call the system's Mount facility to display a dialog box asking the user to mount a specified volume. This can happen when the call contains a reference number parameter or a pathname parameter.

- For those calls that specify a reference number parameter (for example Read, Write, Close), Mount displays the dialog box if bits 14 and 13 are not both set to 1.
- For those calls that specify a pathname parameter, the Mount facility displays the dialog box only if system-preference bit 15 is 1 and bits 14 and 13 are not both set to 1. Otherwise, Mount returns the `Cancel` return code, which normally causes the system to return a volume-not-found error. Thus, an application can be written either to handle volume-not-found errors itself (system-preference bit 15 = 0) or to allow the system to automatically display mount dialog boxes (bit 15 = 1), except when the System Loader is attempting to load a dynamic segment.
- For those calls that result in the System Loader attempting to load a dynamic segment, the System Loader always sets the system preference bit (bit 15) to 1, and then resets it to its original value when the segment has been loaded. Thus, the Volume Mount dialog box is always displayed when a dynamic segment is requested, unless bits 14 and 13 are both set to 1.

Errors (none except general GS/OS errors)

$2032 UnbindInt

Description

This function removes a specified interrupt handler from the interrupt vector table.

For a complete description of the GS/OS interrupt handling subsystem, see the *GS/OS Device Driver Reference*. See also the BindInt call in this chapter.

Parameters

Offset		No.	Size and type
$00	`pCount`	—	Word input value (minimum = 1)
$02	`intNum`	1	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 1; maximum = 1.

`intNum` Word input value: Interrupt identification number of the binding between interrupt source and interrupt handler that is to be undone.

Errors

(none except general GS/OS errors)

$2008 Volume

Description Given the name of a block device, this call returns the name of the volume mounted in the device, along with other information about the volume.

Parameters

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 2)
$02	devName	1	Longword input pointer
$06	volName	2	Longword input pointer
$0A	totalBlocks	3	Longword result value
$0E	freeBlocks	4	Longword result value
$12	fileSysID	5	Word result value
$14	blockSize	6	Word result value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 2; maximum = 6.

`devName` Longword input pointer: Points to a GS/OS input string containing the name of a block device.

`volName` Longword input pointer: Points to a GS/OS output string where GS/OS returns the volume name of the volume mounted in the device.

`totalBlocks` Longword result value: Total number of blocks contained on the volume.

`freeBlocks` Longword result value: The number of free (unallocated) blocks on the volume.

`fileSysID` Word result value: Identifies the file system contained on the volume, as follows:

$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E–$000F	Reserved
$0007	Lisa		

`blockSize` Word result value: The size, in bytes, of a block.

Errors

$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device request
$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2E	`drvrDiskSwitch`	disk switched
$45	`volNotFound`	volume not found
$4A	`badFileFormat`	version error
$52	`unknownVol`	unsupported volume type
$57	`dupVolume`	duplicate volume
$58	`notBlockDev`	not a block device

$2013 Write

Description

This call attempts to transfer the number of bytes specified by the `requestCount` parameter from the caller's buffer to the file specified by the `refNum` parameter, starting at the current file mark.

The function returns the number of bytes actually transferred. The function updates the file mark to indicate the new file position and extends the EOF, if necessary, to accommodate the new data.

Parameters

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 4)
$02	`refNum`	1	Word input value
$04	`dataBuffer`	2	Longword input pointer
$08	`requestCount`	3	Longword input value
$0C	`transferCount`	4	Longword result value
$10	`cachePriority`	5	Word input value

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 4; maximum = 5.

`refNum` Word input value: Identifying number assigned to the file by the Open call.

`dataBuffer` Longword input pointer: Points to the area of memory containing the data to be written to the file.

`requestCount` Longword input value: Number of bytes to write.

`transferCount` Longword result value: Number of bytes actually written.

`cachePriority` Word input value: Specifies whether or not disk blocks handled by the call are candidates for caching, as follows:

$0000 do not cache blocks involved in this call

$0001 cache blocks involved in this call if possible

Errors

$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2E	`drvrDiskSwitch`	disk switched
$43	`invalidRefNum`	invalid reference number
$48	`volumeFull`	volume full
$4E	`invalidAccess`	file not destroy-enabled
$5A	`damagedBitMap`	block number out of range

Chapter 8 **Loading Program Files**

Because the Apple IIGS has a large amount of available memory, a flexible, dynamic facility for loading program files is required. Programs should be able to be loaded in any available location in memory. The burden of determining where to load a program should be on the system, not on the application writer. Furthermore, programs should be able to be broken into smaller program segments that can be loaded independently.

To provide these capabilities, GS/OS comes with two relocating segment loaders called **ExpressLoad** and the **System Loader.** These loaders, collectively referred to as the **GS/OS Loaders,** provide very powerful and flexible facilities that are not available on standard Apple II computers.

How the GS/OS Loaders work

Apple II computers running under ProDOS 8 have a very simple program loader. The loader is the part of the boot code that searches the boot disk for the first **system file** (any file of ProDOS file type $FF whose name ends with .SYSTEM) and loads it into location $2000. If a program wants to load another program, it has to do all the work by making ProDOS 8 calls.

Some programming environments such as Apple II Pascal and AppleSoft BASIC provide loaders for programs running under them. The AppleSoft loader loads either system files, BASIC files, or binary code files. All these files are loaded either at a fixed address in memory or at an address specified in the file.

The Apple IIGS GS/OS Loaders under GS/OS can load programs in any available part of memory, relieving the application writer of deciding where to put the code and how to make it execute properly at that location. Furthermore, the GS/OS Loaders can load individual segments rather than whole files, either at program start or during execution.

The GS/OS Loaders load programs or program segments by first calling the Memory Manager to find available memory. They load each segment independently and perform relocation during the load as necessary. Therefore, a large application can be broken up into smaller program segments, each of which is put into a separate location in memory. The application's segments can also be loaded dynamically, as they are referenced, rather than at program boot time. Additionally, the GS/OS Loaders can be called by the application itself to load and unload program (or data) segments.

Definitions

The GS/OS Loaders process load files, generated from object files by a linker. Definitions of these and related terms may help make the following discussion clearer.

Object files are the output from an assembler or compiler and are the input to a linker.

A **linker** is the program that combines object files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

Load files are the output of a linker and contain memory images, which the GS/OS Loaders can load into memory. There are several types of load files, reflecting the types of programs they contain.

The **GS/OS Loaders** are the part of system software that reads the files generated by the linker and loads them into memory (performing relocation if necessary).

Relocation is the process of modifying a load file in memory so that it will execute correctly. It consists of patching operands to reflect the code's current memory location.

Library files are special object files, containing general program segments that the linker can search.

Run-time library files are special load files, containing general program segments that can be loaded as needed by the GS/OS Loaders and shared between applications.

Object module format (OMF) is the general format used in object files, library files, and load files.

An **OMF file** is a file in object module format (an object file, library file, or load file).

A **segment** is an individual component of an OMF file. Each file contains one or more segments; object files contain object segments, and load files contain load segments.

A **controlling program** is a program that uses GS/OS Loader calls to load and execute another program, and is responsible for shutting down the program when it exits. Operating systems and shells are controlling programs.

Segments and the GS/OS Loaders

The GS/OS Loaders process only those files that conform to the Apple IIGS definition of a load file, as generated by the linker. A load file consists of load segments, each of which can be loaded independently. The load segments are numbered sequentially from 1.

Certain load segments are **static segments.** They are loaded into memory at program start (initial load) and must stay in memory until program completion.

Other load segments are **dynamic segments.** Dynamic segments are loaded not at boot time but during program execution. This can be done automatically (by means of the jump table mechanism) or manually (at the specific request of the application). When dynamic segments are not needed by a program, they can be purged (their contents deallocated) by the program.

Load segments can have several other attributes; see Appendix F for a complete list of attributes.

Segments are classified numerically by kind (the value of the KIND field in the segment header; see Appendix F, "Object Module Format," for more information.). In addition to segments containing program code or data, there are several special kinds of load segments:

- The jump-table segment (KIND = $02), when loaded into memory, becomes part of the **jump table.** The jump table provides a mechanism whereby segments in memory can trigger the loading of other segments not yet in memory.
- The pathname table segment (KIND = $04) contains information about the run-time library files that are referenced. The pathname table and run-time library files are described in Appendix F, "Object Module Format."
- Initialization segments (KIND = $10) in a load file are used for code that is to be executed before all the rest of the load segments are loaded.
- The direct-page/stack segment (KIND = $12) defines the application's direct-page and stack requirements. This segment is loaded into bank $00 and its starting address and length are passed to the controlling program. The controlling program in turn sets the direct register and stack pointer to the start and end of this segment before transferring control to the program.

If a GS/OS Loader is called to perform the initial load of a program, it loads all the static load segments and the jump table and pathname table segments (if they exist).

References to dynamic segments

During the initial load, GS/OS Loaders have all the information needed to resolve all intersegment references between the static load segments. But during the dynamic loading of dynamic load segments, they can only resolve references in the dynamic load segment to the already loaded static load segments. Therefore, the general rule is that static segments can be referenced by any type of segment, but dynamic segments can only be referenced through JSL calls through the jump table.

Unmounted volume

If a GS/OS Loader references a file on a volume that is not mounted, GS/OS either returns error $45 (volNotFound) or displays a mount-volume message (depending on the state of the system preferences at the time of the call; see "SetSysPrefs" in Chapter 7). If a mount message is displayed, GS/OS handles the user interface and returns control to the GS/OS Loader only when the I/O operation is complete or the user has canceled the request for the mount. For all user-callable GS/OS Loader functions, system preferences are controlled by the user. For the internal jump-table load function, the GS/OS Loaders set system preferences to display mount messages and then restore them to their original state.

The GS/OS Loaders and the Memory Manager

The GS/OS Loaders and the Memory Manager work together closely. Depending on how a GS/OS Loader defines a segment, the Memory Manager needs to allocate a memory block for that segment with the appropriate properties.

The GS/OS Loaders define load segments as static or dynamic and as absolute (must be loaded at a specific address), relocatable (can be loaded at any address, but cannot be moved once loaded), or position-independent (can be loaded anywhere and then moved anywhere after loading). The Memory Manager uses its own terminology to describe memory blocks; see the "Memory Manager" chapter in the *Apple IIGS Toolbox Reference.* Loader and Memory Manager terminology are related in this way:

- When a GS/OS Loader loads a static segment, it calls the Memory Manager to allocate a corresponding memory block that is unpurgeable (purge level = 0; the Memory Manager cannot remove it from memory) and locked (the Memory Manager cannot move it unless it is first unlocked).
- When the loader loads a dynamic segment, the Memory Manager allocates a memory block that is marked as purgeable (purge level > 0) but locked.
- When the loader loads a position-independent segment, the Memory Manager allocates a memory block that is marked as movable (the Memory Manager can change their locations in memory if they are not locked); all other segments (whether static or dynamic) are placed in blocks that are fixed (not movable, even if not locked, and of fixed size).

The typical load segment, which is relocatable, is loaded into a memory block having these attributes:

Locked
Fixed
Purge level = 0 (if static)
Purge level = 3 (if dynamic)

When a GS/OS Loader *unloads* a segment, it calls the Memory Manager to make the corresponding memory blocks purgeable.

To unload *all* of a program's segments (all segments associated with a particular user ID), a controlling program calls the GS/OS Loader's UserShutdown routine—which in turn calls the Memory Manager—to purge all the program's dynamic segments and make all its static segments purgeable. The purpose of this is to keep the essential parts of an application in memory, in case it needs to be rerun in the near future. Keeping programs dormant in memory, and executing them again with the GS/OS Loader's Restart routine, can greatly speed up execution of a program selector such as the Finder. However, once the Memory Manager has actually purged one of the static segments of a dormant program, the program is incomplete and must be reloaded from file (with InitialLoad) before running.

◆ *Note:* If many incomplete (partially purged) applications are in memory, the system may get bogged down with NIL memory handles. To avoid this situation, the GS/OS Loaders dispose of all NIL memory handles they know about before executing every InitialLoad or Restart call.

Depending on the `ORG`, `KIND`, `BANKSIZE`, and `ALIGN` fields in the segment header (see "OMF and the GS/OS Loaders," later in this chapter), other memory-block attributes are possible, as shown in Table 8-1.

△ **Important** Although the Memory Manager does not provide bank alignment, the GS/OS Loaders have a memory-block attribute that forces alignment by requesting successive fixed-address blocks at the beginning of each bank until successful. However, having a bank-aligned load segment in a load file almost always causes everything purgeable to be purged. Carefully consider the advantages and disadvantages of bank-aligned segments before including one in a load file. For more information, see *Apple IIGS Toolbox Reference,* Volume 3, and Apple IIGS Technical Note #78. △

A memory block can be made purgeable (unloaded) by a call to a GS/OS Loader. However, other memory-block attributes must be changed through Memory Manager calls. The following memory block information may also be useful to a program:

Start location
Size of segment
User ID

Purge level:	0 = Unpurgeable
	1 = Least purgeable
	3 = Most purgeable

Note also that if the memory handle is NIL (its address value is 0), the memory block has been purged.

■ **Table 8-1** Segment characteristics and memory-block attributes

Segment header attribute	Memory-block attribute
If ORG > 0	Fixed address
If BANKSIZE = $10000	May not cross bank boundary
If 0<Align Factor* <= $100	Page aligned
If Align Factor* > $100	Bank aligned (forced by GS/OS Loaders; see preceding important information)
Bit 13 of KIND = 0	Fixed block (not movable)
Bit 12 of KIND = 1	May not use special memory
Bit 11 of KIND = 1	Fixed bank (not fixed address)
Bit 8 of KIND = 1	Bank-relative (fixed address in any bank); forced by GS/OS Loaders
KIND = 12	Fixed bank (bank $00), page aligned (Direct-page/stack segment)

*If 0 < BANKSIZE < $10000, Align factor = the greater of BANKSIZE or ALIGN; if BANKSIZE has any other value (except for $10000), Align factor = ALIGN.

OMF and the GS/OS Loaders

Object module format (OMF) defines the internal format for Apple IIGS object files, library files, and load files. OMF files consist of segments, each of which has a segment header and a series of OMF records. As Table 8-1 shows, a load segment's characteristics, the type of memory block it inhabits, and its segment header values are all closely related. OMF is documented in detail in Appendix F of this manual.

Object module format includes general capabilities beyond the requirements of the Apple IIGS computer. The GS/OS Loaders, on the other hand, are designed specifically for the Apple IIGS. Therefore, there are certain OMF features that the GS/OS Loaders either do not support or support in a restricted manner. For example,

- The NUMSEX field of the segment header must be 0.
- The NUMLEN field of the segment header must be 4.
- The BANKSIZE field of the segment header must be <= $10000.
- The ALIGN field of the segment header must be <= $10000.

If any of the above is not true, the GS/OS Loaders return error $110B (segment is foreign). The BANKSIZE and ALIGN restrictions are enforced by the linker, and violations of them are unlikely in a load file.

The GS/OS Loaders use BANKSIZE and ALIGN to force memory alignment of segments. Under OMF, ALIGN and BANKSIZE can be any power of 2. But the Memory Manager does not support so general a requirement. Currently, the Memory Manager can only be told that a memory block must be page-aligned or must not cross a bank boundary. To force bank alignment where needed, the GS/OS Loaders use this method:

- Any value of BANKSIZE other than 0 and $10000 results in a memory block that is either page aligned (if BANKSIZE < =$100) or bank aligned (if BANKSIZE > $100). Since the linker makes sure that the segment is smaller than BANKSIZE, the requirement that the segment not extend past the BANKSIZE boundary is met (there will be wasted space in the memory block, however).
- Any value of ALIGN is bumped to either page alignment or bank alignment.
- If the value of BANKSIZE is other than 0 or $10000 and the value of ALIGN is not 0, the greater of the two determines the alignment to be used.

Restarting, reloading, and dormant programs

By working closely with the Memory Manager and GS/OS, the GS/OS Loaders provide a mechanism whereby programs can stay in memory after they terminate and can be relaunched very quickly if they are called again.

When making the GS/OS Quit call, an application always specifies (1) whether it is capable of being relaunched from memory, and (2) whether it wishes to quit to another specific application, and—if so—whether it wants to be relaunched after that application quits. GS/OS notes those specifications and treats a quitting program accordingly.

- If a quitting application is capable of being restarted from memory—that is, if it does not require initialization data to be loaded from disk—GS/OS puts it into a dormant state with the GS/OS Loader's UserShutdown call: it keeps all the application's static segments in memory so that the application can start up very quickly if it is ever called again. When that application is relaunched from memory, it is said to be restarted. GS/OS uses the GS/OS Loader's Restart call for this.
- If an application will be relaunched at a future time, the GS/OS Loaders keep track of its pathname, so that when the time comes it can be reloaded—loaded and executed automatically from disk, using the GS/OS Loader's InitialLoad (or InitialLoad2) call. Of course, if the program is already in memory in a dormant state, it can simply be restarted.

A dormant application's static segments are not protected; if the Memory Manager needs memory, it can purge one or more of them. Once that happens, the application is no longer dormant; it must be reloaded from disk if it is ever relaunched.

◆ *Note:* In some programming languages it is impractical to make completely restartable applications; initialization data must be read from disk every time a program is launched. To permit restartability in such cases, the GS/OS Loaders allow for reload segments, load segments that are always loaded from disk at program launch, even if the program is in a dormant state. Therefore, if a program can be designed with all its initialization information in one or more reload segments, it can call itself restartable when it quits.

The GS/OS Loaders: ExpressLoad and the System Loader

ExpressLoad complements the standard Apple IIGS System Loader by allowing large applications to be loaded in a shorter time.

ExpressLoad works as a front end to the System Loader by automatically determining if the file being loaded is in ExpressLoad format. If the file is in ExpressLoad format, then ExpressLoad processes the file without passing control to the System Loader. If however, the file is not in ExpressLoad format, control is passed to the System Loader, where the file will be processed as normal.

◆ *Note:* A file in ExpressLoad format can still be loaded by the System Loader.

With one exception, the ExpressLoad call set is the same as the standard System Loader's. The exception is the GetLoadSegInfo call. This call is not implemented by ExpressLoad, since the internal data structures used by the standard System Loader are not the same as the data structures used by ExpressLoad.

ExpressLoad does not support the load-from-memory operations that can be specified as part of the InitialLoad2 call. The load_from_memory indication is passed to the standard System Loader.

The standard System Loader processes files that are stored in OMF. ExpressLoad processes files that are stored in OMF and contain a special segment named *ExpressLoad.* Two ways to build an ExpressLoad file are

- to use the APW tool Express or the MPW IIGS tool ExpressIIGS to convert an OMF 2.0 file into ExpressLoad format
- to use the LinkIIGS tool to make an ExpressLoad file directly

△ **Important** Only version 2.0 of OMF can be converted to ExpressLoad format. △

Making GS/OS Loader calls

Because the GS/OS Loaders are an Apple IIGS tool set, their functions are called by making stack-based calls through the Apple IIGS Tool Locator. The calling sequence for GS/OS Loader functions is the standard tool-calling sequence:

1. Push space for the output parameters (if any) onto the stack.
2. Push all input parameters *in the order specified in the call descriptions.*
3. Execute this call block (the syntax in this example is for APW™):

   ```
   ldx    #$11+FuncNum|8
   jsl    Dispatcher
   ```

   where `FuncNum` is the GS/OS Loader function number (the number of the call), `$11` is the tool number for the GS/OS Loaders, and `Dispatcher` is the Tool Locator entry point.
4. Upon return from the call, the A register contains the call status (zero if no error, an error number otherwise), and the carry flag is set if an error has occurred.
5. If there is output, pull each output parameter off the stack.

Table 8-2 lists and briefly describes the GS/OS Loader calls available to applications (plus its standard tool-set calls, some of which are not available to applications). The calls in Table 8-2 are in numerical order by call number, except that newer calls that use GS/OS-specific data structures (such as InitialLoad2) are listed next to their ProDOS 16–compatible counterparts (such as InitialLoad).

The rest of this chapter consists of detailed call descriptions; they are presented in alphabetical order by call name.

■ **Table 8-2** GS/OS Loader calls

Call number	Call name	Description
$0111	LoaderInitialization	Initializes the loader
$0211	LoaderStartup	(Does nothing)
$0311	LoaderShutDown	(Does nothing)
$0411	LoaderVersion	Returns loader version
$0511	LoaderReset	(Does nothing)
$0611	LoaderStatus	Returns loader status
$0911	InitialLoad	Loads a program into memory
$2011	InitialLoad2	Loads a program into memory
$0A11	Restart	Reexecutes a dormant program in memory
$0B11	LoadSegNum	(Load segment by number) Loads a single segment
$0C11	UnloadSegNum	(Unload segment by number) Unloads a single segment
$0D11	LoadSegName	(Load segment by name) Loads a single segment
$0E11	UnloadSeg	Unloads the segment containing a specific address
$0F11	GetLoadSegInfo	Returns a segment's memory-segment table entry (except for files compiled into ExpressLoad format)
$1011	GetUserID	Returns the user ID for a given pathname
$2111	GetUserID2	Returns the user ID for a given pathname
$1111	LGetPathname	Returns the pathname for a given user ID
$2211	LGetPathname2	Returns the pathname for a given user ID
$1211	UserShutDown	Shuts down a program
$1311	RenamePathname	Renames a pathname

$0F11 GetLoadSegInfo

Description

This function returns the memory-segment-table entry corresponding to the specified load segment. The memory-segment table is searched for the specified entry; if the entry is not found, error $1101 is returned. If the entry is found, the contents (except for link pointers to other entries) are moved into the user buffer.

△ **Important** ExpressLoad does not support this function; do not use this function if you are going to compile your program in ExpressLoad format. △

Parameters

Stack before call

Previous contents	
userID	Word—User ID of the load segment
fileNum	Word—Load-file number
segNum	Word—Load-segment number
– *buffAddr* –	Longword—User buffer address
	<—SP

Stack after call

Previous contents	
	<—SP

Notes The output of this call is a filled user buffer.

Errors

$1101	Entry not found

$1011 GetUserID

Description

This function searches the pathname table for the specified pathname. The input pathname is a standard Pascal-type string (a byte count followed by the string of characters). The pathname is first expanded to a full pathname (in GS/OS string format) before the search. If a match is found, the corresponding user ID is returned. A controlling program can use this function to determine whether to perform a Restart or an InitialLoad call on an application.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
pathnameAddr	Longword—User buffer address
	<—SP

Stack after call

Previous contents	
userID	Word—Corresponding user ID
	<—SP

Errors

$1101	Entry not found

$2111 GetUserID2

Description This function is identical to GetUserID except that the input pathname is a GS/OS string rather than a Pascal string.

Parameters Stack before call

Previous contents	
Space	Word—Space for result
– *pathnameAddr* –	Longword—User buffer address
	<—SP

Stack after call

Previous contents	
userID	Word—Corresponding user ID
	<—SP

Errors

$1101	Entry not found

$0911 InitialLoad

Description

A controlling program (such as GS/OS or a shell program) uses this call to load another program into memory, in preparation for executing it.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
Space	Word—Space for result
Space	Longword—Space for result
Space	Word—Space for result
userID	Word—The user ID to be assigned
pathnameAddr	Longword—Address of the load file's pathname
flagWord	Word—`don't-use-special-memory` flag
	<—SP

Stack after call

Previous contents	
buffSize	Word—Size of direct-page/stack buffer
dPageAddr	Word—Address of direct-page/stack buffer
startAddr	Longword—Starting address of the program
userID	Word—The user ID assigned
	<—SP

Notes

If a complete user ID is specified, the GS/OS Loaders use that when allocating memory for the load segments. If the `mainID` portion of the user ID is 0, a new user ID is obtained from the User ID Manager, based on the `typeID` portion of the user ID. If the Type portion is 0, an Application type user ID is requested from the User ID Manager. User IDs are explained under "Miscellaneous Tools" in the *Apple IIGS Toolbox Reference.*

If the don't-use-special-memory flag is TRUE (nonzero), the GS/OS Loaders do not load any static load segments into **special memory.** (Special memory is the part of memory equivalent to that used by a standard Apple II computer under ProDOS 8: all of banks $00 and $01 and parts of banks $E0 and $E1.) However, dynamic load segments are loaded into any available memory, regardless of the state of the don't-use-special-memory flag.

GS/OS is called to open the specified load file using the input pathname. Note that the input pathname is a Pascal string. If any GS/OS errors occur or if the file is not a load file type ($B3–$BE), the GS/OS Loaders return the appropriate error message.

If the load file is successfully opened, the GS/OS Loaders add the load file information to the pathname table and call the LoadSegNum function for each static load segment in the load file.

If an initialization segment (KIND = $10) is loaded, the GS/OS Loaders immediately transfer control to that segment in memory. When the GS/OS Loaders regain control, the rest of the static segments are loaded normally.

If the direct-page/stack segment (KIND = $12) is loaded, its starting address and length are returned as output.

If any of the static segments cannot be loaded, the GS/OS Loaders abort the load and return an error.

After all the static load segments have been loaded, execution returns to the controlling program with the starting address of the first load segment (not an initialization segment) of the load file. Note that the controlling program is responsible for setting up the stack pointer and direct register, and for actually transferring control to the loaded program.

Errors

$1102	OMF version error
$1104	File is not load file
$1109	Segment number out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign

$2011 InitialLoad2

Description

This function is similar to InitialLoad except that four variations of the input information are possible.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
Space	Word—Space for result
Space	Longword—Space for result
Space	Word—Space for result
userID	Word—The user ID to be assigned
buffAddr	Longword—Input address or parameter block
flagWord	Word—`don't-use-special-memory` flag
inputType	Word—Input type
	<—SP

Stack after call

Previous contents	
buffSize	Word—Size of direct-page/stack buffer
dPageAddr	Word—Address of direct-page/stack buffer
startAddr	Longword—Starting address of the program
userID	Word—The user ID assigned
	<—SP

Notes

If `inputType` = 0, this function is exactly equivalent to the InitialLoad call.

If `inputType` = 1, the input load-file pathname is a GS/OS string rather than a Pascal string.

If `inputType` = 2 and the System Loader is being used, the input address points to a parameter block that contains two parameters: `memoryAddress` (4 bytes) and `fileLength` (2 bytes). The memoryAddress parameter specifies where a load file resides in memory and the fileLength parameter specifies its size in bytes. In this case, the System Loader loads the file from memory rather than from a file.

△ **Important** ExpressLoad does not support input type 2. △

This input type is used by GS/OS at system startup to load any load files that were previously read into memory as binary images. In this mode, the GS/OS Loaders do not make any GS/OS calls and can therefore be used when GS/OS is not in memory or has not yet been initialized.

If inputType = 3, the input address points to an entry in the pathname table. The pathname, user ID, and file number from the pathname table entry are used as input for InitialLoad. The jump-table load function (an internal function) uses this entry to load all the static segments in a run-time library.

If inputType = 4, the input address points to a parameter block that contains two parameters: memoryAddress (4 bytes) and fileLength (2 bytes).

△ **Important** ExpressLoad does not support input type 4. △

The memoryAddress parameter specifies where a resource code file resides in memory and the fileLength parameter specifies its size in bytes. In this case, the System Loader loads the file from memory rather than from a file, performing exactly the same function as a Memory Load (inputType = 2) except that all the information about the load segments is purged from the System Loader's tables. The Resource Manager uses this input type to relocate code resources that have been read into memory.

Errors

$1102	OMF version error
$1104	File is not load file
$1109	Segment number out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign

$1111 LGetPathname

Description

This function searches the pathname table for the specified user ID and file number. If a match is found, the address of the pathname in the pathname table is returned. The output pathname is a Pascal string.

GS/OS uses this call to get the pathname of an existing application so that it can set the application prefix before restarting it. Note that the output address is a pointer to a string within a GS/OS Loader internal data structure, and nothing should be written to that address or the following addresses. If you need to retain the pointer, use LGetPathname2.

Parameters

Stack before call

Previous contents	
Space	Longword—Space for result
userID	Word—The user ID to find
fileNum	Word—The file number to find
	<—SP

Stack after call

Previous contents	
pathnameAddr	Longword—Address of pathname (if found)
	<—SP

Errors

$1101	Entry not found
$1103	Pathname error

$2211 LGetPathname2

Description

This function is identical to LGetPathname except that the output pathname is a GS/OS string rather than a Pascal string.

Parameters

Stack before call

Previous contents	
Space	Longword—Space for result
userID	Word—The user ID to find
fileNum	Word—The file number to find
	<—SP

Stack after call

Previous contents	
pathnameAddr	Longword—Address of pathname (if found)
	<—SP

Errors

$1101	Entry not found
$1103	Pathname error

$0111 LoaderInitialization

Description

This routine initializes the GS/OS Loaders. It is called at system initialization time only. All GS/OS Loader tables are cleared, and no assumptions are made about the current or previous state of the system.

Parameters

There is no input or output to this call.

Errors

None

$0511 LoaderReset

Description This routine does nothing and need not be called.

Parameters There is no input or output to this call.

Errors None

$0311 LoaderShutDown

Description This routine does nothing and need not be called.

Parameters There is no input or output to this call.

Errors None

$0211 LoaderStartup

Description This routine does nothing and need not be called.

Parameters There is no input or output to this call.

Errors None

$0611 LoaderStatus

Description This routine returns the status (initialized or not initialized) of the GS/OS Loaders. It always returns TRUE because the GS/OS Loaders are always in the initialized state.

Parameters Stack before call

Previous contents	
Space	Word—Space for result
	<—SP

Stack after call

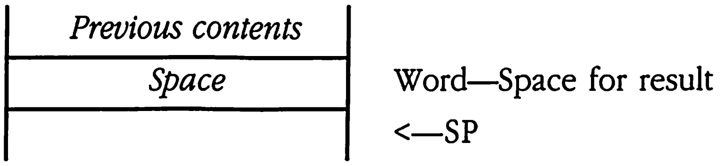
Previous contents	
status	Word—Current GS/OS Loader status; always TRUE
	<—SP

Errors None

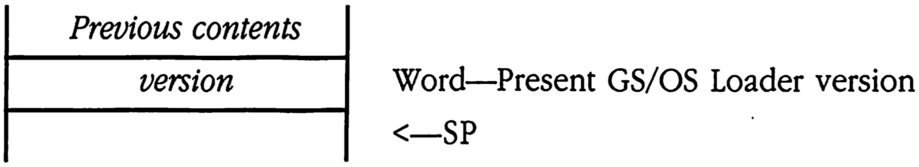
$0411 LoaderVersion

Description This routine returns the version number of the GS/OS Loaders. The version number is in the same format as that returned by the GS/OS call GetVersion.

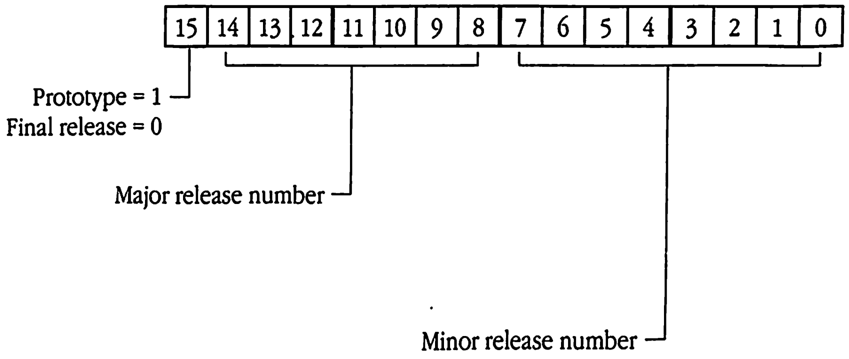
Parameters Stack before call



Stack after call



Notes This is the format of the version word returned by this call:



Errors None

$0D11 LoadSegName (Load Segment by Name)

Description This function loads a named load segment into memory.

Parameters Stack before call

Previous contents	
Space	Word—Space for result
Space	Word—Space for result
Space	Word—Space for result
– *Space* –	Longword—Space for result
userID	Word—The user ID of the caller
– *filenameAddr* –	Longword—The address of the load filename
– *segNameAddr* –	Longword—The address of the load-segment name
	<—SP

Stack after call

Previous contents	
segNum	Word—The load-segment number of the segment
fileNum	Word—The load-file number of the segment
userID	Word—The user ID assigned
– *segAddr* –	Longword—The starting address of the segment
	<—SP

Notes The input pathname is a Pascal string. The loader calls GS/OS to open the specified load file. If GS/OS has a problem, a GS/OS error code is returned. If the file is not a load file (types $B3–$BE), error $1104 is returned.

Next, the load file is searched for a load segment corresponding to the specified load-segment name. If no segment has the segment name requested, error $1101 is returned.

Once a GS/OS Loader has located the requested load segment (and knows its load-segment number), it checks the pathname table to see whether the load file is represented. If so, it uses the file number from the table. Otherwise, the GS/OS Loaders add a new entry to the pathname table with an unused file number. If necessary, the GS/OS Loaders load the jump-table segment (if any) from the load file.

Next, the GS/OS Loaders attempt to load the load segment by calling the LoadSegNum function. If LoadSegNum returns an error, then LoadSegName returns the error. If LoadSegNum is successful, LoadSegName returns the load-file number, the load-segment number, and the starting address of the segment in memory.

Errors

$1101	Segment not found
$1104	File is not load file
$1107	File version error
$1109	Segment number out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign

$0B11 LoadSegNum (Load Segment by Number)

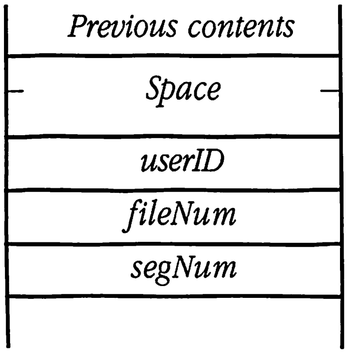
Description

This function loads a specific load segment into memory. This is the workhorse function of the GS/OS Loaders. Normally, a program calls this function to load a dynamic load segment manually. If a program calls this function to load a static load segment, the GS/OS Loaders do not patch any existing references to the newly loaded segment.

> △ **Important** Do not use this function if you want ExpressLoad to be used. Since ExpressLoad files may have their segments rearranged, if an ExpressLoad file is loaded by the System Loader, references to segments by number may be incorrect. Use the LoadSegName function instead. △

Parameters

Stack before call



Longword—Space for result

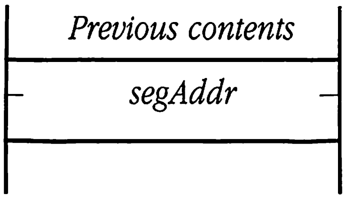
Word—The user of the caller

Word—The load-file number of the segment

Word—The load-segment number of the segment

<—SP

Stack after call



Longword—The starting address of the segment

<—SP

Notes

First, the memory-segment table is searched to see if there is an entry for the requested load segment. If there is already an entry, the handle to the memory block is checked to verify that it is still in memory. If the block is still in memory, this function does nothing further and returns without an error. If the memory block has been purged, the memory-segment table entry is deleted.

Next, the load-file number is looked up in the pathname table to get the load-file pathname. From the file's directory entry, the load-file type is checked; if it is not a load file (types $B3–$BE), error $1104 is returned. The load file's modification date and time values are compared to the file date and file time values in the pathname table. If these values do not match, error $1107 is returned. This indicates that the run-time library file at the specified pathname is not the run-time library file that was scanned when the application was linked together.

The GS/OS Loaders then call GS/OS to open the specified load file. If GS/OS has a problem, a GS/OS error code is returned.

The load file is then searched for a load segment corresponding to the specified load-segment number. If there is no segment corresponding to the load-segment number, error $1101 is returned. If the VERSION field of the segment header contains a value that is not supported by the GS/OS Loaders, error $1102 is returned. If the SEGNUM field does not correspond to the load-segment number, error $1109 is returned. If the NUMSEX and NUMLEN fields are not 0 and 4, respectively, error $110B is returned.

If the load segment is found and its segment header is correct, a memory block is requested from the Memory Manager of the size specified in the LENGTH field in the segment header. If the ORG field in the segment header is not 0, a memory block starting at that address is requested. Other attributes are set according to segment header fields (see "The GS/OS Loaders and the Memory Manager," earlier in this chapter).

If the input user ID is not 0, it is used as the user ID of the memory block.

△ **Important** Because of the possibility of conflict between user IDs, do not use 0 as an input user ID. △

If the requested memory is not available, the Memory Manager and the GS/OS Loaders try several techniques to free up memory:

- The Memory Manager purges memory blocks that are purgeable.
- The Memory Manager moves movable segments to enlarge contiguous memory.
- The GS/OS Loaders call their Cleanup routines (an internal function) to free their own unused internal memory.

If all these techniques fail, the GS/OS Loaders return with the last Memory Manager error.

If enough memory is available, the GS/OS Loaders load the load segment into memory and process its **relocation dictionary,** the part of every relocatable segment that the loader uses to patch the code for correct execution at its current address. See Appendix F, "Object Module Format," for more information.

The loader adds a new entry to the memory-segment table and returns with the memory handle of the segment's memory block.

OMF records

Only the following object module format records are supported by the GS/OS Loaders:

LCONST	($F2)
DS	($F1)
RELOC	($E2)
INTERSEG	($E3)
cRELOC	($F5)
cINTERSEG	($F6)
SUPER	($F7)
END	($00)

Any other records encountered while loading result in error $110A.

Errors

$1101	Segment not found
$1102	OMF version error
$1104	File is not load file
$1107	File version error
$1109	Segment number out of sequence
$110A	Illegal load record found
$110B	Segment is foreign

$1311 RenamePathname

Description

This routine searches the pathname table for a match of the pathname specifed by the `oldAddress` parameter, and then replaces the matched pathname with the new pathname specified by the `newAddress` parameter. The input pathnames must be type 1 strings.

The GS/OS Loaders call the GS/OS ExpandPath routine for each input pathname before it is used for string comparison and replacement. Thus, the input pathnames can be full or partial pathnames of volumes, subdirectories, or files.

GS/OS calls RenamePathname whenever it must change a pathname; thus, any files that the loaders are managing also have their pathnames changed.

Parameters

Stack before call

Previous contents	
oldAddress	Longword—The address of the old pathname
newAddress	Longword—The address of the new pathname
	<—SP

Stack after call

Previous contents	
	<—SP

Errors

None

$0A11 Restart

Description

A controlling program (such as GS/OS) uses this call to restart (relaunch) a dormant application in memory. Only software that is **restartable** can be successfully restarted. For a program to be restartable, it must initialize its variables and not assume that they will be preset at load time. A reload segment can be used for initializing data because it is reloaded from the file during a restart. The controlling program is responsible for knowing whether a given program can be restarted; the GS/OS Loaders do no checking.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
Space	Word—Space for result
Space	Longword—Space for result
Space	Word—Space for result
userID	Word—The user ID of the program to restart
	<—SP

Stack after call

Previous contents	
buffSize	Word—Size of direct-page/stack buffer
buffAddr	Word—Address of direct-page/stack buffer
startAddr	Longword—The starting address of the program
userID	Word—The user ID assigned
	<—SP

Notes

An existing user ID must be specified; otherwise, the GS/OS Loaders return error $1108. If the user ID is not known to the GS/OS Loaders, error $1101 is returned.

Applications can be restarted only if all the segments in the memory-segment table with the input user ID are in memory; these are the application's static segments. If all are there, the GS/OS Loaders resurrect the application from its dormant state by calling the Memory Manager to lock and make unpurgeable all its segments.

The Restart call returns the user ID and the starting address of the first segment, as well as the direct-page/stack information from the pathname table. After all the static segments are resurrected, the GS/OS Loaders look for initialization segments and reload segments; they execute the former and reload the latter.

If there is a pathname table entry for the user ID but not all the segments are in memory, the GS/OS Loaders first call UserShutdown, which purges the user ID from all its tables, and then perform an InitialLoad from the original load file.

Errors

$1101	Application not found
$1108	User ID error

$0E11 UnloadSeg (Unload Segment by Address)

Description

This function unloads the load segment that contains the specified input address.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
Space	Word—Space for result
Space	Word—Space for result
address	Longword—An address within segment to be unloaded
	<—SP

Stack after call

Previous contents	
segNum	Word—The load-segment number of the segment
fileNum	Word—The load-file number of the segment
userID	Word—The user ID of the segment
	<—SP

Notes

The GS/OS Loaders call the Memory Manager to locate the memory block containing the specified address. If no allocated memory block contains the address, error $1101 is returned. The user ID associated with the handle of the memory block returned by the Memory Manager is extracted, and the memory-segment table is scanned to find the user ID and handle. If an entry is not found, error $1101 is returned.

If the entry in the memory-segment table is for a jump-table segment, the specified address should be pointing to the jump-table entry for a dynamic segment reference. The load-file number and segment number of the jump-table entry are extracted. If the entry in the memory-segment table is not for a jump-table segment, the load-file number and segment number of the memory-segment table entry are extracted.

Errors

$1101	Segment not found

$0C11 UnloadSegNum (Unload Segment by Number)

Description

This function unloads a specified (by number) load segment that is currently in memory.

△ **Important** Do not use this function if you want ExpressLoad to be used. Since ExpressLoad files may have their segments rearranged, if an ExpressLoad file is loaded by the System Loader, references to segments by number may be incorrect. Use the UnLoadSeg function instead. △

Parameters

Stack before call

Previous contents	
userID	Word—The user of the segment to be unloaded
fileNum	Word—The load-file number of the segment
segNum	Word—The load-segment number of the segment
	<—SP

Stack after call

Previous contents	
	<—SP

Notes

The GS/OS Loaders search the memory-segment table for the input load-file number and load-segment number. If there is no such entry, error $1101 is returned.

Next, the Memory Manager is called to make the memory block purgeable, using the memory handle in the table entry.

All entries in the jump table referencing the unloaded segment are changed to their unloaded states.

If the input user ID is not 0, it is used as the user ID of the memory block.

△ **Important** Because of the possibility of conflict between user IDs, do not use 0 as an input user ID. △

If both the load-file number and the load-segment number are specified, the specific load segment is made purgeable whether it is static or dynamic. Note that if a static segment is unloaded, the application cannot be restarted. If either input is 0, only dynamic segments are made purgeable.

If the input load-segment number is 0, all dynamic segments in the specified load file are unloaded.

If the input load-file number is 0, all dynamic segments for the user ID are unloaded.

Errors	$1101	Segment not found

$1211 UserShutDown

Description

This function is called by the controlling program to close down an application that has just terminated. If the specified user ID is 0, the current user ID is assumed.

Parameters

Stack before call

Previous contents	
Space	Word—Space for result
userID	Word—The user ID of the program to shut down
flag	Word—The quit flag
	<—SP

Stack after call

Previous contents	
userID	Word—The user ID of the program that was shut down
	<—SP

Notes

The quit flag corresponds to the quit flag used in the GS/OS Quit call.

- If the quit flag is 0, all memory blocks for the user ID are discarded and all the GS/OS Loaders' internal tables are purged of the user ID. The application cannot be restarted. The user ID is also removed from the system so that it can be reused.
- If the quit flag is $8000, all memory blocks for the user ID are discarded and all the GS/OS Loaders' tables (except the pathname table) are purged of the user ID. The application can be reloaded (but not restarted), because its pathname is remembered.
- If the quit flag is any other value, the memory blocks associated with the specified user ID (with `auxID` cleared) are processed as follows:
  - □ All memory blocks corresponding to dynamic load segments are discarded.
  - □ All memory blocks corresponding to static load segments are made purgeable.
  - □ All other memory blocks are purged.

In addition, all dynamic segment entries in the memory-segment table and all entries in the jump-table directory for the specified user ID are removed. The application is now in a dormant state and can be restarted (resurrected by the GS/OS Loaders very quickly because all the static segments are still in memory). However, as soon as any static segment is purged by the Memory Manager for any reason, the GS/OS Loaders must reload the application from its original load file.

Errors

$1101	User ID not found

Chapter 9 **Using the Console Driver**

The **console** is a conceptual component of a computer system; it consists of the principal conduits by which the user sends commands to the computer and receives messages from the computer. On the Apple IIGS, like most personal computers, the console consists of the keyboard (for input) and the video display screen (for output).

The GS/OS console driver is a loaded driver that allows sophisticated manipulation of the Apple IIGS text page. It runs in both 40-column and 80-column mode. The console driver supports many advanced features, while using the standard Apple II BASIC and Pascal control codes.

◆ *Note:* The console driver is for use only by applications that run in text mode. The console driver does not support the standard Apple II Hi-Res or double Hi-Res graphics. If your application uses the Apple IIGS Super Hi-Res graphics screen, it writes to the screen with toolbox calls. See the *Apple IIGS Toolbox Reference.*

General information

The GS/OS console driver allows an application to treat both parts of the console (keyboard and screen) as a single device that can be read from or written to. Because the console has two parts, the console driver does also: an input routine and an output routine, shown in Figure 9-1.

Console output

The Console Output routine writes to the screen. It supports uppercase, lowercase, inverse, and MouseText characters. It also includes a suite of control characters with functions such as any-direction scrolling, character-set selection, and cursor control. Finally, it permits areas of the screen to be saved to off-screen buffers, and text port parameters to be selectively saved and restored—in effect, allowing a simple windowing system.

All commands to the Console Output routine are sent as control characters. This allows the programmer to create strings of commands that will be executed one after another, but that require only a single write call. All operations occur in a rectangular subset of the hardware screen known as the text port. All text outside the text port is protected; that is, it will not be affected by any console calls.

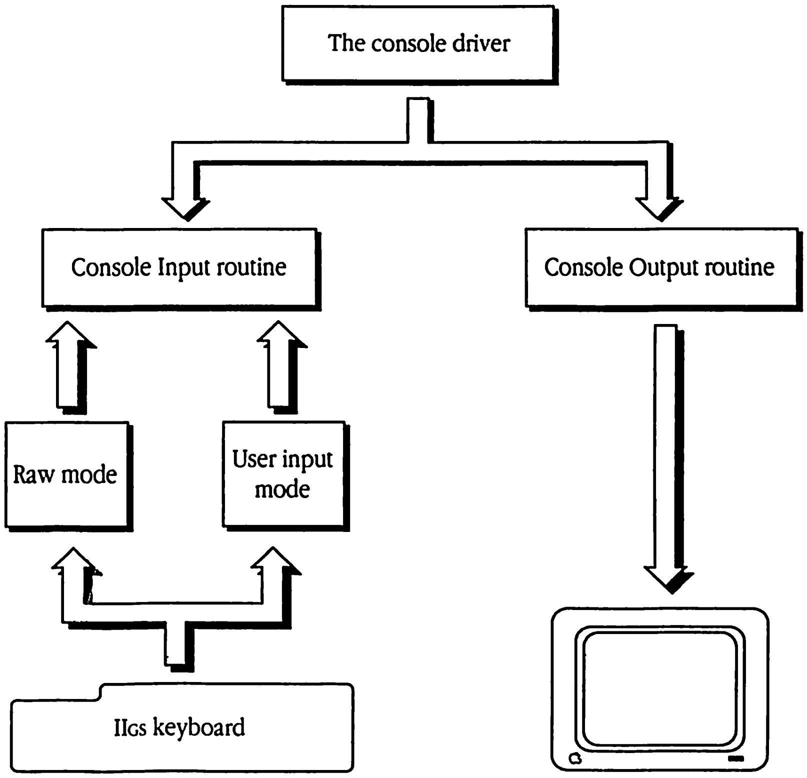
Console input

The Console Input routine accepts characters from the keyboard. There are two basic input modes: raw mode allows for simple keyboard input, whereas the more advanced user-input mode allows for text-line editing and application-defined terminator keys. User-input mode also supports features such as the interrupt key, which allows application-defined editing keystrokes such as the arrow keys to change a setting or using a key combination to bring up a help screen.

The application can supply a default string to the user input mode. If the default string contains more characters than the width of the input field, the extra characters are retained; however, they are displayed only if characters are deleted from the visible part of the field. Horizontal scrolling of the input field is not supported.

The application can also specify options such as overstrike or insert mode on entry. A blinking block cursor signifies overstrike mode; a blinking underline cursor specifies insertion. The cursor blinking rate is based on the current control panel settings.

■ **Figure 9-1** Console driver I/O routines



The user can insert control characters into the input string by pressing Command-Control-*character,* where *character* is replaced by any keyboard character.

◆ *Note:* The Command key is the key historically known as the *Apple* key or as the *Open-Apple* key.

Control characters are highlighted on the screen in inverse video, but are returned in the input string as codes from $00 to $1F. All normal ASCII characters are returned in the range $20–$7F.

The terminators used by the Console Input routine are more advanced than the newline characters specified in GS/OS (see the description of the NewLine call in Chapter 7). User-specified terminators can include not only ASCII codes for the terminator characters, but also the keyboard modifier bits. For example, the Return and Enter keys can be given different functions by separately specifying terminators, one with the keypad flag set and one with it clear.

The Console Output routine

The Console Output routine handles writing to the screen. It supports different screen sizes and defines areas of the screen called *text ports,* which can be used to protect parts of the screen. All commands to the Console Output routine are sent as control characters.

Screen size

The default screen size (in columns of width) is always 80 columns. You can change the screen size by writing the correct screen control code, described in the section "Screen Control Codes" later in this chapter.

The 40-column screen consists of 40 columns of text in 24 lines. The upper-left corner is 0,0 and the lower-right corner is 39,23.

The 80-column screen consists of 80 columns of text in 24 lines. The upper-left corner is 0,0 and the lower-right corner is 79,23.

The text port

The driver maintains an active text port in which all activity occurs. The default size of this text port is the entire screen. However, subsequent calls can be made to resize the port. All text outside the text port is protected—no console driver calls can affect that text.

Two control commands allow the application to save the current text port's definitions, start with a new port, and then retrieve the original port. This allows a simple windowing system. In addition, driver-specific control calls allow the application to read the text port data structure; however, the values in the data structure can be changed only with control commands (see the section "Screen Control Codes" later in this chapter).

This is the structure of the text port record:

```
TextPortRec = {
      byte  ch,
            cv,
            windLeft,
            windTop,
            windRight,
            windBottom,
            windWidth,
            windLength,
            consWrap,
            consAdvance,
            consLF,
            consScroll,
            consVideo,
            consDLE,
            consMouse,
            consFill    }
```

Here are the definitions for the fields:

`ch`
`cv`

The current location of the cursor (horizontal and vertical, from the upper-left corner). The cursor is always within the current text port, but is expressed in absolute screen coordinates.
Default = 0,0

`windLeft`
`windTop`
`windRight`
`windBottom`

Boundaries of the current text port, in absolute screen coordinates.
`windTop` must be <= `windBottom`, and
`windLeft` must be <= `windRight`.
Default = full hardware screen

`windWidth`
`windLength`

Size of the current text port, calculated as follows:
`windWidth` = `windLeft` – `windRight` + 1
`windLength` = `windTop` – `windBottom` + 1
Default = full hardware screen

`consWrap`

A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, the cursor wraps to the first column of the next line after printing in the rightmost column.
Default = TRUE

`consAdvance`

A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, the cursor moves one space to the right after printing.
Default = TRUE

`consLF` A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. Carriage return characters always move the cursor to the first column of the text port. If `consLF` is TRUE, the cursor will also move to the next line (note that this could cause a scroll—see next flag).
Default = TRUE

`consScroll` A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, the screen will scroll if the cursor is moved past the top or bottom of the screen.
Default = TRUE

`consVideo` A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, output is displayed in normal video. If FALSE, output is displayed in inverse video.
Default = TRUE

`consDLE` A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, character $10 (DLE) is interpreted as a space expansion character; when it is encountered in the input stream, the ASCII value of the next character minus 32 becomes the number of spaces to output.
Default = TRUE

`consMouse` A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, MouseText is turned on. When MouseText is on, inverse uppercase characters are displayed as MouseText.
Default = FALSE

`consFill` This is the fill character used for clearing areas of the screen. It is an actual screen byte—the value of the character as stored in memory—so the high-order bit must be turned on for normal display. For example, spaces (ASCII $20) should be specified by $A0. The value in this field is altered whenever inverse mode is changed to normal or vice versa.
Default = $A0 (screen space)

Character set mapping

Output characters go through a number of stages before they are placed in screen memory and appear on the screen. The console driver always uses the Apple IIGS alternate character set, which includes uppercase and lowercase characters, punctuation, numbers, inverse characters, and MouseText characters.

Normally, the console driver accepts input in the standard, 7-bit ASCII range (00–$7F). This input is then mapped to the screen based on the current display mode and MouseText mode settings (turned on or off through control codes in the character stream; see the section "Screen Control Codes" later in this chapter). In addition, input ASCII in the range $80–$FF is mapped directly to the inverse of whatever the current display mode is. Thus screen bytes (characters stored in screen memory) may have values quite different from their original input ASCII values.

Tables 9-1 and 9-2 summarize the output mapping for both normal and inverse display modes with MouseText mapping both disabled and enabled. The tables compare input ASCII values with the characters displayed on the screen and with the equivalent values stored in screen memory. Table 9-3 shows that setting the high-order bit (special direct inverse mode) is a shortcut to getting the inverse of the current mode.

△ **Important** Tables 9-1 through 9-3 show that in some cases sequential ASCII values in the input stream (such as $3F and $40) may map to nonsequential values in screen memory (such as $BF and $80, respectively). Specifically, the range of values interpreted as uppercase characters may not be continuous with the ranges interpreted as special characters and lowercase characters. If your application retrieves bytes directly from screen memory, it may have to compensate for this. △

■ **Table 9-1** Console driver character mapping—MouseText disabled

	Normal display mode		**Inverse display mode**	
Input values	**As displayed**	**As stored**	**As displayed**	**As stored**
$00–1F	Control characters	n/a	Control characters	n/a
$20–3F	Special characters	$A0–BF	Inverse special	$20–3F
$40–5F	Uppercase letters	$80–9F	Inverse upper	$00–1F
$60–7F	Lowercase letters	$E0–FF	Inverse lowercase	$60–7F

■ **Table 9-2** Console driver character mapping—MouseText enabled

	Normal display mode		**Inverse display mode**	
Input values	**As displayed**	**As stored**	**As displayed**	**As stored**
00–1F	Control characters	n/a	Control characters	n/a
20–3F	Special characters	A0–BF	Inverse special	20–3F
40–5F	Uppercase letters	80–9F	MouseText characters	40–5F
60–7F	Lowercase letters	E0–FF	Inverse lowercase	60–7F

■ **Table 9-3** Console driver character mapping—special direct inverse mode

	Normal display mode		Inverse display mode	
Input values	**As displayed**	**As stored**	**As displayed**	**As stored**
80–9F	Uppercase inverse	00–1F	Uppercase normal	80–9F
A0–BF	Inverse special	20–3F	Special characters normal mode	A0–BF
C0–DF	MouseText characters	40–5F	Uppercase normal	C0–DF
E0–FF	Inverse lower	60–7F	Lowercase normal	E0–FF

Screen control codes

In any mode, values from $00 to $1F are interpreted as control codes. Some control codes are one-byte commands; others use from two to four bytes of operands, which follow the control character. If an output stream ends in the middle of a multibyte sequence, the console driver simply uses the first bytes of the next output stream. The actual command is not executed until the entire command string has been read. Here are the defined control codes:

$00 NULL

No operation is performed.

$01 Save Current Text Port and Reset Default Text Port

Saves the current text port and resets to the default text port. If the system is out of memory, no error is returned, and the text port is simply reset.

$02 Set Text Port Size

Accepts the next four bytes as absolute screen coordinates + 32. Sets the current text port to the new parameters. The parameters are in the following order: `windLeft`, `windTop`, `windRight`, `windBottom`. Any parameter outside the screen boundaries is clipped to a legal value. The cursor is set to the upper-left corner of the new text port.

$03 Clear from Beginning of Line

Clears all characters from the left edge to and including the cursor. Sets them to the current `consFill` character.

$04 Pop Text Port

Restores the text port to the most recently saved value (see code $01). If no saved ports exist, resets the text port to the default values. If an 80-column text port is pushed and subsequently restored in 40-column mode, the text port may be altered to fit in the 40-column screen (see code $11, Set 40-Column Mode).

$05 Horizontal Scroll

Interprets the next byte as an 8-bit signed integer depicting the number (*N*) of columns to shift. *N* equal to zero is a null operation. If *N* is less than zero, the text port is shifted to the left; *N* greater than zero shifts to the right. If the shift magnitude is equal to or greater than `windWidth`, the text port is cleared.

The shifted characters are moved directly to their destination location. The space vacated by the shifted characters is set to the current `consFill` character (see the description of `consFill` earlier in this chapter). Characters shifted out of the text port are removed from the screen and are not recoverable.

$06 Set Vertical Position

Interprets the next byte as a text port–relative vertical position + 32. If the destination is outside the current text port, the cursor is moved to the nearest edge.

$07 Ring Bell

Causes the System Beep to be played. It has no effect on the screen.

$08 Backspace

Moves the cursor one position to the left. If the cursor was on the left edge of the text port and `consWrap` is TRUE, the cursor is placed one row higher and at the right edge. If the cursor was also on the topmost row and `consScroll` is TRUE, the text port will scroll backward one line.

$09 Tab (no operation)

This command is ignored.

$0A Line Feed

Causes the cursor to move down one line. If the cursor was at the bottom edge of the text port and `consScroll` is TRUE, the text port scrolls up one line.

$0B Clear to End of Text Port

Clears all characters from the cursor to the end of the current text port and sets them to be equal to the current `consFill` character.

$0C Clear Text Port and Home Cursor

Clears the entire text port and resets the cursor to `windLeft`, `windTop`.

$0D Carriage Return

Resets the cursor to the left edge of the text port; if `consLF` is TRUE, performs a line feed (see $0A, Line Feed).

$0E Set Normal Display Mode

After this character, displays all subsequent characters in normal mode.

$0F Set Inverse Display Mode

After this character, displays all subsequent characters in inverse mode.

$10 DLE Space Expansion

If `consDLE` is TRUE, interprets the next character as number of spaces + 32, and the correct number of spaces is issued to the screen. If `consDLE` is FALSE, the DLE character is ignored and the following character is processed normally.

$11 Set 40-Column Mode

Sets the screen hardware for 40-column display. If changing from 80-column display, copies the first 40 columns of the 80-column display into the 40-column display.

If the current text port does not fit in the 40-column screen, it is adjusted by one of two methods:

- If the text port is 40 columns or narrower, the entire text port (left side, right side, and cursor) is slid over until the right edge is collinear with the right edge of the screen.
- If 41 columns or wider, the port becomes 40 columns and the cursor moves to the left edge.

$12 Set 80-Column Mode

Sets the screen hardware for 80-column display. If changing from 40-column display, copies the 40-column data to the left half of the 80-column display and clears the right half of the screen to the `consFill` character.

$13 Clear from Beginning of Text Port

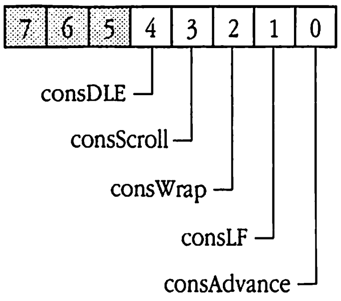
Clears all characters from the beginning of the text port up to and including the cursor location.

$14 Set Horizontal Position

Interprets the next byte as a text port–relative horizontal position + 32. If the destination is outside the current text port, the cursor is moved to the nearest edge.

$15 Set Cursor Movement Word

Interprets the next byte as cursor movement control, and sets the values of these Boolean flags:



The functions of the individual flags are described in the section "The Text Port" earlier in this chapter.

$16 Scroll Down One Line

Scrolls the text port down one line. Does not move the cursor.

$17 Scroll Up One Line

Scrolls the text port up one line. Does not move the cursor.

$18 Disable MouseText Mapping

When MouseText is disabled, uppercase inverse characters are displayed as such (see the section "Character Set Mapping" earlier in this chapter).

$19 Home Cursor

Resets the cursor to the upper-left corner of the text port.

$1A Clear Line

Clears the line that the cursor is on. Resets the cursor to the leftmost column in the window.

$1B Enable MouseText Mapping

When MouseText is enabled, uppercase inverse letters are displayed as MouseText symbols (see the section "Character Set Mapping" earlier in this chapter).

$1C Move Cursor Right

Performs a nondestructive forward-space of the cursor. If `consWrap` is TRUE, the cursor may go to the next line; if `consScroll` is TRUE, the screen may scroll up one line.

$1D Clear to End of Line

Clears from the position underneath the cursor to the end of the current line.

$1E Go to X,Y

Adjusts the cursor position relative to the text port. The parameters passed are X+32 and Y+32. If the new locations are outside the current text port, the cursor is placed on the nearest edge.

$1F Move Cursor Up

Moves the cursor up one line (reverse line feed). If the cursor is already on the uppermost line of the text port and `consScroll` is TRUE, it will cause a reverse scroll.

The Console Input routine

The console driver's Console Input routine, especially in user input mode, provides a convenient method for obtaining user input. It is best suited for the fixed-field, fill-in-the-blanks type of input with simple line-editing commands and program-defined default strings.

The console driver obtains input directly from the keyboard hardware, or from the Apple IIGS Toolbox Event Manager if it is active. The Console Input routine monitors not only the keystrokes but the modifier keys (Shift, Control, Option, and so on) and can make decisions based on both the keystroke and the current modifiers.

The input port

All information about the current input is contained in the input port, a data structure that is maintained by the Console Input routine but can be read, modified, and written back by the application program. The data structure is as follows:

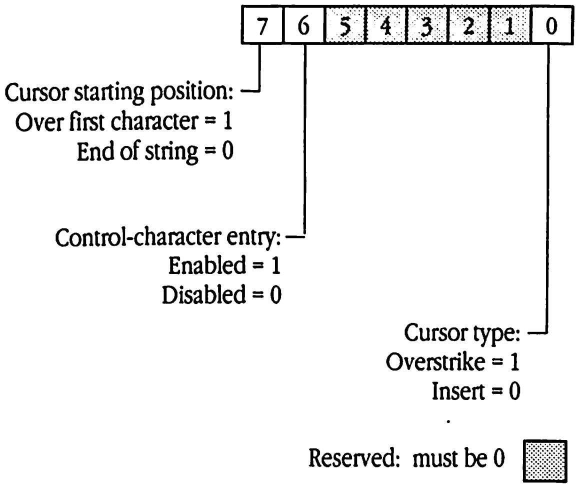
```
InputPortRec = {
        byte  fillChar,
              defCursor,
              cursorMode,
              beepFlag,
              entryType,
              exitType,
              lastChar,
              lastMod,
              lastTermChar,
              lastTermMod,
              cursorPos,
              inputLength,
              inputField,
              originH,
              originX, (word)
              originV      }
```

The meanings of each field are as follows:

`fillChar` The character that fills empty space in the input field. It is displayed by the Console Output routine so it is usually $20 (normal space) (see the section "Character Set Mapping" earlier in this chapter). Another useful fill character is the MouseText "ghost space" character. This can be displayed by setting `fillChar` to $C9. However, since MouseText characters are only available in normal mode, do not use MouseText fill characters when the screen is in inverse mode.
Default = Space ($20)

`defCursor` The default cursor-mode setting. The value in this field is placed into the `cursorMode` field at the beginning of an input cycle from the user. The application controls the cursor mode the user starts with by controlling this setting.
Default = $80 (cursor starts at end of string, control-character entry is disabled, cursor type = insert).

`cursorMode` Contains three status bits that describe the current cursor-mode setting:



If control-character entry is enabled, the user can insert control characters into the stream by typing Option-Control-*character,* where *character* is replaced by any valid keyboard character.

The value in this field may be different from `defCursorMode` because the user can switch between insert and overstrike modes during entry.

`beepFlag` If this flag is nonzero, the Console Input routine beeps on input errors (line too long, and so on).
Default = TRUE

`entryType` Tells the Console Input routine the status of the current input:
0 = initial entry
1 = interrupt reentry
2 = no-wait mode reentry
On exit, the Console Input routine adjusts this value so that it is correct for the next entry. If the application wishes to cancel an in-progress input and start with a new one, it must make the DControl subcall Abort Input.
Default = initial entry

`exitType` Tells the application which type of exit was made. (0 = input not terminated yet, either because end-of-field was reached on raw input or a no-wait exit occurred.) Any other value is the number of the terminator that halted the input.
(Set on exit from the input cycle.)

`lastChar` The ASCII value ($00–$7F) of the most recently typed key.
(Set on exit from the input cycle.)

`lastMod` The value of the modifiers mask of the most recently typed key. See the section "Terminators" later in this chapter for a description of the modifier bits.
(Set on exit from the input cycle.)

`lastTermChar` The ASCII value of the terminator (as specified in the user-supplied terminator list) that caused the most recent input termination.
(Set on exit from the input cycle.)

`lastTermMod` The value of the modifiers mask of the terminator that caused the most recent input termination.
(Set on exit from the input cycle.)

`cursorPos` Index of the cursor within the input string. (0 = over the first character.) The cursor is allowed to move from the beginning of the string to one position past its end.
Default = position of cursor when input begins

`inputLength` The length of the input string at the current state of editing. This is the length that is returned in the Transfer Count.
Default = length of default input string

`inputField` Used internally by the UIR. It is useful only when restoring the InputPortRec. This field is calculated when a new call is made to UIR. Default = no default

`originH` Contains the cursor's horizontal position.

`originX (word)`
Contains a variable used by the UIR.

`originV` Contains the cursor's vertical position.

Using raw mode

Raw mode is the simplest form of user input. The keyboard is simply scanned until (1) a number of keys equal to `requestCount` has been pressed, or (2) a specified terminator has been typed. As with other serial input drivers, the terminator is included in the transferred string. There are no echo, no cursor, and no editing.

Using user input mode

This input mode provides more functions than raw mode. The following steps are required to use it:

1. If the application wishes to supply a default string, it must do so by using a Control subcall (see "DStatus" later in this chapter).
2. If modes other than the default modes are desired, the application must read the input port, adjust it, and write it back.
3. Terminators must be assigned with a SetTerminators call (DControl subcall).
4. The cursor must be positioned to the desired start of the input field with a `Go To X,Y` instruction.

A Read call is made to initiate user input mode. If only simple terminators have been requested, the Console Input routine will return as soon as one has been pressed. If there are interrupt terminators, the application must make calls to determine the type of interruption and determine whether more work (repeated read entries) is necessary.

Terminators

A **terminator** is a character that, when read, terminates or interrupts a Read call. The console driver permits more than one terminator character and also can note the state of modifier keys in considering whether a character is to be interpreted as a terminator.

The console driver keeps track of terminators with a **terminator list.** The terminator list is set using a control call (see "DStatus" later in this chapter). This is the format of a terminator list:

```
TermList = {
        word  termMask,
              termCount,
              termList [ 1 ... termCount ]
        }
```

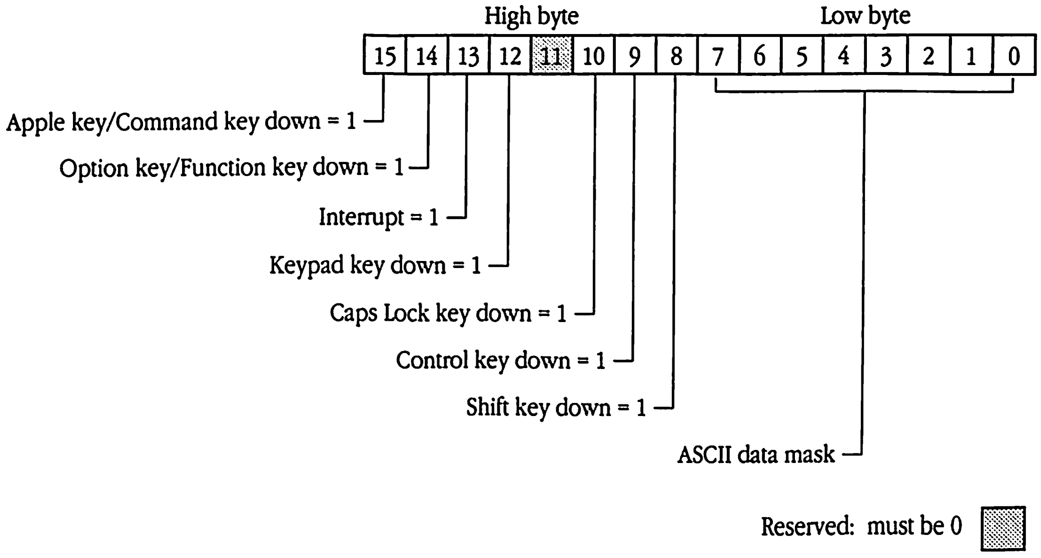
The fields have the following meanings:

`termMask` A mask that is added to the input data with an AND operator before it is compared to the terminator list entries. The high-order byte is the modifiers mask; it is used to mask out irrelevant modifiers (for example, if it doesn't matter whether the keystroke was made from the main keyboard or the keypad). The low-order byte is the ASCII mask; it is used to simplify ASCII comparisons (for example, if it doesn't matter whether a character is uppercase or lowercase).

`termCount` A count of the number of terminators. A count of 0 means terminators are disabled and there is no list. It specifies the number of entries, so it must be multiplied by two to get a byte count. The maximum terminator count is 254.

`termList` A list of terminator characters and their modifiers. Each entry is in the same format as `termMask`; the high byte is the modifiers mask, and the low byte is the ASCII value of the terminator character. After the incoming data is combined with the terminator mask in a logical AND operation, the data is compared with each of the entries in the terminator list. A match causes a termination. In addition, if the application supplies a term list entry with bit 13 set, this entry is an interrupt terminator. The Console Input routine will give up control but is set up to restart the input. The application can use this capability to implement help screens or custom editing keys.

The terminator mask has the following format:



How to disable terminators

The application can disable terminators by doing either of the following:

- setting the mask to 0
- setting the count to 0

In addition, if a memory error occurs while new terminators are being received, the terminator list is dumped.

If an incorrectly formed list (for example, if count = 255) is sent to the Console Input routine, it is discarded and the original terminators remain in place.

Terminators and newline mode

Newline characters as defined by the Character FST are incompatible with terminators as defined by the console driver's user input mode. If you need a combined newline/termination mode, use only the following combinations:

Character FST	**Console driver**
Newline mode enabled	Raw input mode, terminators disabled
Newline mode disabled	Raw input mode, terminators enabled
Newline mode disabled	User input modes

User-input editing commands

The following editing commands are supported by the console user input mode:

← *or* Control-H	Move cursor backward one position.
→ *or* Control-U	Move cursor forward one position.
Option-→	Move cursor to end of next word.
Option-←	Move cursor to beginning of previous word.
Option-> *or* Option-.	Move cursor to end of line.
Option-< *or* Option-,	Move cursor to beginning of line.
Delete *or* Control-D *or* Control-Delete *or* Option-Delete *or* Option-D	Delete character to left of cursor and move cursor and string to left (destructive backspace).
Control-F *or* Option-F	Delete the character underneath the cursor and move the rest of the string to the left.
Control-X *or* Option-X *or* Clear	Delete entire input string.
Control-Y *or* Option-Y	Clear string from cursor to end.
Control-Z *or* Option-Z	Reset input string to application-specified default.
Control-E *or* Option-E	Toggle between insertion and overstrike characters.
Option-Control-*character*	Insert control character into input string (if enabled; control-character insertion is enabled by setting a bit in `cursorMode`).

Using no-wait mode

No-wait mode is defined so that drivers will not hold control of the system. When in wait mode, a Read call does not terminate until the requested number of characters (or a terminator) is received. When in no-wait mode, the system returns immediately from a Read call as soon as there is no more input available. In such a case, it is the responsibility of the application program to continue calling the input routines until the final number of characters has been transferred.

Device calls to the console driver

The GS/OS console driver supports the standard set of device calls:
DInfo
DStatus
DControl
DRead
DWrite

The standard calls are described in Chapter 7. The rest of this chapter documents the driver-specific DStatus and DControl subcalls, and describes how the console driver handles any of the standard device calls differently from the ways documented in Chapter 7. Any calls or subcalls not discussed here are handled exactly as documented in Chapter 7.

DStatus ($202D)

This call is used to request status information from the console driver. For DStatus, the console driver supports most of the standard subcalls and several device-specific subcalls. Status subcalls are specified by the value of the status code parameter. The following status codes are supported:

Status code	Subcall name	Status code	Subcall name
$0000	GetDeviceStatus	$8002	GetTerminators
$0001	GetConfigParameters	$8003	SaveTextPort
$0002	GetWaitStatus	$8004	GetScreenChar
$8000	GetTextPort	$8005	GetReadMode
$8001	GetInputPort	$8006	GetDefaultString

Calls with status codes of less than $8000 are standard Status subcalls; calls with status codes of $8000 and over are device-specific subcalls. The calls are described more fully in the following sections.

Standard DStatus subcalls

Standard DStatus subcalls that are not described here function exactly as documented in *GS/OS Device Driver Reference.*

GetConfigParameters (DStatus subcall)

The console driver obtains its setup information from battery RAM and therefore uses no control parameters. This call returns an empty control parameter record (a zero).

The minimum request count is 2. The maximum transfer count is 2.

GetTextPort (DStatus subcall)

`statusCode` = $8000

status list = a text port record

This subcall copies the contents of the current text port record into the status list buffer. See the section "The Text Port" earlier in this chapter for more details.

The minimum request count is 0. The maximum transfer count is 16.

GetInputPort (DStatus subcall)

`statusCode` = $8001

status list = input port record

This subcall copies the contents of the current input port record into the status list buffer. See "The Input Port" earlier in this chapter for more details.

The minimum request count is 0. The maximum transfer count is 12.

GetTerminators (DStatus subcall)

`statusCode` = $8002

status list = terminator list record

This subcall copies the current terminator list into the status list buffer. The format of the list is count, enable/mask, terminator list. See the section "Terminators" earlier in this chapter for details.

This call transfers only complete terminator lists. The minimum request count is (number of entries * 2) + 4. The transfer count is set to this value. The maximum transfer count is 514: 4 bytes of header and 255 terminator words.

SaveTextPort (DStatus subcall)

`statusCode` = $8003

status list = text port size and contents

This subcall copies not the text port record but the actual text port screen data into the status list buffer. The format of the data as written is `windWidth`, `windLength`, screen bytes (the contents of screen memory within the limits of the port). The size of the status list in bytes is therefore (`windWidth` times `windLength`) plus 2.

This call transfers only a complete screen data record. The minimum request count is the status list size as calculated.

GetScreenChar (DStatus subcall)

`statusCode` = $8004

status list = 1 byte

This subcall copies the current screen byte (that is, the byte underneath the cursor) to the status list. Note that this is the actual value of the byte in screen memory, which has a complex relation to the character's ASCII value. See the section "Character Set Mapping" earlier in this chapter.

The minimum request count is 1. The maximum transfer count is 1.

GetReadMode (DStatus subcall)

`statusCode` = $8005

status list = 2 bytes

This subcall copies the current read mode flag into the status list. If zero, input is in user input mode. If $8000, input is in raw mode. The value of the read mode flag is set by the DControl subcall SetReadMode, described later in this chapter.

The minimum request count is 2. The maximum transfer count is 2.

GetDefaultString (DStatus subcall)

`statusCode` = $8006

status list = character string

This subcall copies the current default input string into the status list. This string (set with the DControl subcall SetDefaultString) is placed in the input field at the beginning of each cycle of user input. The string can have only standard ASCII ($00–$7F) characters, and can be no more than 254 characters long.

The request count in this case defines the maximum number of bytes that can be returned.

DControl ($202E)

This call is used to send control information to the console driver. For DControl, the console driver supports most of the standard subcalls and several device-specific subcalls. Control subcalls are specified by the value of the control code parameter. The following control codes are supported:

Control code	Meaning	Control code	Meaning
$0000	ResetDevice	$8002	RestoreTextPort
$0001	FormatDevice	$8003	SetReadMode
$0002	EjectMedia	$8004	SetDefaultString
$0003	SetControlParam	$8005	AbortInputeters
$0004	SetWaitStatus	$8006	AddTrap
$8000	SetInputPort	$8007	ResetTrap
$8001	SetTerminators		

Calls with control codes of less than $8000 are standard Control subcalls; calls with control codes of $8000 and over are device-specific subcalls. The calls are described more fully in the following sections.

Standard DControl subcalls

Standard DControl subcalls that are not described here function exactly as documented in the *GS/OS Device Driver Reference.*

FormatDevice (DControl subcall)

This subcall is not applicable to character devices. It returns with no error. The transfer count is 0.

EjectMedia (DControl subcall)

This subcall is not applicable to character devices. It returns with no error. The transfer count is 0.

SetConfigParameters (DControl subcall)

The console driver obtains its setup information from parameter RAM and has no configuration parameters. The transfer count is 0.

SetInputPort (DControl subcall)

`controlCode` = $8000

control list = input port record

This subcall transfers data from the control list to the input port record. The data must be in the format of an input port record; see the section "The Console Input Routine" earlier in this chapter.

The minimum request count is 12. The maximum transfer count is 12.

SetTerminators (DControl subcall)

`controlCode` = $8001

control list = terminator list record

This subcall copies data from the control list to the terminator list. The format of the list is described in the section "Terminators" earlier in this chapter. The length of a terminator list in bytes is (2 * *count*) + 4, where *count* is the number of entries in the list. The minimum list length is 4; the maximum list length is 514 (2 header words plus 255 terminator characters).

The minimum request count for this subcall is 4. Furthermore, the request count must match the calculated length based on the entry count parameter in the list. If there is a match, the transfer count is set to the length of the list. If the length is incorrectly stated, the previous terminators remain in effect and error $22 (`drvrBadParm`) is returned. The driver requests memory from the GS/OS Info Manager to store the terminators; if the request fails the previous and new lists of terminators are lost and error $26 (`drvrNoResrc`) is returned.

RestoreTextPort (DControl subcall)

`controlCode` = $8002

control list = text port record

This subcall copies data (previously obtained through the DStatus subcall GetTextPort) from the control list back into screen memory (and thereby onto the screen). The format of the data is `windWidth`, `windLength`, screen bytes (the data to be written to screen memory within the limits of the port). If the buffer is larger than the current text port, only the upper-left part of the data (as much as will fit) is transferred to the screen. If the buffer is smaller than the current text port, only that much of the text port (starting from the upper-left corner) will be changed; the rest of it will remain as it was before the subcall was made.

Only a complete screen record can be transferred. The minimum request count is 4. Furthermore, the request count must match the calculated length based on the width and length parameters in the control list. The total data length is therefore (`windWidth` * `windLength`) + 4. If the list is complete, the transfer count is set to that value.

SetReadMode (DControl subcall)

`controlCode` = $8003

control list = 2 bytes

This subcall sets the flag that specifies the console driver's read mode. Only the high-order bit is significant and all other bits must be set to zero. A value of $0000 selects user input mode; $8000 selects raw mode.

The minimum request count is 2. The maximum transfer count is 2.

SetDefaultString (DControl subcall)

`controlCode` = $8004

control list = character string

This subcall sets the default string for user input. This string is placed in the input field at the beginning of each cycle of user input. The string can have only the standard ASCII ($00–$7F) characters, and can be no more than 254 characters long. Control characters will be displayed in inverse video. To disable the current default input string, pass a length of 0 as the request count. The driver requests memory from the GS/OS Info Manager to store the default string; if the request fails, error $26 (resource not available) is returned.

The minimum request count is 0. The maximum transfer count is 254.

AbortInput (DControl subcall)

`controlCode` = $8005

control list = none

This subcall cancels an input session currently in progress. If `entryType` is zero, there is no input in progress and this call is ignored. Otherwise, `entryType` is reset to zero, and if a cursor is on the screen, it is removed.

The minimum request count is 0. The transfer count is 0.

AddTrap (DControl subcall)

`controlCode` = $8006

control list = none

This call takes the user-supplied address and installs it in the console driver trap vector. The trap handler will be called by a `JSL` instruction. If the handler wishes to handle the call, it should pull the return address off the stack, handle the call, then exit via an `RTL` instruction. When the trap handler is called, the environment is set to the same environment as device drivers. Furthermore, if the trap handler does not wish to handle the call it must restore all registers and direct-page locations that it has used.

To issue the subcall, you must set the request count to 4 and set the longword pointed to by the control list pointer to the address of the trap handler. If a trap is already installed, an error will be returned.

The minimum request count is 4. The transfer count is 4.

ResetTrap (DControl subcall)

`controlCode` = $8007

control list = none

This call will remove a user-installed trap vector if the vector is installed. The request count should be set to zero.

The transfer count is 0.

DRead ($202F)

This call reads characters from the keyboard. Depending on read mode, the call either begins waiting for raw entry values, or activates the user input mode.

In raw mode, the keyboard is scanned until (1) the transfer count equals the request count, or (2) a terminator is pressed. The terminator character is returned as the last character of the string.

In user input mode, the request count becomes the length of an edit field on the screen. This edit field begins at the current cursor location. An optional default string is displayed in the edit field. The user can edit this field using the standard editing controls, and finish editing by typing a terminator key. The terminator is treated as an editing key—it is not included in the returned string.

In either mode, an additional return condition exists if no-wait mode is selected. On exit, the transfer count reports the length of the final string.

DWrite ($2030)

This call transfers the contents of the buffer, one byte at a time, through the console driver and to the screen. The entire buffer is transferred, and since all byte values ($00 to $FF) are defined, there are no possible errors (as long as the driver is open).

Chapter 10 **Handling Interrupts and Signals**

Interrupt handlers are programs that execute in response to a hardware interrupt. Interrupts and interrupt handlers are commonly used by device drivers to operate their devices more efficiently and to make possible simple background tasks such as printer spooling.

Under GS/OS, a **signal** is a software message from one subsystem to a second that something of interest to the second has happened. The most common kind of signal is a software response to a hardware interrupt, but signals need not be triggered by interrupts. **Signal handlers** are programs that execute in response to the occurrence of a signal. They are similar to interrupt handlers except that signal handlers can make operating system calls. The GS/OS Call Manager is responsible for managing and dispatching to both interrupt handlers and signal handlers.

An interrupt handler is commonly written in conjunction with a driver and is installed when the driver starts up. A signal handler is commonly written in conjunction with either a driver or an application, and it is installed by the driver or application during execution. This chapter discusses requirements for designing and installing both types of handlers.

Interrupts

An **interrupt** is a hardware signal that is sent from an external or internal device to the CPU. On the Apple IIGS, when the CPU receives an interrupt the following actions occur:

1. The CPU suspends execution of the current program, saves the program's state, and transfers control to the Apple IIGS firmware interrupt dispatcher. The firmware dispatcher sets up a specific firmware interrupt environment.
2. If it is an interrupt that has a GS/OS interrupt handler, the firmware dispatcher passes control to GS/OS. GS/OS sets up a specific GS/OS interrupt environment and in turn transfers control to the proper handler.
3. The interrupt handler performs the functions required by the occurrence of the interrupt. After it has done its job, the interrupt handler returns control to GS/OS.
4. GS/OS restores the firmware interrupt environment and returns control to the firmware dispatcher. The firmware dispatcher restores the state of the interrupted application and returns execution to it as if nothing had happened.

In a nonmultitasking system such as GS/OS, interrupts are commonly used by device drivers to operate their devices more efficiently and to make possible simple background tasks such as printer spooling.

This section discusses what the sources of interrupts are, how interrupt handlers are dispatched to, how interrupt handlers function within their execution environment, and how interrupt sources are connected to interrupt handlers. It also discusses interrupt-handler lifetime and how GS/OS treats unclaimed interrupts.

Interrupt sources

Each distinct hardware device that can generate an interrupt is known as an **interrupt source.** For example, each Apple IIGS expansion slot with a hardware card is an interrupt source, and internal devices such as the mouse and serial ports are also sources. Every interrupt source that is explicitly identifiable by the firmware has a unique identifier known as its vector reference number (VRN). VRNs are used to associate interrupt sources with interrupt handlers.

VRNs are permanently associated with specific interrupt sources; they will not change with future revisions to GS/OS or the Apple IIGS computer. If your interrupt handler now appropriately handles an interrupt source with VRN = *n*, it will be able to handle VRN = *n* on any future versions of GS/OS on any Apple IIGS.

Table 10-1 lists the currently defined VRNs and their associated interrupt sources.

■ **Table 10-1** VRNs and interrupt sources

VRN	Interrupt source
$0008	AppleTalk port
$0009	Serial input port
$000A	Scan line
$000B	Sound-chip waveform completion
$000C	VBL
$000D	Mouse button or movement
$000E	Quarter-second timer
$000F	Keyboard
$0010	ADB response (keyboard)
$0011	SRQ (keyboard)
$0012	Desk Manager
$0013	Flush command (keyboard)
$0014	Microcontroller abort (keyboard)
$0015	Clock chip 1-second timer
$0016	VGC external interrupt source (unused)
$0017	Other interrupt source (slot)

As new interrupt sources (such as internal and external slots, timers, counters, etc.) are defined in future versions of the Apple IIGS, each will be assigned a unique VRN by Apple Computer, Inc.

Interrupt dispatching

Interrupt dispatching is the process of handing control to the appropriate interrupt handler after an interrupt occurs. In the Apple IIGS, most interrupt dispatching and interrupt handling are performed by firmware. Although the Apple IIGS hardware generates a number of distinct interrupt notifications—ABORT, COP, BRK, NMI, and IRQ—the only interrupt of interest to GS/OS interrupt–handler writers is IRQ (Interrupt Request). The firmware dispatches each IRQ by polling the interrupt handlers through the firmware interrupt vectors (one for each VRN defined in Table 10-1) until one of them signals that it has handled the interrupt.

Because of critical timing constraints, the firmware interrupt dispatcher polls the AppleTalk and serial port vectors first, before polling the less time-critical vectors such as vertical blanking, quarter-second timer, and keyboard. If none of the firmware handlers associated with defined sources accepts the interrupt, the firmware dispatcher polls through vector $0017 (other interrupt source). If the interrupt still remains unhandled, the firmware dispatcher passes control through the user interrupt vector at $00 03FE. Finally, if no handlers associated with the user interrupt vector accept the interrupt, it becomes an **unclaimed interrupt,** described later in this section.

There are two ways in which GS/OS can get control from the firmware dispatcher during this process, in order to pass control on to a GS/OS interrupt handler:

1. Through one of the firmware interrupt vectors. When GS/OS gets control this way, it polls only the interrupt handlers that are associated with the particular vector reference number (VRN) of that interrupt vector. These handlers are installed with the GS/OS call BindInt, described later in this section.
2. Through the user interrupt vector ($00 03FE). When GS/OS gets control this way, it polls all the installed ProDOS 16 interrupt handlers. ProDOS 16 interrupt handlers are installed with the GS/OS ProDOS 16–compatible call `ALLOC_INTERRUPT`, described in Appendix A.

Within a polling sequence, the polling order is undefined.

Interrupt handler structure and execution environment

A GS/OS interrupt handler consists of code in either a device driver, application, or desk accessory. The interrupt handler must have a single defined entry point. When an interrupt occurs, GS/OS sets up a specific execution environment and then calls the interrupt handler with a `JSL` instruction to that entry point.

The code beginning at the specified entry point should first determine whether or not the interrupt is the one to be handled by this interrupt handler. If it is not, the interrupt handler should restore the execution environment as set up by GS/OS, set the carry flag (c = 1), and return with an `RTL`. If the interrupt is the proper one, the interrupt handler should perform whatever tasks necessary to handle the interrupt, restore the proper execution environment, clear the carry flag (c = 0), and return with an `RTL`.

What execution environment GS/OS sets up for an interrupt handler depends on its type. As far as execution environments are concerned, there are three basic types:

- GS/OS interrupt handler bound to the AppleTalk or serial port firmware vector
- GS/OS interrupt handler bound to any other firmware vector
- ProDOS 16 interrupt handler installed through the user interrupt vector

Table 10-2 shows the execution environment of each of these handlers when it starts executing. The table also notes which parts of the environment need to be preserved (or restored on exit). Boldface entries in the table indicate the components of the environment that the handler must restore before returning.

■ **Table 10-2** Interrupt-handler execution environments

Component	GS/OS AppleTalk or serial handler	Other GS/OS handler	ProDOS 16 handler
Registers			
A, X, Y	Undefined	Undefined	Undefined
D	**Undefined**	**$0000**	Undefined
S	Undefined[*]	Undefined[*]	Undefined[*]
DB	**Undefined**	**$00**	Undefined
PB	Handler entry point	Handler entry point	Handler entry point
PC	Handler entry point	Handler entry point	Handler entry point
P register flags			
e	**0 (native mode)**	**0 (native mode)**	**0 (native mode)**
m	**1 (8-bit)**	**1 (8-bit)**	**0 (16-bit)**
x	**1 (8-bit)**	**1 (8-bit)**	**0 (16-bit)**
i	1 **(disabled)**[†]	1 **(disabled)**[†]	1 **(disabled)**[†]
c	Undefined[‡§]	Undefined[‡§]	1[§]
Speed	**Fast**	**Fast**	**Fast**

[*] On entry, the 3-byte return address to GS/OS is on top of the stack. When the interrupt handler executes its RTL, this 3-byte address is popped from the stack.

[†] An interrupt handler must never enable interrupts.

[‡] If c = 0 on entry, the interrupt has not yet been handled; if c = 1 on entry, the interrupt has already been handled.

[§] If the interrupt handler handles the interrupt, it sets c = 0 before returning. If not, it sets c = 1 before returning.

Note from Table 10-2 that the carry flag is always set (c = 1) on entry to a ProDOS 16 interrupt handler, whereas it can be either 0 or 1 on entry to a GS/OS interrupt handler. ProDOS 16 handlers are polled only as long as the interrupt is still unclaimed; as soon as one handler takes it and clears the carry flag, polling stops. On the other hand, all GS/OS handlers bound to a particular VRN are polled during an interrupt, even if another handler with that VRN has already cleared the interrupt. That way, all handlers associated with a VRN can do updating or other desired tasks at each interrupt.

The first GS/OS handler to respond to an interrupt should perform its normal functions, including reenabling the interrupt source, clearing the carry flag, and returning. Subsequent handlers, on seeing that c = 1 on entry, may perform other tasks as desired but should not themselves reenable the interrupt source, change the value of the carry flag, or permanently modify the environment.

Here are some other points to remember in designing an interrupt handler:

- If the interrupt handler needs to use direct-page space, it must save and restore the contents of any locations that it uses.
- An interrupt handler must never enable interrupts.
- Because interrupts cannot be disabled for longer than 0.25 seconds in the Apple IIGS (an AppleTalk requirement), interrupt handlers must execute in less than a quarter-second.
- Because GS/OS is not reentrant, an interrupt handler should not make GS/OS calls. If your interrupt handler needs to make operating system calls, you should make it a signal handler instead. See "Signals," later in this chapter.

Connecting interrupt sources to interrupt handlers

You install and remove GS/OS interrupt handlers by making the standard GS/OS calls BindInt and UnbindInt, respectively.

To avoid unclaimed interrupts, make sure that the code that installs an interrupt handler does not enable the interrupt source until the interrupt handler is installed. Likewise, the code that removes an interrupt handler must disable the interrupt source before removing the handler.

BindInt call

This call establishes a binding, or correspondence, between a specified interrupt source and a specified GS/OS interrupt handler. GS/OS adds the interrupt handler to the set of handlers to be polled when the specified (by VRN) interrupt occurs. The polling order is undefined within the handlers bound to that interrupt vector.

The interrupt identification number returned by the call uniquely identifies the binding between interrupt source and interrupt handler. Its only use is in the GS/OS UnbindInt call. Note that several interrupt handlers may be bound to the same interrupt source.

For a description of the BindInt call, see Chapter 7.

UnbindInt call

This call severs the binding previously established between an interrupt source and interrupt handler by a BindInt call. It makes the interrupt handler unavailable.

For a description of the UnbindInt call, see Chapter 7.

◆ *Note:* ProDOS 16 interrupt handlers are installed and removed with the ProDOS 16 calls `ALLOC_INTERRUPT` and `DEALLOC_INTERRUPT`. See Appendix A.

Interrupt handler lifetime

The lifetime of an interrupt handler is the time during which its code is resident in memory and capable of being executed. During its lifetime, the interrupt handler may be installed (able to handle its interrupts) or removed (still resident in memory but unable to handle its interrupts).

The interrupt handler is installed when the device driver or application makes a BindInt call for it, and removed when the device driver or application makes an UnbindInt call. *The program that performs the BindInt call must perform an UnbindInt call before the lifetime of the interrupt handler ends.* There is no automatic mechanism for removing GS/OS interrupt handlers when an application quits, and a dispatch to the previous entry point of an installed but now completely gone interrupt handler could cause a system crash or loss of data.

◆ *Note:* Drivers can make BindInt and UnbindInt calls; this is an exception to the rule that drivers cannot make operating system calls.

A GS/OS interrupt handler has a lifetime equivalent to the code containing it. For example, if the interrupt handler is part of a device driver, it lives as long as the device driver is in memory and capable of being executed. Thus, the lifetime of a GS/OS interrupt handler may span several GS/OS applications. In this case, the lifetime ends when the user executes a non-GS/OS application or the hardware reboots.

Unclaimed interrupts

If none of the interrupt handlers on an Apple IIGS accepts a given interrupt, it is known as an unclaimed interrupt. Possible causes of unclaimed interrupts include the following:

- software problems, such as a failure to bind the interrupt handler before enabling the interrupt source it handles
- interrupt-related hardware problems, such as failure by the interrupting device to maintain an "I am the source of the interrupt" flag after signalling an interrupt to the processor

- hardware failures such as intermittent shorts of the interrupt line to ground
- random transient phenomena such as cosmic-ray or subatomic-particle bombardment

An unclaimed interrupt is a serious problem but shouldn't cause a system failure if the interrupt was due to a random transient phenomenon. Therefore, GS/OS maintains an unclaimed interrupt counter that is initialized to 0 at GS/OS startup time. Whenever an unclaimed interrupt occurs, GS/OS increments the counter. Whenever an interrupt is serviced by an interrupt handler, GS/OS sets the counter back to 0. If the counter ever reaches 65,536, GS/OS might cause a system failure.

Signals

A signal is a message from one software subsystem to a second that something of interest to the second has occurred. When a signal occurs, GS/OS typically places it in the signal queue for eventual handling. As soon as it can, GS/OS suspends execution of the current program, saves the program's state, removes the signal from the queue, calls the signal handler in the receiving subsystem to process the signal, and finally restores the state and returns to the suspended program.

The most important feature of signal handlers is that they are allowed to make GS/OS calls. That is why the signal queue exists; GS/OS removes signals from the queue and executes their signal handlers only when GS/OS is free to accept a call.

The most common kind of signal is a software response to a hardware interrupt. For example, a modem driver may use a *loss of carrier* interrupt to trigger a corresponding signal, whose signal handler calls GS/OS to close a file of terminal input data. Similarly, a spooling printer driver may translate a *line completion* interrupt into a corresponding signal whose signal handler uses GS/OS calls to read the next line from a spool file and move it into the printer's output buffer.

In principle, however, signals need not be triggered by interrupts: a signal can indicate, for example, a *message received* condition on a network interface or a *new volume mounted* condition on a disk drive.

◆ *Note:* Signals are not meant to provide a general mechanism for interprocess communication in a multitasking environment. Their principal capability is synchronization of handler execution with time periods when the operating system is able to accept calls.

Signals are analogous to interrupts but are handled with less urgency. If immediate response to an interrupt request is needed, and if the routine that handles the interrupt needn't make any operating system calls, then it should be an interrupt handler. On the other hand, if a certain amount of delay can be tolerated, the full range of operating system calls are available to a handler if it is a signal handler.

This section discusses what signal sources are, how GS/OS dispatches to signal handlers, how signal handlers function within their execution environment, how signal sources are connected with signal handlers, and how the occurrence of a signal is announced.

Signal sources

A **signal source** is software; it is a routine that announces the occurrence of a signal when it detects the prerequisite conditions for that signal. For example, a modem device driver may contain an interrupt handler capable of detecting the conditions needed to announce the *loss of carrier* signal. In that case the interrupt handler's primary purpose is to be a signal source. The most common class of signal sources is probably interrupt handlers within device drivers.

Signal sources announce signals to GS/OS by making the system service call SIGNAL (described in the *GS/OS Device Driver Reference*). When a signal source announces a signal to GS/OS, it passes along the information needed to execute the source's signal handler. (That information was sent to the signal source when the signal was armed; see "Arming and Disarming Signals," later in this chapter.) GS/OS accepts that information and either executes that signal's signal handler immediately or saves the information for later; GS/OS then returns control to the process that announced the signal.

◆ *Note:* A signal source that announces a signal as the result of an interrupt should generate no more than one signal per interrupt, to avoid the possibility of overflowing the signal queue.

Signal dispatching and the signal queue

Signal dispatching is the process of calling signal handlers. GS/OS dispatches signals only when it is not busy processing a GS/OS call, so that signal handlers are always able to make system calls.

When a signal occurs, if GS/OS is not busy handling a GS/OS call and if the system is in a noninterrupt state, the GS/OS Call Manager executes the signal handler immediately. On the other hand, if a GS/OS call is in progress when the signal occurs, the signal cannot be dispatched; the Call Manager instead places the signal in the signal queue. Signals are placed in the queue in order of signal priority; queued signals with higher priority numbers are placed in front of signals with lower priorities, meaning that they will be executed first.

The signal queue can hold a maximum of 16 signals. If a signal arrives and the queue is full, the queue overflows, and the signal call returns an error.

GS/OS dispatches a queued signal by pulling it off the front of the queue (that is, by taking the oldest signal with the highest priority) and calling the signal's handler. To process signals as quickly as possible, minimize the time during which interrupts are disabled, and assure that all signals are eventually handled, GS/OS uses the signal-dispatching strategy described in Table 10-3.

■ **Table 10-3** GS/OS signal-dispatching strategy

Situation	Action taken
GS/OS is exiting from a system call; system is in noninterrupt state.	Execute all queued signals.
GS/OS is exiting from a system call; system is in interrupt state.	Execute only the first queued signal.
Signal arrives while GS/OS is inactive and the system is in noninterrupt state.	Execute all queued signals, including the one being signaled.
Signal arrives while GS/OS is inactive and the system is in interrupt state.	Queue the arriving signal and execute only the first queued signal.
Signal arrives while GS/OS is active.	Do not execute any signals and queue the arriving signal.

In addition, to make absolutely sure that no signals are left unexecuted, GS/OS uses the VBL interrupt to execute all remaining signals in the queue every 0.5 seconds.

Signal handler structure and execution environment

A signal handler is a subroutine somewhere in memory that is called by GS/OS in response to the signal that it handles. The signal handler must have a single defined entry point. When it dispatches to the signal handler, GS/OS saves the state of the current application and sets up a specific signal handler environment; GS/OS then calls the signal handler with a JSL instruction to its entry point. The features of the signal handler environment are shown in Table 10-4. Boldface entries in the table indicate the components of the environment that the handler must restore before returning.

■ **Table 10-4** Signal-handler execution environment

Component	State
Registers	
A	Undefined
X	Undefined
Y	Undefined
D	Current direct page
S	Current stack pointer
DBR	Undefined
P register flags	
e	**0 (native mode)**
m	**0 (16-bit)**
x	**0 (16-bit)**
i	**1 (disabled)**[*]
Speed	High

[*] A signal handler must never enable interrupts.

Here are some other points related to signal handler design:

- Signal handlers must return with an RTL.
- Because interrupts cannot be disabled for longer than 0.25 seconds on the Apple IIGS (an AppleTalk requirement), and because signal handlers may run in an interrupt environment (during which interrupts are disabled), signal handlers must execute in less that a quarter-second.
- Signal handlers must never enable interrupts.
- An interrupt may preempt execution of a signal handler, but a signal handler is never preempted to execute another signal handler, even one of higher priority. Any signal handler that you write can count on execution without interference from another signal handler.
- The lifetime of a signal handler is the same as the lifetime of the software that contains it. Therefore, if your signal handler is part of a device driver, it can span several applications.

Arming and disarming signals

A program needs to arm, or install, a signal in order to use it. Arming a signal is the process of providing its signal source with the information needed to execute its signal handler. This information includes the signal handler's code entry point and the signal's priority. Arming implies that the signal handler is ready to process occurrences of the signal.

When the program no longer needs to use the signal, it must disarm (remove) it. Disarming a signal is the process of telling the signal source that the signal handler will no longer process occurrences of the signal.

Therefore, every signal source must support the ArmSignal and DisarmSignal functions for its signal. How the source implements the functions is source-specific; however, it must at least save the information passed to it by ArmSignal and maintain a flag noting whether the signal is currently armed or disarmed. Two standards exist for ArmSignal and DisarmSignal calls: one for signal sources in device drivers and one for all other signal sources.

Arming device driver signal sources

To arm a signal that is generated by a device driver, the caller (application or device driver) performs an ArmSignal subcall of the GS/OS call DControl, passing the following information to the driver that contains the signal source:

- The signal code, an arbitrary value defined by the signal source to identify the signals that the source generates. The signal code is used only in the DisarmSignal call.
- The signal priority to be given to signals from this source; $0000 is the lowest priority and $FFFF is the highest.
- The signal-handler address, the entry point of the handler for signals generated by this source.

The driver receives the call (from the device dispatcher) as an Arm_Signal subcall of the driver call Driver_Control. The format in which these parameters are passed, and the procedure for making the ArmSignal subcall, are documented under "DControl" in the *GS/OS Device Driver Reference;* the format in which the driver receives the parameters is documented under "Driver_Control" in the *GS/OS Device Driver Reference.*

△ **Important** Before it arms a given signal, the program making the ArmSignal call must ensure that the signal handler for that signal is ready to process the signal. △

The ArmSignal subcall can return error number $22 (drvrBadParm) or error number $29 (drvrBusy, which is this case means that the signal is already armed).

Disarming device driver signal sources

To disarm a signal that is generated by a device driver, the caller (application or device driver) performs a DisarmSignal subcall of the GS/OS call DControl, passing the following information to the driver that contains the signal source:

- The signal code, the value assigned by the caller when the signal was armed (with the ArmSignal call).

The driver receives the call (from the device dispatcher) as a Disarm_Signal subcall of the driver call Driver_Control. The format in which the parameter is passed, and the procedure for making the DisarmSignal subcall, are documented under "DControl" in the *GS/OS Device Driver Reference;* the format in which the driver receives the parameters is documented under "Driver_Control" in the *GS/OS Device Driver Reference.*

△ **Important** The program making the DisarmSignal call must not disable or remove the signal handler from memory until after the call is made. △

The Disarm Signal subcall can return error $22 (drvrBadParm, which in this case means that the signal was never armed).

Arming other signal sources

A signal source that is not part of a device driver must have an ArmSignal entry point that behaves essentially like the ArmSignal subcall of DControl. The application or device driver calls the entry point by using a JSL instruction, as shown in this APW assembly-language example:

```
pea    parameter_block|-16    ;push high word of param block ptr
pea    parameter_block        ;push low word of para block ptr
jsl    arm_signal_e           ;long jump to arm procedure
```

The parameter block should have the following form:

```
dc     i2'signal_code'
dc     i2'priority'
dc     i4'handler_address'
```

These parameters have the same format and meaning as those described under "Arming Device Driver Signal Sources," earlier in this section.

On an ArmSignal call, a non-device-driver signal source must return with the carry flag clear (c = 0) if no error occurred, or with the flag set (c = 1) and the error code in the accumulator if an error occurred. The call should support these errors:

Code	Meaning
A = $0001	Invalid signal code
A = $0002	Signal already armed

Disarming other signal sources

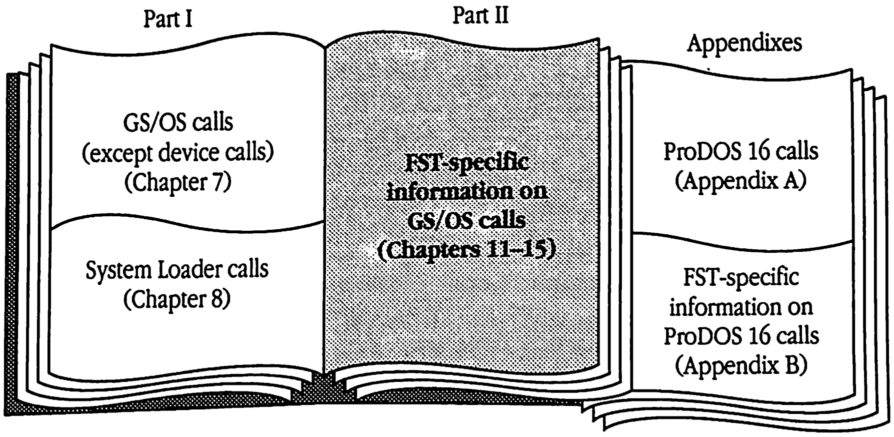
A signal source that is not part of a device driver must have a DisarmSignal entry point that behaves essentially like the DisarmSignal subcall of DControl. The application or device driver calls the entry point, as shown in this APW assembly-language example:

```
pea    signal_code        ;push signal code onto stack
jsl    disarm_signal_e    ;call disarm procedure for the specific signal
```

On a DisarmSignal call, a nondevice-driver signal source must return with the carry flag clear (c = 0) if no error occurred, or with the flag set (c = 1) and the error code in the accumulator if an error occurred. The call should support this error:

Code	Meaning
A = $0001	Invalid signal code

Part II The File System Level


Part I
GS/OS calls (except device calls) (Chapter 7)
System Loader calls (Chapter 8)
Part II
FST-specific information on GS/OS calls (Chapters 11–15)
Appendixes
ProDOS 16 calls (Appendix A)
FST-specific information on ProDOS 16 calls (Appendix B)

Chapter 11 **File System Translators**

This chapter describes how GS/OS is able to communicate with many different types of files and devices, in a manner that is transparent to the application. The operating system does this by supporting

- a generic GS/OS file interface (the abstract file system, described in Chapter 1) with which applications communicate
- individual file system translators (FSTs) that act as intermediaries between the GS/OS file interface and specific file systems and devices

This chapter discusses FSTs in general; the following chapters in Part II describe the individual FSTs supplied with GS/OS.

◆ *Note:* The file system translators in GS/OS handle both standard GS/OS (class 1) calls and ProDOS 16–compatible (class 0) calls. Only the standard GS/OS calls are described in this chapter and the rest of Part II; for information on how FSTs handle ProDOS 16–compatible calls, see Appendix B.

The FST concept

Every file system, such as ProDOS or Macintosh HFS, stores directories, subdirectories, files, and possibly other data structures on disk volumes in a format unique to that file system. Furthermore, each file system provides a slightly different set of system calls for accessing its files. The uniqueness of these data structures and system calls makes it very difficult for an application program that uses one file system to also access a volume created under another file system. Thus, application programs are nearly always written to run with one particular file system.

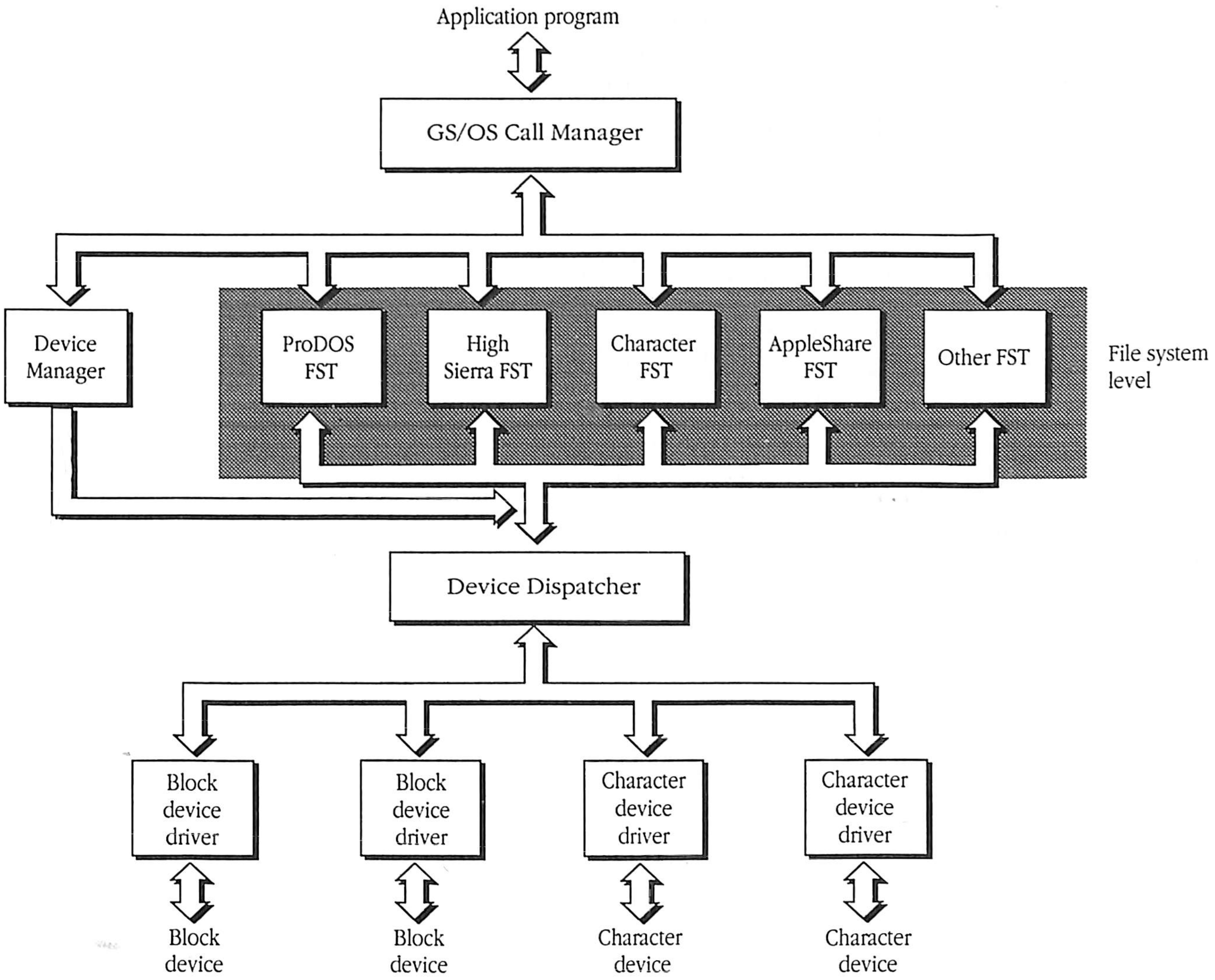
A **file system translator** (FST) is a GS/OS software module that accepts GS/OS calls made by applications and translates those calls into a form acceptable to the particular file system the FST supports. Likewise, the FST takes data read from a device and converts it to a form consistent with the generic GS/OS file interface (the abstract file system, described in Chapter 1). This makes it possible to write an application in which the same set of file I/O calls can access files on volumes created by any file system for which there is an FST. Application programs can thus transparently access files from any file system, using standard GS/OS system calls.

◆ *Note:* FSTs provide only the file access capabilities of GS/OS (see Chapter 4), which are similar to those of ProDOS 16. Because all FSTs use the same standard set of calls, they cannot implement all access capabilities and all calls for all file systems. Moreover, some FSTs cannot even support all of the capabilities provided by GS/OS. The High Sierra FST, for example, does not permit calls that write to disk.

Figure 11-1 shows the conceptual position of FSTs in the GS/OS hierarchy. They make up the file system level, which mediates between the GS/OS Call Manager at the application level and individual device drivers at the device level. When an FST receives a call, the call has already been processed by the GS/OS Call Manager. The FST either processes the call further and returns successfully or encounters an error condition and returns unsuccessfully with an error code. FSTs call the Device Dispatcher, which performs the actual I/O with calls to the device drivers. In addition, FSTs depend on various services provided by the Call Manager, such as pathname prefix management and error handling.

To GS/OS, all FSTs are equal. Any FST can be removed from the system by the user, and any FST can be added. The user adds or removes FSTs from GS/OS by moving FST files into or out of the subdirectory `SYSTEM/FSTS` on the boot disk.

■ **Figure 11-1** The file system level in GS/OS



Calls handled by FSTs

GS/OS calls can be classified by the part of the operating system that handles them. File calls are handled by FSTs, device calls are handled by the Device Manager, and other calls are handled by the GS/OS Call Manager itself. Table 11-1 lists all the GS/OS calls handled by FSTs.

- **Table 11-1** GS/OS calls handled by FSTs

Call no.	Call name	Call no.	Call name	Call no.	Call name
$2001	Create	$2010	Open	$2018	SetEOF
$2002	Destroy	$2012	Read	$2019	GetEOF
$2004	ChangePath	$2013	Write	$201C	GetDirEntry
$2005	SetFileInfo	$2014	Close	$2020	GetDevNumber
$2006	GetFileInfo	$2015	Flush	$2024	Format
$2008	Volume	$2016	SetMark	$2025	EraseDisk
$200B	ClearBackupBit	$2017	GetMark	$2033	FSTSpecific

As an application writer, you can expect that every FST will in some way support each of the calls listed in Table 11-1. Depending on the file system accessed, the call may be meaningful, it may do nothing and return no error, or it may do nothing and return an error. See the description of each FST for details.

All of the calls listed in Table 11-1 are described in Chapter 7, "GS/OS Call Reference," although only the generic FSTSpecific call is described in that chapter. FSTSpecific is a call whose function is completely definable by each FST. For example, the High Sierra FST (see Chapter 13, "High Sierra FST") uses the call to control file type emulation. FSTSpecific is described in detail for each FST that uses it, in the chapter that describes that FST.

Programming for multiple file systems

When you first write an application for GS/OS, it may seem strange not to know what file system your own application's files will be stored on. In reality, it makes your job simpler, but you may have to be careful in the beginning to avoid making some common incorrect assumptions.

Don't assume file characteristics

File-system independence is a cornerstone of the GS/OS design. To be most useful and efficient, and to avoid file system–specific problems, your application should also be as file system–independent as possible.

In general, you will be working with file information in the format returned by the GS/OS call GetFileInfo, rather than in the format of any real file system. For example, don't assume file-typing conventions other than the file type and auxiliary type provided in the GS/OS abstract file system; it is the job of each FST to translate that information into the file-type format for each file system.

Remember that different file systems use different block sizes. Don't simply assume that a block is 512 (or 256, or 520, or 1024) bytes; if you need to know the exact size of a block on a volume, use the GS/OS Volume call to the device holding that volume.

When you manipulate filenames and pathnames, observe the following guidelines:

- Don't assume any fixed limit on name length.
- Don't assume other restrictions, such as a limited ASCII character set.
- Always allow for the GS/OS pathname syntax: both colons and slashes are valid separators, and colons can *only* be separators. In addition, all eight bits of each byte of a pathname are significant. Using all eight bits of each byte may be particularly difficult for text-based applications, which have no way to force the standard Apple II character set to display characters such as sigma or the copyright symbol. You might want to use special typographical conventions for these special characters, or you might choose not to create files with such characters in their names. You could present the user with a list of existing filenames (with some substitution for the characters that are unavailable), thus providing the user with a method to retrieve such files.
- Don't preprocess pathnames, that is, pass user-entered pathnames directly to GS/OS with no application preprocessing. If you prevent users from entering non-ProDOS pathnames, it might help prevent illegal pathname syntax errors, but it also keeps users from creating files on non-ProDOS disks with anything but ProDOS pathname syntax. It also can keep them from accessing files on non-ProDOS disks that they created with another GS/OS application.

Detailed filename and pathname rules are presented in Chapter 1.

In general, go through the GS/OS file system level (by making standard GS/OS calls) as much as possible, rather than performing file-system-specific or device-specific operations that may require the presence of a particular FST, device driver, or device. Use the file-system independence and device independence of GS/OS to your own advantage.

Use GetDirEntry

If your program needs to catalog a volume, don't read directory files directly—that is, don't use the Read call to find out what is in a directory. GetDirEntry gives you the information in a standard format for all file systems, whereas with Read you need to know the exact format of a directory file for the specific file system you are accessing. And, because the files of interest may be in any of a number of file systems, it is far simpler to use GetDirEntry and let GS/OS take care of the details for you.

Don't build your own device list

Some applications construct a list of on-line devices only when they start up. This works fine if the list never changes, but under GS/OS new devices can be added dynamically during execution. Therefore, instead of constructing your own device list, scan the device list each time you need to use it. For example, use repeated DInfo or Volume calls with consecutive device numbers until an error is returned (such as an invalid device number), signaling that there are no more on-line devices.

Handle errors properly

Your application's normal error-handling routines may be adequate for processing errors under GS/OS, as long as you remember always to check for errors. A typical file-system-specific error might occur, for example, from attempting to save a file from a file system that normally allows saving, such as ProDOS, to a High Sierra disc. As long as your program is prepared to receive and act on any file error that GS/OS can generate, there should be no problem. Remember also that, because different file systems have different size limits on parameters, error $53 (parameter out of range) may be a very common occurrence.

When you receive a GS/OS error, you can use the Window Manager routine ErrorWindow to display the error message for that error. For more information about ErrorWindow, see the *Apple IIGS Toolbox Reference,* Volume 3.

On the other hand, you may needlessly restrict your application's capabilities if you assume an error will occur when it may not. For example, if your program is written assuming a read-only file system, it may unnecessarily prevent a user from saving a file to a different file system that is not read-only. In general, it is probably better to let GS/OS decide what file permissions and file calls are appropriate and then act on the returned errors if necessary.

Furthermore, what you do when an error occurs can be significant. For example, if a user attempts to save a very large file to a volume whose file system does not support the size of that file, your application should put up a directory dialog box to let the user save the data to another file system, rather than simply abort the save and lose the data.

Remember also that GS/OS allows access to character devices with file calls. Therefore, calls such as Read or SetMark may be applied to devices (like a printer) for which they have no meaning. Thus your error handling should allow not only for different file systems, but for completely different devices as well. In fact, it is common for character devices to return status information with error codes; if your file-access routines do not check for typical character-device errors, you may lose critical information.

Optimize file access

The file system translators written for GS/OS are designed to make file reads and writes as fast and efficient as possible. You may be able to read a file under GS/OS faster than you can under the file's native operating system. Furthermore, the disk caching available under GS/OS makes reading faster still.

As much as possible, consecutive file blocks are written to consecutive sectors on disk for fast access. More important, though, FSTs are optimized for large, multiblock transfers; for the application writer, this means that it is best to read and write data in chunks as large as possible. If you are interested in speed, try also to avoid newline read mode (which forces every character to be examined in turn) and the Flush call (which is slowed by the careful checking and updating it must perform).

For the fastest possible multiblock copying, use the GS/OS call BeginSession to defer block writes temporarily while copying, and then use EndSession to flush the cache when you are done copying. BeginSession and EndSession are most useful when doing multiple-file copies, because directory blocks are not written to disk as every file is copied. See the descriptions of BeginSession, EndSession, and SessionStatus in Chapter 7, "GS/OS Call Reference."

Present and future FSTs

GS/OS applications can read files from any file system for which there is an installed GS/OS file system translator. Currently, Apple Computer, Inc., defines the following file systems, each specified by its own file system ID. The list of potential FSTs is shown in Table 11-2.

■ **Table 11-2** File system IDs

File system ID	Description	File system ID	Description
$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E–$FFFF	Reserved
$0007	Lisa		

As new file systems are defined, Apple Computer assigns them unique file system IDs. In theory, then, all of the above file systems (and any future systems) can be accessed through GS/OS once FSTs are written for them. In practice, new FSTs will be created as dictated by demand and time constraints. The currently existing FSTs are described individually in subsequent chapters. Future releases of GS/OS will include file system translators for other file systems.

Disk initialization and FSTs

Disk initialization is a complex issue under an operating system that supports multiple file systems and many different types of devices.

For example, a system can be configured with several FSTs. A user might wish to write any one of the file systems on a disk. Also, a disk drive might support multiple low-level formatting styles.

Your application can use the GS/OS call Format or EraseDisk to initialize disks. The Format call formats the disk and writes out the new file system; the EraseDisk call simply writes out a new file system without formatting the disk. Either call causes GS/OS to put the initialization dialog box on the screen, allowing the user to select among valid file system and formatting choices (given the current system configuration of FSTs and device drivers). After the user makes the desired choices, the appropriate FST then initializes the disk as requested.

◆ *Note:* Because the initialization dialog box allows the user to cancel, it is probably not necessary for your application also to make the user confirm that a format or erasure is desired.

For both calls, the return parameter `fileSysID` indicates which file system (if any) the user chose. Format and EraseDisk are described in more detail in Chapter 7, "GS/OS Call Reference."

Chapter 12 **The ProDOS FST**

The ProDOS file system translator (ProDOS FST) provides a transparent application interface to the ProDOS file system. The ProDOS FST can access any block device whose GS/OS device driver can perform 512-byte block reads and writes. This chapter describes the ProDOS FST and shows only those aspects of the FST's call handling that are different from the descriptions in Chapter 7.

◆ *Note:* The file system translators in GS/OS handle both standard GS/OS (class 1) calls and ProDOS 16–compatible (class 0) calls. Only the standard GS/OS calls are described in this chapter and the rest of Part II; for information on how FSTs handle ProDOS 16–compatible calls, see Appendix B.

The ProDOS file system

The ProDOS file system is the native file system for most of the Apple II family of computers. All applications that run under either ProDOS 8 or ProDOS 16 create and read ProDOS files (if they create files at all).

The ProDOS file system is characterized by a hierarchical structure, 512-byte logical blocks, a 16 MB maximum file size, and a 32 MB maximum volume size. ProDOS files are either standard (sequential) files or directory files; no random-access, record-based file types are recognized as such by ProDOS.

ProDOS filenames can be up to 15 characters long. They can consist of the numerals `0–9`, the uppercase letters `A–Z`, and the period (`.`), in any combination (except that the first character must be a letter). A ProDOS volume name is like a filename but is preceded by a slash (`/`) or a colon (`:`). A ProDOS pathname consists of a sequence of slash-separated filenames, starting with a volume name.

The ProDOS file system is described in the *ProDOS 8 Technical Reference Manual* and the *Apple IIGS ProDOS 16 Reference.*

GS/OS and the ProDOS FST

The GS/OS abstract file system described in Chapter 1 is closely related to the ProDOS file system. Therefore, the ProDOS file system duplicates many features of the abstract file system exactly, and many GS/OS calls to the ProDOS FST behave exactly as described in Chapter 1. Here are the principal differences:

- ProDOS 8 and ProDOS 16 do not create or recognize extended files, which are equivalent to the resource forks of Macintosh files. However, the ProDOS FST under GS/OS can store and retrieve extended files in ProDOS format by defining a new storage type ($0005). It also can extend an existing file by setting the extend bit in the storage type ($8005) during a Create call.

  When a file is stored in this format, a GS/OS application can retrieve its resource fork and its data fork. Applications under ProDOS 8 and ProDOS 16, however, cannot access the file at all; attempts to open the file result in error $4B (unsupported storage type).

- Under GS/OS, a ProDOS pathname can have either slashes (/) or colons (:) as filename separators. The GS/OS Call Manager converts both types of separators to an internal format before passing on the pathname to the ProDOS FST.
- Because ProDOS files and volumes have maximum sizes smaller than those supported by GS/OS, parameters related to size (such as `EOF`, `position`, `blockCount`, `requestCount`, and `transferCount`) may not be accepted by the ProDOS FST if they are too large. In such cases the ProDOS FST returns error $53 (`paramRangeErr`).
- In GS/OS, the `fileType` field is 2 bytes long. Since the ProDOS file system can only handle a 1-byte auxiliary type, any value greater than $FF results in error $53 (`paramRangeErr`). If this error is returned by the ProDOS FST, validate the range of each parameter within the range limit listed in the ProDOS 8 manual.
- In GS/OS, the `auxType` field is 4 bytes long. Since the ProDOS file system can only handle a 2-byte auxiliary type, any value greater than $FFFF results in error $53 (`paramRangeErr`). If this error is returned by the ProDOS FST, validate the range of each parameter within the range limit listed in the ProDOS 8 manual.
- Because several file-entry fields in ProDOS directories on disk are smaller than their equivalent parameters in the GS/OS calls that access file entries, the high-order parts of some of those parameters are always zero when a file entry is read, and must also be zero when a file entry is stored. See the individual call descriptions in the following section "ProDOS FST Calls."

ProDOS FST Calls

The following sections describe how the ProDOS FST handles certain GS/OS calls differently from the general procedures described in Chapter 7. Calls listed in Table 12-1 are described in these sections. Calls not listed in the table are handled exactly as described in Chapter 7.

■ **Table 12-1** GS/OS calls handled differently by the ProDOS FST

Call number	Call name
$201C	GetDirEntry
$2006	GetFileInfo
$2005	SetFileInfo
$2033	FSTSpecific

GetDirEntry ($201C) for ProDOS FST

GetDirEntry returns file information contained in a volume directory or subdirectory entry. Under the ProDOS FST, the following fields have limitations different from the general values permitted by GS/OS:

`fileType` Only the low-order byte contains information.

`EOF` Only the three low-order bytes contain information.

`blockCount` Only the two low-order bytes contain information.

`auxType` Only the two low-order bytes contain information.

`optionList` After the required size words, the next word contains $0001 as the ProDOS FST ID. For more information about this parameter, see Chapter 6.

`resourceEOF` Only the three low-order bytes contain information.

`resourceBlockCount` Only the two low-order bytes contain information.

GetFileInfo ($2006) for ProDOS FST

GetFileInfo returns certain file attributes for an existing block file. Under the ProDOS FST, the following fields have limitations different from the general values permitted by GS/OS:

`fileType`	Only the low-order byte contains information.
`auxType`	Only the two low-order bytes contain information.
`storageType`	Only the low nibble of the low byte contains information.
`optionList`	After the required size words, the next word contains $0001 as the ProDOS FST ID. For more information about this parameter, see Chapter 6.
`EOF`	Only the three low-order bytes contain information.
`blocksUsed`	Only the two low-order bytes contain information.

SetFileInfo ($2005) for ProDOS FST

SetFileInfo assigns certain file attributes to an existing block file. Under the ProDOS FST, the following fields have limitations different from the general values permitted by GS/OS:

`fileType`	Only the low-order byte can be nonzero; otherwise, error $53 (`paramRangeErr`) is returned.
`auxType`	Only the two low-order bytes can be nonzero; otherwise, error $53 (`paramRangeErr`) is returned.
`optionList`	After the required size words, the next word contains $0001 as the ProDOS FST ID. For more information about this parameter, see Chapter 6.

FSTSpecific ($2033) for ProDOS FST

Description

FSTSpecific is a call that can be defined individually for any file system translator.

Parameters

This is the FSTSpecific parameter block:

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06		3	Subcall-specific parameter or parameters

The following parameters have particular values for this call.

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 3; maximum = 3.

`fileSysID` Word input value: File system ID of the FST to which the call is directed. For ProDOS, `fileSysID` = $0001.

`commandNum` Word input value: Number that specifies which particular subcall of FSTSpecific to execute, as follows:

$0001	SetTimeStamp
$8001	GetTimeStamp
$0002	SetCharCase
$8002	GetCharCase

See the individual subcall descriptions later in this chapter.

(subcall-specific) Word or longword input or result value: Depends on the specific subcall. See the individual subcall descriptions later in this chapter.

Errors

(none except general GS/OS errors)

SetTimeStamp (ProDOS FSTSpecific subcall)

Description The SetTimeStamp subcall allows the user to set the time stamp option.

Parameters This is the FSTSpecific parameter block for the SetTimeStamp subcall:

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 1)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`timeOption`	3	Word input value

The following parameters have particular values for this subcall.

`commandNum` For SetTimeStamp, `commandNum` = $0001..

`timeOption` Word input value: Specifies the time stamp option to use, as follows:

Comments This call affects the performance of the ProDOS FST. By default, the ProDOS FST time stamps all files and all subdirectories to the root level. This means that a change to a file in a subdirectory is reflected in the modification date of each parent subdirectory all the way to the root level of the volume.

GetTimeStamp (ProDOS FSTSpecific subcall)

Description The GetTimeStamp subcall allows the user to get the time stamp option.

Parameters This is the FSTSpecific parameter block for the GetTimeStamp subcall:

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	timeOption	3	Word input value

The following parameters have particular values for this subcall.

`commandNum` For GetTimeStamp, `commandNum` = $8001.

`timeOption` Word input value: Specifies the time stamp option to use, as follows:

$0000	Time stamp files only	(Fastest)
$0001	Time stamp files + parent directories	(Slower)
$0002	Time stamp files + all directories to the root	(Slowest)

SetCharCase (ProDOS FSTSpecific subcall)

Description The SetCharCase subcall allows you to change the way that the ProDOS FST saves and returns filenames. If you wish, the ProDOS FST can save the filename in uppercase and lowercase. A subdirectory name saved with lowercase letters may not be accessible under versions of ProDOS 8 older than 1.8; to avoid this problem, use version 1.8 or greater of ProDOS 8, or rename the subdirectory with all uppercase.

Parameters This is the FSTSpecific parameter block for the SetCharCase subcall:

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	caseOption	3	Word input value

The following parameters have particular values for this subcall.

`commandNum` For SetCharCase, `commandNum` = $0002.

`caseOption` Word input value: Specifies the case option to use, as follows:

$0000	Turn off upper/lowercase
$0001	Turn on upper/lowercase

GetCharCase (ProDOS FSTSpecific subcall)

Description The GetCharCase subcall allows you to get the current case settings.

Parameters This is the FSTSpecific parameter block for the GetCharCase subcall:

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 1)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	caseResult	3	Word output value

The following parameters have particular values for this subcall.

`commandNum` For GetCharCase , `commandNum` = $8002.

`caseResult` Word output value: Indicates the current case setting, as follows:

$0000	Lowercase is disabled
$0001	Lowercase is enabled

Chapter 13 **The High Sierra FST**

This chapter describes the GS/OS High Sierra file system translator (High Sierra FST). The High Sierra FST provides transparent application access to compact read-only optical discs (CD-ROM) and other media upon which High Sierra or ISO 9660–formatted files may reside.

The High Sierra and ISO 9660 file formats are not documented here. See the publications listed under "CD-ROM and the High Sierra/ISO 9660 Formats" in this chapter for more information. For information on the Apple extensions to ISO 9660, see Appendix C.

◆ *Note:* The file system translators in GS/OS handle both standard GS/OS (class 1) calls and ProDOS 16–compatible (class 0) calls. Only the standard GS/OS calls are described in this chapter and the rest of Part II; for information on how FSTs handle ProDOS 16–compatible calls, see Appendix B.

CD-ROM and the High Sierra/ISO 9660 formats

Compact discs provide a new and promising method of information storage and retrieval. Compact discs can hold vast amounts of information on a medium that is durable and inexpensive to manufacture. The information can be played back using existing, well-established technology based on CD music players.

A single CD-ROM disc holds about 550 megabytes of information. This large capacity is CD-ROM's main advantage, but it comes at a price. CD-ROM players have much slower access times than magnetic disk drives; it can take up to one second to find a byte of information on a CD-ROM disc, compared to less than a tenth of a second on a large-capacity hard disk.

CD-ROM's biggest disadvantage, however, is that—at present—its optical storage technology is read-only. Users can read from a CD, but they cannot write to it (hence the name CD-*ROM*).

The **High Sierra Group format** (named for the location of an ad hoc committee's original meeting place) and the **ISO 9660 format** (the International Standards Organization's version of High Sierra) are two nearly identical CD-ROM file formats that support the large files a compact disc can hold. They also attempt to minimize the penalties caused by slow access. Here are some of the highlights of the formats that are relevant to GS/OS:

- Logical sectors contain 2048 bytes (2 KB) of data. A logical sector can contain 1, 2, or 4 logical blocks.
- Files can contain data in any form or for any purpose; High Sierra/ISO 9660 specifies nothing about file contents.
- File identifiers can consist of three parts: a filename, a filename extension, and a version number. Directories have the filename part only. Under High Sierra, nondirectory files need one or more of the three parts (except that a file cannot be identified by a version number alone). Under ISO 9660, a nondirectory file must include all three parts.

  The filename is 0 or more characters (uppercase `A–Z`, digits `0–9`, or underscore); it must be followed by a period. The filename extension is 0 or more characters, and it must be followed by a semicolon. The version number is 1 to 6 digits. The sum of the filename and extension must be between 2 and 31 characters, including the period. Under ISO 9660, then, the smallest possible legal filename takes the form `A.;1` or `.A;1`.

◆ *Note:* See the section "Apple Extensions to ISO 9660," later in this chapter, for information on how to devise High Sierra/ISO 9660 filenames that can be translated to other file systems with different conventions.

- High Sierra/ISO 9660 is hierarchical; files may be placed in subdirectories. To speed access to files deep within subdirectories, a Path Table can be loaded into RAM for fast searching. The Path Table is an index to all subdirectories on disc. In addition, directory entries are kept small (and therefore easy to search fast) by putting auxiliary directory information—such as creation dates and access permissions—into extended attribute records (XARs), stored separately.
- Both ISO 9660 and High Sierra support associated files (equivalent to resource forks of GS/OS extended files); however, the High Sierra FST supports associated files for ISO 9660–formatted files only.
- High Sierra/ISO 9660 supports hidden files.

The High Sierra/ISO 9660 format from which Apple's High Sierra FST was designed is defined in these two documents:

- *Working Paper for Information Processing—Volume and File Structure of Compact Read-Only Optical Discs for Information Interchange,* published by the CD-ROM Ad Hoc Advisory Committee, May 28, 1986. This is the original High Sierra Group proposal.
- *ISO 9660: Information Processing—Volume and File Structure of CD-ROM for Information Interchange,* published by the International Standards Organization, first edition, 1988. This is the ISO 9660 standard, a slightly modified version of the High Sierra Group format.

◆ *Note:* Although High Sierra and ISO 9660 were developed specifically for compact disc storage, nothing in either format requires the files to be on CD-ROM. It is possible to have High Sierra/ISO 9660 files on any storage medium that can be formatted to accept them.

Limitations of the High Sierra FST

In translating file calls back and forth between CD-ROM drivers and GS/OS, the High Sierra FST cannot support all aspects of the High Sierra/ISO 9660 file system, nor can it meaningfully implement all GS/OS application calls. The High Sierra FST provided by Apple has the following features:

- It supports associated files (GS/OS extended files) for ISO 9660–formatted discs only.
- It permits only a single volume descriptor—the Standard File Structure Volume Descriptor—per physical volume.
- It does not support multivolume sets (named and logically linked groups of volumes occupying more than one disc).
- It does not support multi-extent files (files occupying more than one disc).
- It does not support random-access, record-based files; that is, it can read such files as streams of bytes, but it cannot access individual records directly.
- It maps the existence bit of the file flags into the invisibility bit of the GS/OS access word.
- It ignores the file permissions field in the extended attribute record.
- It is a read-only implementation.

This last limitation imposes strong restrictions on GS/OS system calls that write data to the disc. Those calls always return a write-protect error, after identifying that the file or device requested is present and is in High Sierra or ISO 9660 format.

In accessing files on a CD-ROM disc, remember that, under High Sierra or ISO 9660, block size is not fixed across volumes. If necessary, you can use the GS/OS Volume call to get the block size for a particular volume. Block counts returned by other calls are always in terms of blocks of the size returned by the Volume call.

An **associated file** in ISO 9660 is analogous to the resource fork of a GS/OS extended file. If an ISO 9660 file named `MyFile` has an associated file, the associated file has these characteristics:

- It is also named `MyFile` (its file identifier is identical).
- Its associated bit (in the file flags byte of the directory record) is set.
- Its directory entry resides immediately *before* the other `MyFile`'s directory entry.

Thus, GS/OS refers to the first `MyFile` (whose associated bit is set) as the resource fork of the extended file `MyFile`, and the immediately following `MyFile` (whose associated bit is clear) as the data fork of `MyFile`. Only data files can have associated files; directories cannot.

High Sierra/ISO 9660 does not provide an explicit file-typing convention. This can be a problem because many applications select a particular file type as a filter when calling the Standard File Tool Set to display files to the user. In such a case, files from a High Sierra/ISO 9660 disc would never be selectable.

To remedy this problem, the High Sierra FST, through the call FSTSpecific, defines and implements a convention by which High Sierra/ISO 9660 filenames can be used to convey file type information. See the discussion under "FSTSpecific," later in this chapter. In addition, Apple Computer, Inc., has defined a protocol that extends ISO 9660 to store file type and other information needed by either ProDOS or Macintosh HFS files. See the next section, "Apple Extensions to ISO 9660."

Apple extensions to ISO 9660

To facilitate the transformation of ProDOS files or Macintosh HFS files to ISO 9660 files on CD-ROM without loss of needed ProDOS or Macintosh file information, Apple Computer has defined a protocol that extends the ISO 9660 specification. Discs created using the Apple extensions are valid ISO 9660 discs. They retain the filename as well as the file type/auxiliary type (ProDOS) or file type/creator/bundle-bit/icon resource (Macintosh) information needed to reconstruct the original files from which they were made.

Because ISO 9660 does not provide for file typing and icons, the extra information is stored in a special data structure in the file's directory record. Filenames are preserved through transformations of ProDOS or Macintosh filenames to valid ISO 9660 names, and back again.

This section does not discuss the protocol in detail. Please see Appendix C, "Apple Extensions to ISO 9660," if you need to create or work with ProDOS or Macintosh files stored as ISO 9660 files. Here are the highlights of the protocol:

- **The protocol identifier:** The protocol identifier consists of 32 bytes in the `systemIdentifier` field of the Standard Volume Descriptor of an ISO 9660 volume. It consists of the characters `"APPLE COMPUTER, INC., TYPE: "` followed by 4 bytes of protocol flags. The current version of the type description gives the version number of the protocol and indicates whether the disc's files should be transformed to ProDOS filenames when read.

  The presence of the protocol identifier indicates that the Apple extensions have been applied to the disc's files.

- **The `SystemUse` field:** The `SystemUse` field in the file's directory record is an optional field. The Apple extensions use that field to store the extra file information. If the `SystemUse` field is present, and if it begins with the proper signature word, the subsequent information in the field can be interpreted as ProDOS or Macintosh HFS information.
- **ProDOS filename transformations:** If you (through an authoring tool) are creating ISO 9660 files from ProDOS files, you can transform ProDOS filenames to valid ISO 9660 filenames in such a way that users (through a receiving system) can access the files using their original ProDOS filenames. Take the following steps:
  1. Replace all periods in the ProDOS filename with underscores. If the file is a directory file, that completes the transformation.
  2. If the file is not a directory file, append the characters `.;1` to the filename. It is now a valid ISO 9660 filename.

  The receiving system performs the above transformation on user-supplied filenames before searching for them on disc and reverses the transformation before presenting filenames to the user.

  If the transformation is to be done, it must be applied to all files on a disc.
- **HFS filename transformations:** Unlike with ProDOS, it is not possible to make a simple, reversible transformation from all valid Macintosh HFS filenames to valid ISO 9660 filenames. To make the transformation as consistent as possible, however, Apple recommends the following guidelines:
  - □ Convert all lowercase characters to uppercase.
  - □ Replace all illegal characters, including periods, with underscores.
  - □ If the filename needs to be shortened, truncate the rightmost characters.
  - □ If the file is not a directory file, append the characters `.;1` to the filename.

  Such a transformation is not perfectly reversible, but its results are at least consistent across all files and discs.

High Sierra FST calls

Table 13-1 lists all the GS/OS system calls supported by the High Sierra FST. Those listed under *Meaningful* in the table perform meaningful tasks; the others always return an error (with the exception of Flush; see the call description later in this chapter).

■ **Table 13-1** High Sierra FST calls

Meaningful		**Not meaningful**	
$2006	GetFileInfo	$2001	Create
$2008	Volume	$2002	Destroy
$2010	Open	$2004	ChangePath
$2012	Read	$2005	SetFileInfo
$2014	Close	$2013	Write
$2016	SetMark	$2015	Flush
$2017	GetMark	$2018	SetEOF
$2019	GetEOF	$200B	ClearBackupBit
$201C	GetDirEntry	$2024	Format
$2020	GetDevNumber	$2025	EraseDisk
$2033	FSTSpecific		

With the exception of Flush, all the calls listed under *Not meaningful* in Table 13-1 do nothing and return error $2B (write-protected). Flush also does nothing, but it returns with the carry flag cleared (no error).

The following sections describe the calls listed as meaningful in Table 13-1 that the High Sierra FST handlles differently from standard GS/OS practice, as documented in Chapter 7. Other meaningful calls *not* described below are handled exactly as documented in Chapter 7. Refer also to Chapter 7 for complete explanations of the calls and parameters listed here.

GetFileInfo ($2006) for High Sierra FST

GetFileInfo returns certain attributes of an existing block file. The file may be open or closed.

Parameters

`access` The only possible values for this parameter under High Sierra/ISO 9660 are $01 (read-permission only) and $05 (read-permission only, file invisible).

`fileType` This output word value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file type mapping table. See Appendix C, "Apple Extensions to ISO 9660"; see also the call FSTSpecific, described later in this chapter.

`modDateTime` This output double longword value always has the same value as `createDateTime`.

`auxType` This output longword value is always $0000 unless the Apple extensions to ISO 9660 have been applied. See Appendix C.

`optionList` This is a longword input pointer to a data area to which results can be returned. If an Extended Attribute Record (XAR) exists for the file, the High Sierra FST returns the contents of the XAR in the data area pointed to. If an XAR does not fit in the allotted space, the High Sierra FST returns as much of the data as possible and generates error $4F (buffer too small).

Errors

In addition to the standard GS/OS GetFileInfo errors, the High Sierra FST can return these errors from a GetFileInfo call:

$4E	`invalidAccess`	access not allowed
$4F	`buffTooSmall`	buffer too small

Volume ($2008) for High Sierra FST

Given the name of a block device, Volume returns the name of the volume mounted in the device and other information about the volume.

Parameters

`freeBlocks` This longword output value is aways $0000.

`fileSysID` This word result value describes the file system of the volume being accessed. For High Sierra, `fileSysID` = $000B; for ISO 9660, `fileSysID` = $000C.

Open ($2010) for High Sierra FST

This call causes the FST to establish an access path to a file. Once an access path is established, the user may perform file reads and other related operations on the file.

A file can be opened more than once as long as it is not opened for write access, and each open assigns a different reference number. Because High Sierra/ISO 9660 files are read-only, it is always possible to have multiple open copies of a document.

Parameters

`requestAccess` If this word input parameter is included, and if its value is anything other than $0000 (use default permissions stored with file) or $0001 (read-access requested), the High Sierra FST returns error $4E (access not allowed).

`fileType` This word output value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file-type mapping table. See Appendix C, "Apple Extensions to ISO 9660"; see also the FSTSpecific call, described later in this chapter.

`auxType` This longword output value is always $0000 unless the Apple extensions to ISO 9660 have been applied. See Appendix C.

`modDateTime` This double longword output parameter always has the same value as `createDateTime`.

`optionList` This is a longword input pointer to a data area to which results can be returned. If an Extended Attribute Record (XAR) exists for the file, the High Sierra FST returns the contents of the XAR in the data area pointed to. If an XAR does not fit in the allotted space, the High Sierra FST returns as much of the data as possible and generates error $4F (`buffTooSmall`).

`fileType` This output word value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file type mapping table. See Appendix C, "Apple Extensions to ISO 9660"; see also the call FSTSpecific, described later in this chapter.

`auxType` This output longword value is always $0000 unless the Apple extensions to ISO 9660 have been applied. See Appendix C.

Errors

In addition to the standard GS/OS GetFileInfo errors, the High Sierra FST can return these errors from a GetFileInfo call:

$4E	`invalidAccess`	access not allowed
$4F	`buffTooSmall`	buffer too small

Read ($2012) for High Sierra FST

This call attempts to transfer the requested number of bytes, starting at the current mark, from a specified file into a specified buffer. The file mark is updated to reflect the number of bytes read.

The High Sierra FST allows applications to read directory files as well as data files (but only with standard GS/OS calls; ProDOS 16 Read calls to directories return error $4E—invalidAccess). Even so, as a reminder that directory structures differ for different file systems, the High Sierra FST always returns error $66 (FSTCaution) after a successful read of a directory.

Also, the High Sierra FST does not allow Read calls and GetDirEntry calls with the same file reference number: if an open file has previously been accessed by GetDirEntry, and a Read call is made with the same reference number, the High Sierra FST returns error $4E (invalidAccess). To avoid that error, open the directory twice.

Errors In addition to the standard GS/OS Read errors, the High Sierra FST can return these errors from a Read call:

$4E	invalidAccess	access not allowed
$66	FSTCaution	directory read successfully

GetDirEntry ($201C) for High Sierra FST

This call returns information about a directory entry in the volume directory or a subdirectory. Before executing this call, the application must open the directory or subdirectory. The call allows the application to step forward or backward through file entries or to specify absolute entries by entry number.

The High Sierra FST does not allow Read calls and GetDirEntry calls with the same file reference number. If an open file has previously been accessed by Read, and a GetDirEntry call is made with the same reference number, the High Sierra FST returns error $4E (`invalidAccess`). To avoid that error, open the directory twice.

Parameters

`fileType` This output word value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file-type mapping table. See Appendix C, "Apple Extensions to ISO 9660"; see also the FSTSpecific call, described later in this chapter.

`modDateTime` This double longword output parameter always has the same value as `createDateTime`.

`auxType` This longword output value is always $0000 unless the Apple extensions to ISO 9660 have been applied. See Appendix C.

`fileSysID` This word result value describes the file system of the directory being accessed. For High Sierra, `fileSysID` = $000B; for ISO 9660, `fileSysID` = $000C. If any other type of directory is accessed, the High Sierra FST returns error $52 (unsupported volume type).

`optionList` This is a longword input pointer to a data area to which results can be returned. If an Extended Attribute Record (XAR) exists for the file, the High Sierra FST returns the contents of the XAR in the data area pointed to. If an XAR does not fit in the allotted space, the High Sierra FST returns as much of the data as possible and generates error $4F (`buffTooSmall`).

Errors

In addition to the standard GS/OS Read errors, the High Sierra FST can return these errors from a GetDirEntry call:

$4F	`buffTooSmall`	buffer too small
$52	`unknownVol`	unknown volume type

FSTSpecific ($2033) for High Sierra FST

Description

FSTSpecific is a call that can be defined individually for any file system translator. The High Sierra FST uses the call FSTSpecific to control file type mapping. It simulates file types in High Sierra/ISO 9660 files (which do not have file types) by mapping filename extensions to specific GS/OS file types. FSTSpecific maintains a table in memory that controls which extensions correspond to which file types.

The default table contains only two entries; it equates filenames with extensions of `.txt` and `.bat` to GS/OS file type $04 (text file).

FSTSpecific uses a command number as one of its parameters and therefore functions as four different calls. The four calls are:

MapEnable	Enables or disables file type mapping
GetMapSize	Returns size, in bytes, of current file type map
GetMapTable	Returns the current file type map
SetMapTable	Replaces the current file type map

◆ *Note:* This mapping is independent of and unrelated to the file typing implemented by the Apple extensions to ISO 9660 described in Appendix C.

Parameters

This is the FSTSpecific parameter block:

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 3)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06		3	Subcall-specific parameter or parameters

`pCount` Word input value: Number of parameters in this parameter block. Minimum = 3; maximum = 3.

`fileSysID` Word input value: File system ID of the FST to which the call is directed. For High Sierra, `fileSysID` = $000B; for ISO 9660, `fileSysID` = $000C.

`commandNum` Word input value: A number that specifies which particular subcall of FSTSpecific to execute, as follows:

$0000	MapEnable
$0001	GetMapSize
$0002	GetMapTable
$0003	SetMapTable

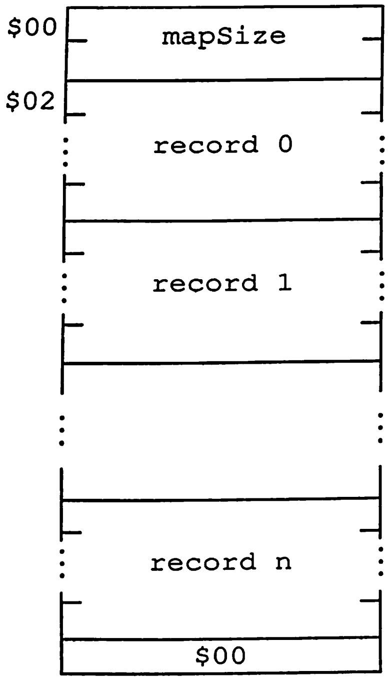
See the individual subcall descriptions later in this chapter.

(subcall-specific) Word or longword input or result value: Depends on the specific subcall. See the individual subcall descriptions later in this chapter.

Errors (none except general GS/OS errors)

What a map table is

The map table is the data structure that records which filename extensions are to be assigned to which file types. The format of a map table is as follows:



Length of table, including terminator

First map record (variable length)

Next map record

Last map record

Terminator (0 byte)

A map record consists of a text string (with MSBs off), followed by a 0 byte, followed by a file type byte. The text string can be any length and can include any legal characters for a High Sierra filename (text must be uppercase, for example). In APW assembly language, a map table can be defined as follows:

```
mapTable    dc    i2'end-mapTable+1'       ;Length of table.
            dc    c'.TXT',h'00 04'         ;Record 0.
            dc    c'.TYPE',h'00 7f'        ;Record 1.
end         dc    h'00'                    ;Terminator.
```

MapEnable (High Sierra FST FSTSpecific subcall)

The MapEnable subcall toggles file mapping on or off.

Parameters

This is the FSTSpecific parameter block for the MapEnable subcall:

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 3)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	enable	3	Word input value

The following parameters have particular values for this subcall.

commandNum For MapEnable, commandNum = $0000.

enable Word input value: Specifies whether file type mapping is enabled or disabled, as follows:

$0000 File type mapping disabled

$0001 File type mapping enabled

GetMapSize (High Sierra FST FSTSpecific subcall)

The GetMapSize subcall returns the size of the current file map.

Parameters This is the FSTSpecific parameter block for the GetMapSize subcall:

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 3)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`mapSize`	3	Word result value

The following parameters have particular values for this subcall.

`commandNum` For GetMapSize, `commandNum` = $0001.

`mapSize` Word result value: Indicates the size (in bytes) of the current map table.

GetMapTable (High Sierra FST FSTSpecific subcall)

The subcall GetMapTable returns a pointer to the current map table.

Parameters This is the FSTSpecific parameter block for the GetMapTable subcall:

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 3)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`bufferPtr`	3	Longword input pointer

The following parameters have particular values for this subcall.

`commandNum` For GetMapTable, `commandNum` = $0002.

`bufferPtr` A longword input pointer to a memory area large enough to hold the map table that will be returned by the call.

SetMapTable (High Sierra FST FSTSpecific subcall)

The subcall SetMapTable sets the current map table to the one pointed to by the input pointer.

Parameters This is the FSTSpecific parameter block for the SetMapTable subcall:

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 3)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`mapPtr`	3	Longword input pointer

The following parameters have particular values for this subcall.

`commandNum` For SetMapTable, `commandNum` = $0003.

`mapPtr` Longword input pointer to the new map table. As long as there is space in memory for the new table, it will replace the old one. If there is not enough space, error $54 (`outOfMem`) is returned and the original table remains in effect. No validity checking is done on the table.

Chapter 14 **The Character FST**

The Character file system translator (Character FST) provides a file-system-like interface to character devices such as the console, printers, and modems. The Character FST works with both generated and loaded drivers.

◆ *Note:* The file system translators in GS/OS handle both standard GS/OS (class 1) calls and ProDOS 16–compatible (class 0) calls. Only the standard GS/OS calls are described in this chapter and the rest of Part II; for information on how FSTs handle ProDOS 16–compatible calls, see Appendix B.

Character devices as files

The Character FST enables applications to read from and write to character devices as if they were files. That is, your application can open, read, write, and close a printer, modem, console, or other character device exactly as it performs those actions on a block device.

Not all GS/OS calls can be made to character devices, of course, and those that do may not always function exactly the same as for block devices. This chapter discusses those calls that do apply to character devices and notes any character device–specific features they have.

◆ *Note:* Although GS/OS lets you treat character devices as files in some ways, you cannot create, destroy, or rename character files with GS/OS calls. The system and the user control the existence and the names of character devices.

The Character FST allows multiple Open calls, with both read and write access, to a character file. Block-device FSTs, on the other hand, can allow multiple Opens for read access only.

Character FST calls

This section describes how the Character FST handles certain GS/OS calls differently from the general procedures described in Chapter 7. Calls listed in Table 14-1 are supported by the Character FST.

All other GS/OS calls return error $58 (`notBlockDev`).

The following descriptions explain how the Character FST responds to some of these calls differently from the standard procedures documented in Chapter 7. Any of the supported calls not described here function exactly as documented in Chapter 7.

■ **Table 14-1** GS/OS calls supported by the Character FST

Call number	Call name	Call number	Call name
$2010	Open	$2014	Close
$2012	Read	$2015	Flush
$2013	Write	$2011	Newline

Open ($2010) for Character FST

The Open call establishes an access path to a character file. With the `requestAccess` parameter, an application can request limited access rights to the character file.

Parameters

`pCount` Maximum value = 3. Unlike with block devices, you cannot use the Open call to a character device to get information normally returned by GetFileInfo.

`pathname` This input pointer must point to a character device name.

`requestAccess` The following values are allowed:

$00	open with available permissions
$01	open for read access only
$02	open for write access only
$03	open for both read and write access

Errors

In addition to the standard GS/OS Open errors, the Character FST can return these errors from an Open call:

$24	`drvrPriorOpen`	character device already open
$26	`drvrNoResrc`	resources not available
$28	`drvrNoDevice`	no device connected
$2F	`drvrOffLine`	device off line or no media present

Read ($2012) for Character FST

The Read call attempts to transfer the requested number of bytes from the specified character file into the application's data buffer.

Parameters `pCount` Minimum = 4; maximum = 4.

`cachePriority` Not used. Data transfers with character devices are not cached.

Errors In addition to the standard GS/OS Read errors, the Character FST can return these errors from a Read call:

$23	`drvrNotOpen`	character device not open
$2F	`drvrOffLine`	device off line or no media present

Write ($2013) for Character FST

The Write call attempts to transfer the requested number of bytes from the application's data buffer to the specified character file.

Parameters `pCount` Minimum = 4; maximum = 4.

`cachePriority` Not used. Data transfers with character devices are not cached.

Errors In addition to the standard GS/OS Write errors, the Character FST can return these errors from a Write call:

$23	`drvrNotOpen`	character device not open
$2F	`drvrOffLine`	device off line or no media present

Close ($2014) for Character FST

The Close call terminates access to the character file. Close also involves flushing the file (see the Flush call), to ensure that all data has been transferred before a character file is closed.

Errors In addition to the standard GS/OS Close errors, the Character FST can return these errors from a Close call:

$23	`drvrNotOpen`	character device not open
$2F	`drvrOffLine`	device off line or no media present

Flush ($2015) for Character FST

The Flush call completes any pending data transfer to the character file. If the character device is synchronous, all data transfer is by definition completed when the Write call returns, so the Flush routine simply returns with no error. If the device is asynchronous (for example, if it is interrupt-driven or has direct memory access), the Flush routine waits until all data has been transferred and then returns. If the file is multiply opened, all output access paths to the character file (not just the one with the specified `refNum`) are flushed.

Errors In addition to the standard GS/OS Flush errors, the Character FST can return these errors from a Flush call:

$23	`drvrNotOpen`	character device not open
$2F	`drvrOffLine`	device off line or no media present

Chapter 15 **The AppleShare FST**

The AppleShare FST is the implementation of AppleShare for GS/OS. It is meant to supersede AppleShare IIGS, the implementation of AppleShare for ProDOS 16. When GS/OS is running, GS/OS makes calls directly to the AppleTalk routines via the AppleShare FST.

The AppleShare FST works only with file servers that support AppleTalk Filing Protocol (AFP) version 2.0 or greater, such as the AppleShare File Server version 2.0.

This chapter describes the AppleShare FST and shows only those aspects of the FST's call handling that are different from the descriptions in Chapter 7.

Pathname syntax

There are two kinds of restrictions on pathname syntax: those imposed by GS/OS, and those imposed by the FST (because of naming restrictions in AFP).

GS/OS may impose a maximum length on pathnames, but does not impose a restriction on the **span** of a pathname. The span of a pathname is the maximum number of characters in a filename; that is, the maximum number of characters between pathname separators, including volume names.

The AppleShare FST does not impose a maximum length on pathnames, but restricts the maximum span to 31 characters. AFP volume names are normally less than 28 characters, but GS/OS does not check the extent of the span. A volume name with a length of 28–31 characters causes GS/OS to return error $45 (`volNotFound`).

GS/OS allows the slash (`/`) or the colon (`:`) to be a separator; the first slash or colon that appears in the pathname identifies the separator. A colon can never be used in a filename. A slash cannot be used in a filename if the separator is also a slash. The AppleShare FST does not permit a NULL byte in a pathname. All other characters are permitted.

◆ *Note:* The high bit of a character is significant. Characters with values greater than or equal to 128 are considered extended ASCII characters and are typically displayed as special symbols on Apple IIGS, Macintosh, and IBM systems.

A number that appears as the first filename in a partial pathname is assumed by GS/OS to be a prefix designator. Since numbers are valid filenames in AFP, a prefix designator should always be used explicitly with a partial pathname that begins with a number, as in the following example:

`0:555:Hello` is a valid pathname to the file named `Hello` in a folder named `555` in the volume corresponding to prefix designator `0`.

`555:Hello` causes an invalid path syntax error since GS/OS assumes that `555:` is a prefix designator for prefix `555`, which is invalid.

Macintosh and GS/OS file types

AppleShare file servers supporting AFP version 2.0 or greater maintain both Macintosh file type and creator information as well as the ProDOS file type and auxiliary type. Since each operating system has distinct file type information, a workstation can set one kind of file type for Macintosh and another for ProDOS.

The AppleShare FST uses the ProDOS file type and auxiliary type fields; it depends on the server to derive appropriate type information for Macintosh files. The AppleShare File Server version 2.0 does this using a convention also used by Apple File Exchange and the MPW IIGS cross-development tools.

ProDOS files are distinguished by a Macintosh creator of `pdos`.

The file type conversions are shown in Tables 15-1 and 15-2. If more than one rule applies, the one closest to the top of the table is used.

■ **Table 15-1** ProDOS-to-Macintosh file type conversion

ProDOS file type	**ProDOS auxiliary type**	**Macintosh creator**	**Macintosh file type**
$00	$0000	`pdos`	`BINA`
$B0 (SRC)	(Any)	`pdos`	`TEXT`
$04 (TXT)	$0000	`pdos`	`TEXT`
$FF (SYS)	(Any)	`pdos`	`PSYS`
$B3 (S16)	(Any)	`pdos`	`PS16`
$uv	$wxyz	`pdos`	$70 $uv $wx $yz*

*$70 is a lowercase *p*.

■ **Table 15-2** Macintosh-to-ProDOS file type conversion

Macintosh creator	**Macintosh file type**	**ProDOS file type**	**ProDOS auxiliary type**
(Any)	`BINA`	$00	$0000
(Any)	`TEXT`	$04 (TXT)	$0000
`pdos`	`PSYS`	$FF (SYS)	$0000
`pdos`	`PS16`	$B3 (S16)	$0000
`pdos`	$XYΔΔ*	$XY	$0000
`pdos`	$70 $uv $wx $yz†	$uv	$wxyz
(Any)	(Any)	$00	$0000

* Where X and Y are hex digits (that is, 0–9 or A–F), and Δ is a space.
† $70 is a lowercase *p*.

As Table 15-2 shows, the ProDOS file type `SYS` (= $FF) has a Macintosh file type of `PSYS`. The ProDOS file type `S16` (= $B3) has a Macintosh file type of `PS16`. The ProDOS unknown file type (= $00) has a Macintosh file type of `BINA`. ProDOS text files (`TXT` = $04) with an auxiliary type of $0000 (that is, normal ASCII text, no records) has a Macintosh file type of `TEXT`. These special cases allow Macintosh to display unique icons for these file types.

Macintosh files with creator `pdos` and a file type of the form $XY (where XY are any two hex digits followed by two spaces) will get the ProDOS file type $XY and auxiliary type $0000. Macintosh files with creator `pdos` and a file type of the form $70uvwxyz ($70 is a lowercase *p*) have the ProDOS file type $uv and auxiliary type $wxyz (note the order of the bytes: on the Macintosh they are stored high–low instead of low–high).

APW source files that have the ProDOS file type $B0 are given the Macintosh file type `TEXT` so that they can be edited more easily.

◆ *Note:* The conversions shown in Tables 15-1 and 15-2 do not necessarily translate back and forth. That is, if you convert a ProDOS file type and auxiliary type to a Macintosh creator and type as shown in Table 15-1, and then convert it back to a ProDOS file type and auxiliary type as shown in Table 15-2, you do not always get the original ProDOS file type and auxiliary type. For example, the auxiliary types for SYS and S16 files are reset to $0000, and SRC (APW source) files are changed to TXT (plain text) files. This situation typically occurs when a file is copied from one server to another using a Macintosh.

Access privileges

In a shared-file environment, you must decide what use is going to be made of the contents of your application's files, and open those files in a way that doesn't interfere with other users' access to those files. The standard GS/OS Open call allows you to specify the access you would like to have to the file in the `requestAccess` parameter, as described in Chapter 4. However, the access returned by the Open call indicates only the access you would get to the file under the best possible conditions. The actual access you get when opening the file is controlled by the following factors:

- your setting of the `requestAccess` parameter in the Open call
- the access privileges to ancestor and parent directories, set either by your application or by other applications using by the FSTSpecific call SetPrivileges

- access restrictions (called *deny modes*) imposed either by your application or by other applications using the FSTSpecific call SpecialOpenFork
- the access to the file and folder, as set by the `access` parameter in the GetFileInfo and SetFileInfo calls

The actual access you get to the file is explained in the following sections.

If you specify a `requestAccess` parameter of $0001, $0002, or $0003

△ **Important** Try to use the least restrictive `requestAccess` parameter when possible; for example, only ask for read if that is all you will do with the file. △

If you set the `requestAccess` parameter to $0001 (read-only), the file is opened as read-only, deny write. Multiple users can read the file, but no user can write to the file.

If it is $0002 (write-only), the file is opened as write-only, deny read/write. No other user can open the file. For example, you might want to do this when writing to a log file or when copying files.

If it is $0003 (read/write), the file is opened as read/write, deny read/write. No other user can open the file.

If the file cannot be opened with the requested mode, error $4E (`invalidAccess`) is returned.

◆ *Note:* If you want to open a file with permissions different from these, use the FSTSpecific subcall SpecialOpenFork, documented later in this chapter. SpecialOpenFork is essentially the same as the Open command, but allows you to control the access that other users have while you have the file open.

By default, GS/OS turns buffering on for files or directories when the Open call is made. The buffer is not filled until the first Read or GetDirEntry call is made, so that your application can turn buffering off after the Open call but before the first read.

Directories with neither `seeFiles` nor `seeFolders` access rights cannot be opened (since the only valid operation on an open directory is GetDirEntry), and error $4E (`invalidAccess`) is returned.

With a standard GS/OS Open call, all of the parameters after the resource number are file information. If you use a standard GS/OS Open call with the `pCount` parameter set to greater than 4 —that is, you are asking for file information to be returned—and you don't have privileges to see the object you are opening (if the object is in a drop box, for example), the call returns error $4E (`invalidAccess`), since you don't have access to the file information you requested.

If you specify a `requestAccess` parameter of $0000

If you specify a `requestAccess` parameter of $0000 (as permitted), GS/OS takes the following actions:

1. GS/OS attempts to open the file as read/write, deny read/write.
2. If this fails, an attempt is made to open the file as read-only, deny write.
3. If this fails, an attempt is made to open the file as write-only, deny read/write.
4. If this also fails, error $4E (`invalidAccess`) is returned.

△ **Important** Because the AppleShare FST tries all of these combinations, you usually don't want to use a standard GS/OS Open call with a `requestAccess` parameter equal to $0000. If you do use that parameter, you won't know what access you really got to the file until you actually try to read or write to the file.

Constructing multi-user applications

A multi-user application allows several users to access and possibly change common data at the same time. A typical example is a database program that lets several users view and edit records at the same time. In this case, the read/write protections are applied to individual records instead of the entire file.

To apply read/write protections to individual records, you use AppleShare FSTSpecific calls to take the following steps:

1. Use the AppleShare FSTSpecific subcall SpecialOpenFork to open the appropriate fork of the file. With this call you not only provide the access you want to the file, but the access you will allow others to the file. For example, a database file might be opened for read/write, deny nothing. This way, all users can open the file and read and write to it at the same time.

◆ *Note:* Buffering is disabled by default for the SpecialOpenFork call to prevent inconsistencies between the contents of the buffer and the file.

2. Use the FSTSpecific subcall ByteRangeLock to prevent one workstation from writing to the file and corrupting information being read or written by another workstation. The ByteRangeLock subcall takes an open file reference number, some flags, an offset into the file, and a length specifying the number of bytes to be locked or unlocked. When a range of bytes is locked, no other workstation can read or write those bytes; in fact, the same workstation using a different reference number cannot access those bytes.

For example, to add a new record to a database, you would take the following steps:

1. Call SpecialOpenFork to open the appropriate fork of the file and specify read/write access, but do not deny other users read/write access to the file (you will be locking the appropriate range of bytes).
2. Call ByteRangeLock to lock the header of the file and read it in to determine where to place the new record.
3. Call ByteRangeLock to lock the range where the new record will be located.
4. Update the header to indicate that the new record has been allocated, write out the header, and call ByteRangeLock to unlock it.
5. Write the new record to the range you have locked, and call ByteRangeLock to unlock the range.

Remember that you should have locked any range of bytes that you are reading or writing, and that you should reread a range of bytes if you have unlocked and locked it again.

Interrupts and AppleTalk calls

If interrupts are disabled when the FST has to make an AppleTalk call, GS/OS returns error $27 (`drvrIOError`) instead of making the call. In most cases, this error is propagated back to the user. Note that some calls (such as GetMark) may not require an AppleTalk call to be made and will complete correctly with interrupts disabled; other calls (such as Read and Write with small request counts, or GetDirEntry) might not complete with interrupts disabled (depending on the current mark, any data that is buffered, and so on).

△ **Important** Do not make system calls with interrupts disabled. △

Using the option list

This is the `optionList` parameter block:

Offset		Size and type
$00	`bufferSize`	Word input value (minimum = 5)
$02	`dataSize`	Word input value
$04	`fileSysID`	Word input value
$06	`finderInfo`	32 bytes
$26	`parentDirID`	Longword input value
$2A	`accessRights`	Longword result value

`bufferSize` Word input value: Specifies the size of the buffer. For AppleShare, the length must be greater than or equal to $002E.

`dataSize` Word input value: Specifies the length of the parameter block. For AppleShare, the length is $002A.

`fileSysID` Word input value: The file system ID of the FST to which the call is directed. For AppleShare, the `fileSysID` parameter must always be $000D.

`finderInfo` 32 bytes: The Finder information for a file is described in *Inside Macintosh.*

`parentDirID` Longword input value: ID of the parent directory.

`accessRights` Longword result value: For directories, this field is in the same format as that used in the GetPrivileges and SetPrivileges calls. For files, the field is set to all 0's.

◆ *Note:* The `accessRights` field was included to allow applications like the Finder to determine what access a user has to a folder without having to do a separate GetPrivileges call.

Controlling directory and file buffers

When buffering is off, each GetDirEntry call immediately causes an enumerate of one entry from the server. When a GetDirEntry call is made with buffering on, the requested entry is returned from the buffer if possible. Otherwise, the buffer is filled with as many entries from the server as possible, including the requested entry; then the requested entry is returned.

The buffer is not prefilled when the folder is opened. The number of entries kept in the buffer is variable and depends on the size of the long and short names of the files and directories.

- When buffering is off, every Read and Write call transfers data directly between the user's data buffer and the server.
- When buffering is on, and a read or write larger than the buffer size is made, any unwritten data in the buffer is written and the read/write is made directly between the user's data buffer and the server.
- When buffering is on and a read or write smaller than the buffer size is made, the block containing the first byte to be read or written is read into the buffer. If the block was already in the buffer, no read is done. If a different block is in the buffer, any unwritten data is written and the new block is read into the buffer. The read or write then proceeds to the end of the buffer. If the read or write extends past the end of the buffer, any unwritten data is written and the next block is read into the buffer. The read or write then is completed by reading or writing between the buffer and the server.

Unbuffered reads with zero or one newline characters are handled directly by the server (that is, the read to the server requests the same number of bytes as the user requested). Unbuffered reads with two or more newline characters are read one character at a time from the server (until a newline is encountered or all bytes have been read or the EOF is reached); because this takes considerable time, you usually should not read two or more newline characters with buffering off.

Buffered reads with one or more newline characters become reads of one block at a time. Each block is read into the buffer and the bytes are then copied to the user's data buffer one at a time (while being compared against all the newline characters). Buffered reads with no newline characters are as described earlier in this section.

ProDOS 16 and ProDOS 8 compatibility

The AppleShare FST is compatible with the ProDOS 16 and ProDOS 8 implementations of AppleShare. All calls added to the ProDOS 8 MLI to support AppleShare are still usable from ProDOS 8. The `RamDispatch` vector at $E11014 continues to support full native-mode calls from either ProDOS 8 or GS/OS. In addition, device-specific calls for the AppleShare FST allow uniform access to the AppleShare file-system features such as byte range locking.

The ProDOS 16–compatible `OPEN` call works in the same fashion as the `OPEN` call for AppleShare for ProDOS 16; for more information on the ProDOS 16–compatible `OPEN` call, see Appendix A.

△ **Important** Use the standard GS/OS Open call whenever possible, since that call allows the read/write access to be established. △

The AppleShare FST reports errors $46 (`fileNotFound`) and $44 (`pathNotFound`) correctly; that is, when a file is not found, the FST checks for the existence of the parent and issues error $44 (`pathNotFound`) if the parent does not exist. This differs from ProDOS 16, which reported the `pathNotFound` error for both error conditions.

Calls to the AppleShare FST

The following sections describe how the AppleShare FST handles certain GS/OS calls differently from the general procedures described in Chapter 7. Calls not listed in these sections are handled exactly as described in Chapter 7.

Create ($2001) for AppleShare FST

The ProDOS file type and auxiliary type are set to the values given in the call; by default, the Macintosh creator is set to `pdos` and the Macintosh file type is derived according to the rules given in the section "Macintosh and GS/OS File Types" earlier in this chapter. All files are created as extended files (that is, they have both a data and a resource fork), since there is no way to distinguish between a fork of length 0 and a fork that does not exist.

In a standard GS/OS Create call, the `EOF` and `resourceEOF` parameters are ignored. Because the definition of the call states that the forks' EOFs are set to 0, it is impossible with AFP to allocate space in a fork past its EOF.

Only the low byte of the file type and the low word of the auxiliary type are used. If the high byte of the file type or the high word of the auxiliary type is nonzero, error $53 (`paramRangeErr`) is returned.

SetFileInfo ($2005) for AppleShare FST

The ProDOS file type and auxiliary type are set to the values specified in the call; by default, the Macintosh creator is set to `pdos` and the Macintosh file type is derived according to the rules given in the section "Macintosh and GS/OS File Types" earlier in this chapter.

The `optionList` parameter only uses the `finderInfo` field (the other fields cannot be set); any data past the `finderInfo` field is ignored.

If the `fileSysID` parameter is not the same as AppleShare's file system ID ($0D), then the `optionList` parameter is ignored. All FSTs return their file system ID in the first word of the `optionList` parameter and ignore that parameter if the `fileSysID` parameter does not match theirs. Thus, your application can always get and set the `optionList` parameter as part of the copying process, even when it is copying from one file system to another.

Only the low byte of the file type and low word of the auxiliary type are used. If the high byte of the file type or high word of the auxiliary type is nonzero, error $53 (`paramRangeErr`) is returned.

GetFileInfo ($2006) for AppleShare FST

Directories with neither `seeFiles` nor `seeFolders` access will have the read bit in their access word cleared; files and directories with `seeFiles` or `seeFolders` access have their read bit set. If the file's resource fork is not empty, the `storageType` parameter is returned as $05 (extended); otherwise it is returned as $01, $02, or $03 (seedling, sapling, or tree), depending on the length of the data fork.

Open ($2010) for AppleShare FST

Directories with neither `seeFiles` nor `seeFolders` access have the read bit in their access word cleared; files and directories with `seeFiles` or `seeFolders` access have their read bit set. If the file's resource fork is not empty, the `storageType` parameter is returned as $05 (extended); otherwise it is returned as $01, $02, or $03 (seedling, sapling, or tree), depending on the length of the data fork.

The standard GS/OS Open call used with the AppleShare FST prevents one user from writing data that another user is reading, but does not allow multiple users to read a file without explicitly asking for read-only access. The actions that GS/OS takes depend on the value specified for the `requestAccess` parameter, as detailed in the section "Access Privileges" earlier in this chapter.

Read ($2012) for AppleShare FST

The Read call is not supported for directories, and an attempt to read a directory returns error $4E (`invalidAccess`). ProDOS directories will not be synthesized. Use the GetDirEntry call to enumerate directories.

If you attempt to read part of the range that has already been locked by another workstation, GS/OS returns error $4E (`invalidAccess`) and sets the transfer count to indicate the number of bytes transferred before the locked range was encountered.

◆ *Note:* If you get this error because of a locked range, don't assume that the locked range starts immediately after the bytes that were already transferred into your buffer.

Regardless of the value in the cache priority field, data is not put in the system cache. By default, the FST maintains a buffer containing the last block that was read from or written to. This block buffer can be controlled on a file-by-file basis by the FSTSpecific call BufferControl.

If buffering is disabled and newline mode has been enabled with more than one newline character, the read is completed one byte at a time because the server's newline mechanism provides for only one newline character.

△ **Important** Reading a file with buffering disabled and more than one newline character imposes tremendous overhead; avoid it if at all possible. △

Write ($2013) for AppleShare FST

Regardless of the value in the cache priority field, data will not be put in the system cache. By default, the FST maintains a block buffer containing the 512 bytes of the block that contains the current mark. This block buffer can be controlled on a file-by-file basis by the FSTSpecific call BufferControl.

Writes to directories are not allowed; GS/OS returns error $4E (`invalidAccess`).

Close ($2014) for AppleShare FST

In response to a Close call, the file is always closed and the reference number for that file is always invalidated, even if there is an error. This is because any error an application gets may not be correctable by the application or the user. For example, the data to be flushed before the close might be locked by another workstation, or the connection with the server might be lost. In such situations, GS/OS indicates that the error occurred and cleans up the system as much as possible.

SetEOF ($2018) for AppleShare FST

If a fork is extended, the additional bytes that are allocated might not all be set to 0.

In a standard GS/OS SetEOF call, if the base indicates that the EOF should be set to EOF minus displacement, the server's current EOF is determined and the EOF is set in relation to that. This may be different from the workstation's assumption of the EOF if another workstation has modified the fork's EOF. This may also delete data that another workstation has written between the time when the current EOF was determined and the time when the new EOF was set.

GetEOF ($2019) for AppleShare FST

The fork's EOF is determined from the server; this may not match the workstation's assumption of the EOF if another workstation has modified the fork's EOF. Note that another workstation could change the EOF after completion of this call, making the results inaccurate.

GetDirEntry ($201C) for AppleShare FST

GetDirEntry is not supported for files. It returns the error $4E (invalidAccess).

Directories enumerated by GetDirEntry that have neither seeFiles nor seeFolders access will have the read bit in their access word cleared. Directories with seeFiles or seeFolders access will have the read bit set.

The FST internally maintains the directory entry number (the entryNum parameter) to allow forward and backward scanning of the directory. By default, several entries are buffered for better performance (this can be disabled by using the FSTSpecific call BufferControl). Error $61 (endOfDir) is returned when an entry is requested that does not exist in the buffer (or when buffering is disabled for the directory), and that entry cannot be read from the server.

Since AppleShare is a shared-file system, the entry number may change for a file even while the directory is being scanned, because other users can add or delete files in the directory. Also, if the base and displacement fields are both 0, the total number of entries is returned.

◆ *Note:* Because other users can create and delete files while you are enumerating the directory, more or fewer entries may actually be returned if the directory is enumerated.

The best way to enumerate a directory is to open the directory and make successive GetDirEntry calls with base and displacement both set to $0001. When you get error $61 (endOfDir), you are finished enumerating, and should remove duplicate entries from your list.

ReadBlock ($0022) for AppleShare FST

This call returns error $88 (`appleshareNetError`) for AppleShare devices, in order to be compatible with System Disk 3.2. Remember, the preferred method for identifying a network volume is to make a Volume call and check if the `fileSysID` parameter = $0D.

WriteBlock ($0023) for AppleShare FST

This is an invalid operation for an AppleShare device. This call always returns an error. The current error code is error $4E (`invalidAccess`). This is different from error $88 returned under version 3.2, and it may change in the future.

Format ($2024) for AppleShare FST

This is an invalid operation for an AppleShare device. This call always returns an error. The current error code is $2B (`drvrWrtProt`). This is different from error $88 returned under version 3.2, and it may change in the future.

EraseDisk ($2025) for AppleShare FST

This is an invalid operation for an AppleShare device. This call always returns an error. The current error code is $2B (`drvrWrtProt`). This is different from error $88 returned under version 3.2, and it may change in the future.

GetBootVol ($2028) for AppleShare FST

If GS/OS is booted over AppleTalk, this command returns the name of the user volume on the server the user logged onto during booting. All system files should be present on this volume, just like any other boot volume.

GetFSTInfo ($202B) for AppleShare FST

The `fileSysID` parameter is returned as $0D (AppleShare). The attribute parameter is returned as $0000 (System Call Manager should not put pathnames all in uppercase, do not clear high bits of pathname, this is a block FST, formatting is not supported). The `blockSize` parameter is currently returned as 512; this value is useful only in determining the number of bytes used, the number of bytes free, and the total number of bytes on a volume (since these values are given in blocks).

FSTSpecific ($2033) for AppleShare FST

The FSTSpecific call can be defined individually for any file system translator.

Description The Appleshare FST uses the `commandNum` parameter of the FSTSpecific call to make several different calls. The calls, their functions, and their command numbers are shown in Table 15-3.

Parameters This is the FSTSpecific parameter block:

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 3)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06		3	Subcall-specific parameter or parameters

`pCount` Word input value: The number of parameters in this parameter block. The number varies for each subcall, as described later in this chapter.

`fileSysID` Word input value: The file system ID of the FST to which the call is directed. For AppleShare, the `fileSysID` parameter must always be $000D.

`commandNum` Word input value: A number that specifies which particular subcall of FSTSpecific to execute, as shown in Table 15-3. For details about each subcall, see the individual subcall descriptions later in this chapter.

(subcall-specific) One or more word or longword input or result values: These depend on the specific subcall; see the individual subcall descriptions later in this chapter.

Errors

$45	`volNotFound`	a pathname specifies a volume name that does not match any mounted AppleShare volume (even if a volume by that name exists for a different file system)
$52	`unknownVol`	a reference number for a file opened by another FST was used, or a pathname used a device name for a device other than an AFP (AppleShare) driver
$53	`paramRangeErr`	command number out of range
$54	`outOfMem`	out of memory

■ **Table 15-3** AppleShare FSTSpecific subcalls

Subcall	commandNum	Description
(Invalid number)	$0000	
BufferControl	$0001	Turns buffering on or off for a specified file or directory
ByteRangeLock	$0002	Locks a specified range of bytes
SpecialOpenFork	$0003	Opens a specified fork of a file, allows you to specify the type of access you want to the file, and allows you to specify the access you will allow others to the file
GetPrivileges	$0004	Retrieves the access privileges of a specified directory
SetPrivileges	$0005	Sets the access privileges for a specified directory
UserInfo	$0006	Returns the user name and primary group name of a user
CopyFile	$0007	Causes a file on a server to be copied by the server
GetUserPath	$0008	Returns a pointer to a GS/OS string containing the pathname of the user's directory on the user volume
OpenDesktop	$0009	Returns a desktop reference number
CloseDesktop	$000A	Frees all resources allocated when a specified desktop reference number was opened
GetComment	$000B	Returns the comment associated with a specified file, directory, or volume
SetComment	$000C	Specifies the comment for a specified file, directory, or volume
GetSrvrName	$000D	Returns the server name and zone name for a specified volume
(reserved)	$000E–$FFFF	

BufferControl (AppleShare FSTSpecific subcall)

Description

This subcall turns buffering on or off for a specified file or directory.

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 3)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`refNum`	3	Word input value
$08	`flags`	4	Word input value

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 3; maximum = 4.

`commandNum` For BufferControl, `commandNum` = $0001.

`refNum` Word input value: The reference number of a file or directory whose buffering is to be enabled or disabled (reference number $0000 is invalid).

`flags` Buffer Disable flags (default = $0000)

Bit 15	set	Disable buffering
	clear	Enable buffering
Bits 0–14		reserved

Errors

$43	`invalidRefNum`	invalid reference number

ByteRangeLock (AppleShare FSTSpecific subcall)

Description This subcall locks a specified range of bytes to prevent one workstation from writing to the file and corrupting information being read or written by another workstation.

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 7)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	refNum	3	Word input value
$08	lockFlag	4	Word input value
$0A	lockOffset	5	Longword input value
$0E	lockLength	6	Longword input value
$12	lockStart	7	Longword output value

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 7; maximum = 7.

`commandNum` For ByteRangeLock, `commandNum` = $0002.

`refNum` Word input value: The reference number of the file to lock.

`lockFlag` Determines the lock status and the type of offset for the file, as follows:

Bit 15	set	Lock range
	clear	Unlock range
Bit 14	set	Offset relative to EOF
	clear	Offset relative to start of file
Bits 0–13		reserved

For this parameter, the following constants can be combined:

`lockRange` = $8000
`relativeToEOF` = $4000

`lockOffset` Longword input value: The offset into the file (may be negative if relative to the end of the file).

`lockLength` Longword input value: Length of the range to be locked.

◆ *Note:* You can lock a range past the EOF of the file to extend the size of the file.

`lockStart` Longword output value: Actual start of the locked range (in relation to the beginning of the file) as returned by the server.

Errors

$43	`invalidRefNum`	invalid reference number
$4D	`outOfRange`	position out of range; user already has some or all of range already locked, or is unlocking a range not locked by that user
$4E	`invalidAccess`	access denied; some or all of range has been locked by another user
$53	`paramRangeErr`	invalid parameter

SpecialOpenFork (AppleShare FSTSpecific subcall)

Description

This subcall opens a specified fork of a file, allows you to specify the type of access you want to the file, and allows you to specify the access you will allow others to the file.

◆ *Note:* Buffering is disabled by default for the SpecialOpenFork call to prevent inconsistencies between the buffer's and the file's contents.

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 5)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`refNum`	3	Word output value
$08	`pathPtr`	4	Longword input pointer
$0C	`accessWord`	5	Word input value
$0E	`forkNum`	6	Word input value

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 5; maximum = 6.

`commandNum` For SpecialOpenFork, `commandNum` = $0003.

`refNum` Word output value: The reference number returned by GS/OS to the access path. Use this reference number as you would a reference number returned by an Open call.

`pathPtr` Longword input pointer: Pointer to GS/OS string representing the pathname of the file to be opened.

`accessWord` Word input value: The access mode giving the read/write permissions desired and to be denied to others. If the bit is set, the condition is asserted.

Bit 0	Request read access
Bit 1	Request write access
Bits 2, 3	Reserved
Bit 4	Deny read to others
Bit 5	Deny write to others
Bits 6–15	Reserved

◆ *Note:* This parameter has the same meaning as in the ProDOS 8 SpecialOpenFork command.

`forkNum` Word input value: Resource number, as follows (default $0000):

$0000	Causes the data fork to be opened
$0001	Causes the resource fork to be opened

Errors

$4E	`invalidAccess`	access denied; file is being read by another user

GetPrivileges (AppleShare FSTSpecific subcall)

Description This subcall retrieves the access privileges of a specified file or directory.

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 4)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	pathnamePtr	3	Longword input pointer
$0A	accessRights	4	Longword output value
$0E	ownerNamePtr	5	Longword input pointer
$12	groupNamePtr	6	Longword input pointer

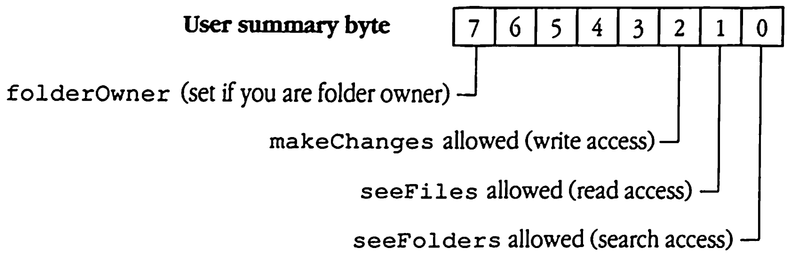
The following parameters have particular values for this subcall.

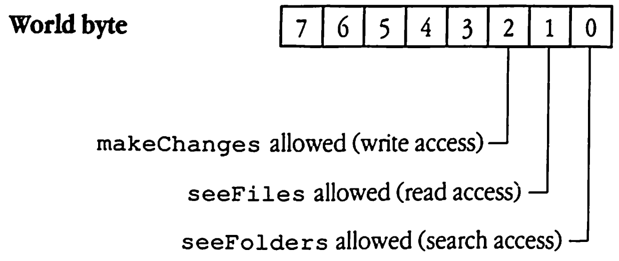
`pCount` Word input value: The number of parameters in this parameter block. Minimum = 4; maximum = 6.

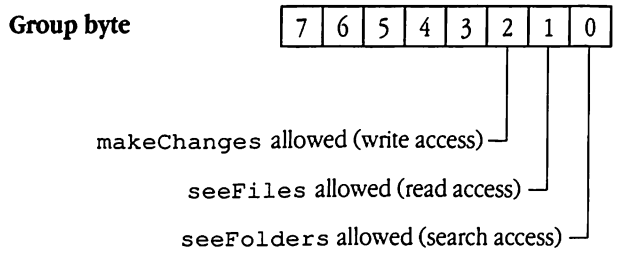
`commandNum` For GetPrivileges, `commandNum` = $0004.

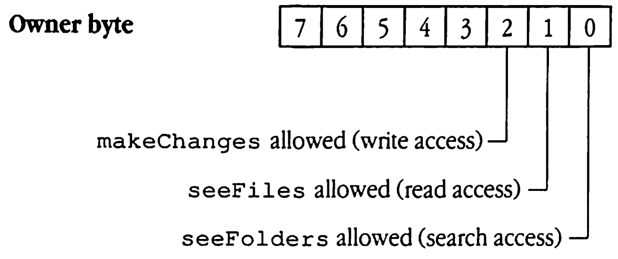
`pathnamePtr` Longword input pointer: Pointer to GS/OS string that contains the pathname of the directory whose access privileges are to be retrieved.

accessRights Longword output value: Four bytes that define the access privileges. For each of these bytes, bit 0 is search access (see folders), bit 1 is read access (see files), and bit 2 is write access (make changes).


User summary byte
7
6
5
4
3
2
1
0
folderOwner (set if you are folder owner)
makeChanges allowed (write access)
seeFiles allowed (read access)
seeFolders allowed (search access)


World byte
7
6
5
4
3
2
1
0
makeChanges allowed (write access)
seeFiles allowed (read access)
seeFolders allowed (search access)


Group byte
7
6
5
4
3
2
1
0
makeChanges allowed (write access)
seeFiles allowed (read access)
seeFolders allowed (search access)


Owner byte
7
6
5
4
3
2
1
0
makeChanges allowed (write access)
seeFiles allowed (read access)
seeFolders allowed (search access)

`ownerNamePtr` Longword input pointer: Pointer to GS/OS result buffer where the owner name will be returned as a GS/OS string. If the directory is owned by the guest user (usually displayed as `<Any User>`), the owner name is returned as a null string.

`groupNamePtr` Longword input pointer: Pointer to GS/OS result buffer where the group name will be returned as a GS/OS string. If the directory has no group associated with it, the group name is returned as a null string.

Errors	$4B	`badStoreType`	pathname specifies a file instead of a directory

SetPrivileges (AppleShare FSTSpecific subcall)

Description This command sets the access rights, owner name, and group name fields for a specified file or directory.

The string `<Any User>` is not a valid user name (unless you have a registered user by that name).

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 4)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`pathnamePtr`	3	Longword input pointer
$0A	`accessRights`	4	Longword input value
$0E	`ownerNamePtr`	5	Longword input pointer
$12	`groupNamePtr`	6	Longword input pointer

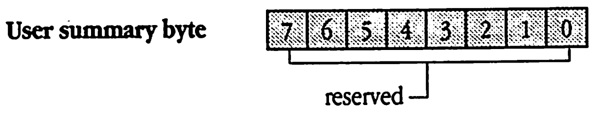
The following parameters have particular values for this subcall.

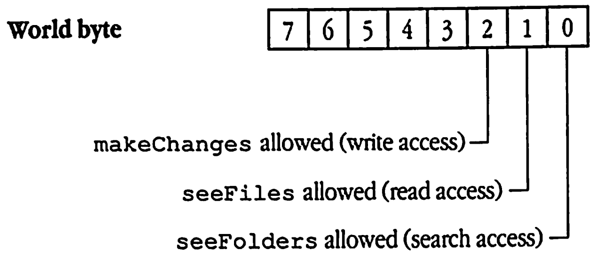
`pCount` Word input value: The number of parameters in this parameter block. Minimum = 4; maximum = 6.

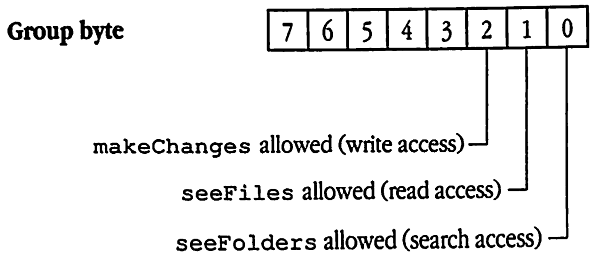
`commandNum` For SetPrivileges, `commandNum` = $0005.

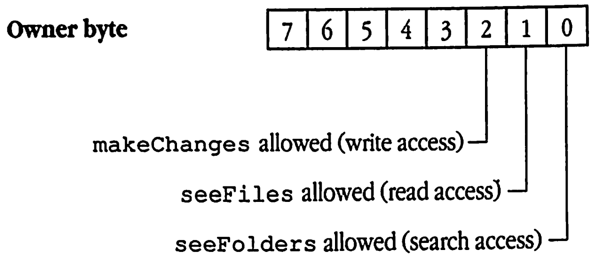
`pathnamePtr` Longword input pointer: Pointer to GS/OS string that contains the pathname of the file or directory whose access privileges are to be set.

`accessRights` Longword input value: The access privileges, as follows:




World byte
7
6
5
4
3
2
1
0
makeChanges allowed (write access)
seeFiles allowed (read access)
seeFolders allowed (search access)


Group byte
7
6
5
4
3
2
1
0
makeChanges allowed (write access)
seeFiles allowed (read access)
seeFolders allowed (search access)


Owner byte
7
6
5
4
3
2
1
0
makeChanges allowed (write access)
seeFiles allowed (read access)
seeFolders allowed (search access)

`ownerNamePtr` Longword input pointer: Pointer to owner name. Setting the owner name to the null string assigns the directory to the guest user (usually known as `<Any User>`). The pointer points to a structure similar to a GS/OS output buffer where the first word (normally the total buffer length) is ignored, the next word is the string length, and the rest of the buffer is the string itself. This structure definition allows you to do a GetPrivileges call, modify the data, and do a SetPrivileges call using the same pointer.

groupNamePtr Longword input pointer: Pointer to group name. Setting the group name to the null string causes no group to be associated with the directory (and therefore the group's access rights are ignored). The pointer points to a structure similar to a GS/OS output buffer where the first word (normally the total buffer length) is ignored, the next word is the string length, and the rest of the buffer is the string itself. This structure definition allows you to do a GetPrivileges call, modify the data, and do a SetPrivileges call using the same pointer.

Errors

$4B	badStoreType	the pathname specifies a file instead of a directory
$4E	invalidAccess	the specified user is not the current owner of the directory
$7E	unknownUser	the specified user name is not the name of a registered user
$7F	unknownGroup	the specified group name is not the name of a registered group

UserInfo (AppleShare FSTSpecific subcall)

Description This command returns the user name and primary group name of a user.

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 4)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	devNum	3	Word input value
$08	userNamePtr	4	Longword input pointer
$0C	groupNamePtr	5	Longword input pointer

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 4; maximum = 5.

`commandNum` For UserInfo, `commandNum` = $0006.

`devNum` Word input value: Device number of a volume on the desired server whose user info is to be returned.

`userNamePtr` Longword input pointer: Pointer to GS/OS result buffer where the user name will be returned as a GS/OS string. If the user is logged on as a guest, the user name is returned as a null string.

`groupNamePtr` Longword input pointer: Pointer to GS/OS result buffer where the primary group name will be returned as a GS/OS string. If the user has no primary group, this name is returned as a null string.

Errors (none except general GS/OS errors)

CopyFile (AppleShare FSTSpecific subcall)

Description This command causes a file on a server to be copied by the server. The copy may be between different volumes as long as both volumes are on the same server.

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 4)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`srcPtr`	3	Longword input pointer
$0A	`destPtr`	4	Longword input pointer

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 4; maximum = 4.

`commandNum` For CopyFile, `commandNum` = $0007.

`srcPtr` Longword input pointer: Pointer to GS/OS string that contains the source pathname.

`destPtr` Longword input pointer: Pointer to GS/OS string that contains the destination pathname.

Errors

$4A	`badFileFormat`	the server does not support this call
$53	`paramRangeErr`	one of the volumes is not a server volume or the volumes are not on the same server

GetUserPath (AppleShare FSTSpecific subcall)

Description

This subcall returns a pointer to a GS/OS string that contains the pathname of the user's directory on the user volume.

The path is constructed on each call. The string's contents will not change until the next call to GetUserPath. The string is suitable for use as a parameter to a SetPrefix call.

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 3)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$08	`prefixPtr`	3	Longword output pointer

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 3; maximum = 3.

`commandNum` For GetUserPath, `commandNum` = $0008.

`prefixPtr` Longword output pointer: GS/OS returns a pointer to a string that contains the pathname of the user's directory on the user volume, using colons as separators and without a trailing colon.

△ **Important** Do not modify the string. This is not a pointer to a result buffer. The data pointed to is kept inside the AppleShare FST, which is why you should not modify it. △

Errors

$60	`dataUnavail`	no user volume is mounted or the user name could not be determined

OpenDesktop (AppleShare FSTSpecific subcall)

Description

This subcall takes a pathname of a volume or a pathname of a file or folder and returns a desktop reference number for the volume. That number must be supplied for all other desktop database calls.

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 4)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`dtRefNum`	3	Word input value
$08	`pathnamePtr`	4	Longword input pointer

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 4; maximum = 4.

`commandNum` For OpenDesktop, `commandNum` = $0009.

`dtRefNum` Word output value: Desktop reference number of the desktop for the volume.

`pathnamePtr` Longword input pointer: Pointer to GS/OS string containing the pathname of the volume whose desktop will be opened, or the pathname of a file or folder on the volume whose desktop will be opened.

Errors

(none except general GS/OS errors)

CloseDesktop (AppleShare FSTSpecific subcall)

Description This command takes a desktop reference number and a volume name or pathname and frees all resources allocated when that reference number was opened.

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 4)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	dtRefNum	3	Word input value
$08	pathnamePtr	4	Longword input pointer

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 4; maximum = 4.

`commandNum` For CloseDesktop, `commandNum` = $000A.

`dtRefNum` Word input value: Desktop reference number of the volume whose desktop will be closed.

`pathnamePtr` Longword input pointer: Pointer to GS/OS string containing the pathname of the volume whose desktop will be closed, or the pathname of a file or folder on the volume whose desktop will be closed.

Errors (none except general GS/OS errors)

GetComment (AppleShare FSTSpecific subcall)

Description This command takes a desktop reference number and a pathname and returns the comment associated with the file, directory, or volume. If no comment has been stored, a null string is returned for the comment.

Offset		No.	Size and type
$00	`pCount`		Word input value (minimum = 5)
$02	`fileSysID`	1	Word input value
$04	`commandNum`	2	Word input value
$06	`dtRefNum`	3	Word input value
$08	`pathnamePtr`	4	Longword input pointer
$0C	`commentPtr`	5	Longword input pointer

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 5; maximum = 5.

`commandNum` For GetComment, `commandNum` = $000B.

`dtRefNum` Word input value: Desktop reference number of the volume.

`pathnamePtr` Longword input pointer: Pointer to GS/OS string containing the pathname of the file, directory, or volume whose comment is retrieved.

`commentPtr` Longword input pointer: Pointer to GS/OS result buffer where the comment will be returned as a GS/OS string; the string in this case will not be longer than 199 characters.

Errors (none except general GS/OS errors)

SetComment (AppleShare FSTSpecific subcall)

Description This subcall sets the comment for a specified file, directory, or volume.

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 4)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	dtRefNum	3	Word input value
$08	pathnamePtr	4	Longword input pointer
$0C	commentPtr	5	Longword input pointer

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 4; maximum = 5.

`commandNum` For SetComment, `commandNum` = $000C.

`dtRefNum` Word input value: Desktop reference number of a file or directory.

`pathnamePtr` Longword input pointer: Pointer to GS/OS string containing the pathname of the file, directory, or volume whose comment will be set.

`commentPtr` Longword input pointer: Pointer to GS/OS string defining the comment. If the string is null, or not supplied (that is, if `pCount` = 4), then the comment is removed. If the comment string is longer than 199 characters, it is truncated to 199 characters without an error.

Errors (none except general GS/OS errors)

GetSrvrName (AppleShare FSTSpecific subcall)

Description This subcall takes a pathname and returns the server name and zone name for that volume. If either the `srvrNamePtr` parameter or `srvrZonePtr` parameter is null ($0000 0000), the string is not returned. If the server name or zone name is unknown, it is returned as a null string.

Offset		No.	Size and type
$00	pCount		Word input value (minimum = 4)
$02	fileSysID	1	Word input value
$04	commandNum	2	Word input value
$06	pathnamePtr	3	Longword input pointer
$0A	srvrNamePtr	4	Longword input pointer
$0E	srvrZonePtr	5	Longword input pointer

The following parameters have particular values for this subcall.

`pCount` Word input value: The number of parameters in this parameter block. Minimum = 4; maximum = 5.

`commandNum` For GetSrvrName, `commandNum` = $000D.

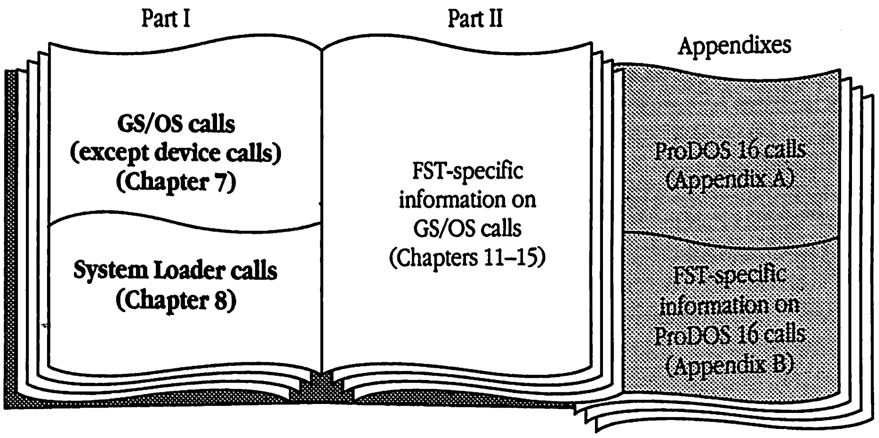
`pathnamePtr` Longword input pointer: Pointer to GS/OS string containing the pathname of the volume whose server name and zone name is to be returned.

`srvrNamePtr` Longword input pointer: Pointer to GS/OS result buffer where the server name will be returned as a GS/OS string.

`srvrZonePtr` Longword input pointer: Pointer to GS/OS result buffer where the server zone will be returned as a GS/OS string.

Errors (none except general GS/OS errors)

Appendixes


Part I
Part II
Appendixes
GS/OS calls
(except device calls)
(Chapter 7)
System Loader calls
(Chapter 8)
FST-specific
information on
GS/OS calls
(Chapters 11–15)
ProDOS 16 calls
(Appendix A)
FST-specific
information on
ProDOS 16 calls
(Appendix B)

Appendix A **GS/OS ProDOS 16 Calls**

This appendix provides a detailed description of all the GS/OS ProDOS 16 calls, arranged in alphabetical order by call name. These calls are provided only for compatibility with ProDOS 16. For the standard GS/OS calls, see Chapter 7, "GS/OS Call Reference," in Part I of this book.

The descriptions in this appendix follow the same conventions as those for the standard GS/OS calls.

$0031 ALLOC_INTERRUPT

Description

This function places the address of an interrupt handler into GS/OS's interrupt vector table.

For a complete description of GS/OS's interrupt-handling subsystem, see Chapter 10. See also the `DEALLOC_INTERRUPT` call in this appendix.

Parameters

Offset		Size and type
$00	`intNum`	Word result value
$02	`intCode`	Longword input pointer

`intNum` Word result value: An identifying number assigned by GS/OS to the binding between the interrupt source and the interrupt handler. Its only use is as input to the `DEALLOC_INTERRUPT` call.

`intCode` Longword input pointer: Points to the first instruction of the interrupt handler routine.

Errors

$25	`irqTableFull`	interrupt vector table full
$53	`paramRangeErr`	parameter out of range

$0004 CHANGE_PATH

Description

This call changes a file's pathname to another pathname on the same volume, or renames a volume.

`CHANGE_PATH` cannot be used to change a device name. You must use the configuration program to change device names.

Parameters

Offset		Size and type
$00	`pathname`	Longword input pointer
$04	`newPathname`	Longword input pointer

`pathname` Longword input pointer: Points to a Pascal string that represents the name of the file whose pathname is to be changed.

`newPathname` Longword input pointer: Points to a Pascal string that represents the new pathname of the file whose name is to be changed.

Comments

A file may not be renamed while it is open.

A file may not be renamed if rename access is disabled for the file.

A subdirectory *s* may not be moved into another subdirectory *t* if *s* = *t* or if *t* is contained in the directory hierarchy starting at *s*. For example, "rename `/v` to `/v/w`" is illegal, as is "rename `/v/w` to `/v/w/x`".

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$47	`dupPathname`	duplicate pathname
$4A	`badFileFormat`	version error
$4B	`badStoreType`	unsupported storage type

$4E	`invalidAccess`	file not destroy-enabled
$50	`fileBusy`	file open
$52	`unknownVol`	unsupported volume type
$53	`paramRangeErr`	parameter out of range
$57	`dupVolume`	duplicate volume
$58	`notBlockDev`	not a block device
$5A	`damagedBitMap`	block number out of range

$000B CLEAR_BACKUP_BIT

Description

This call alters a file's state information to indicate that the file has been backed up and not altered since the backup. Whenever a file is altered, GS/OS sets the file's state information to indicate that the file has been altered.

Parameters

Offset		Size and type
$00	`pathname`	Longword input pointer

`pathname` Longword input pointer: Points to a Pascal string that gives the pathname of the file or directory whose backup status is to be cleared.

Errors

$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2B	`drvrWrtProt`	write-protected disk
$2E	`drvrDiskSwitch`	disk switched
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4A	`badFileFormat`	version error
$52	`unknownVol`	unsupported volume type
$58	`notBlockDev`	not a block device

$0014 CLOSE

Description

This call closes the access path to the specified file, releasing all resources used by the file and terminating further access to it. Any file-related information that has not been written to the disk is written, and memory-resident data structures associated with the file are released.

If the specified value of the `fileRefNum` parameter is $0000, all files at or above the current system file level are closed.

Parameters

Offset		Size and type
$00	`fileRefNum`	Word input value

`fileRefNum` Word input value: The identifying number assigned to the file by the `OPEN` call. A value of $0000 indicates that all files at or above the current system file level are to be closed.

Errors

$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$2E	`drvrDiskSwitch`	disk switched
$43	`invalidRefNum`	invalid reference number
$48	`volumeFull`	volume full
$5A	`damagedBitMap`	block number out of range

$0001 CREATE

Description

This call creates a standard file, an extended file, or a subdirectory on a volume mounted in a block device. A standard file is a ProDOS-like file containing a single sequence of bytes; an extended file is a Macintosh-like file containing a data fork and a resource fork, each of which is an independent sequence of bytes; a subdirectory is a data structure that contains information about other files and subdirectories.

This call cannot be used to create a volume directory; the `FORMAT` call performs that function. Similarly, it cannot be used to create a character-device file.

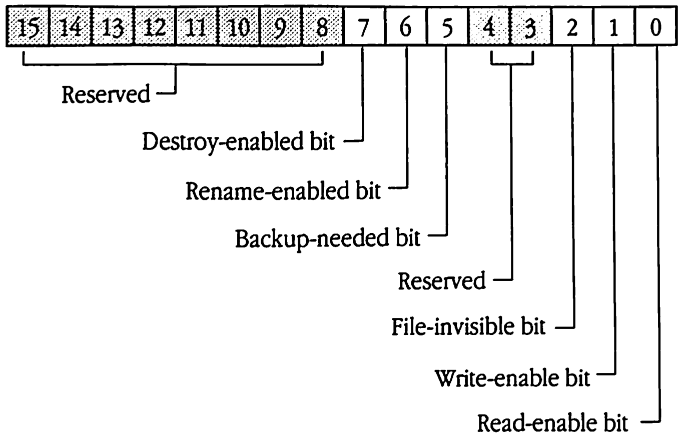
This call sets up file system state information for the new file and initializes the file to the empty state.

Parameters

Offset		Size and type
$00	`pathname`	Longword input pointer
$04	`fAccess`	Word input value
$06	`fileType`	Word input value
$08	`auxType`	Longword input value
$0C	`storageType`	Word input value
$0E	`createDate`	Word input value
$10	`createTime`	Word input value

`pathname` Longword input pointer: Points to a Pascal string representing the pathname of the file to be created. This is the only required parameter.

`fAccess` Word input value: Specifies how the file may be accessed after it is created and whether or not the file has changed since the last backup.



The most common setting for the access word is $00C3.

Software that supports file hiding (invisibility) should use bit 2 of the flag to indicate whether or not to display a file or subdirectory.

`fileType` Word input value: Used conventionally by system and application programs to categorize the file's contents. The value of this field has no effect on GS/OS's handling of the file, except that only certain file types may be executed directly by GS/OS.

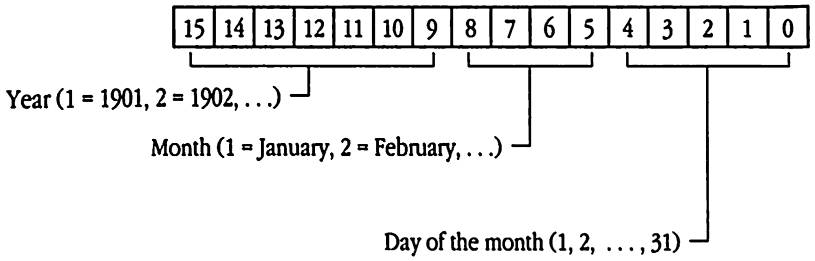
`auxType` Longword input value: Used by system and application programs to store additional information about the file. The value of this field has no effect on GS/OS's handling of the file. By convention, the interpretation of values in this field depends on the value in the `fileType` field.

`storageType` Word input value: The value of this parameter determines whether the file being created is a standard file, extended file, or subdirectory file, as follows:

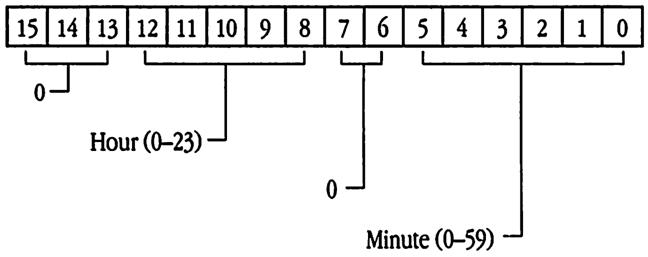
$0000–$0003*	create a standard file
$0005	create an extended file
$8005	convert existing standard file to contain a resource fork
$000D	create a subdirectory file

*If this field contains $0000, $0002, or $0003, GS/OS interprets it as $0001 and actually changes it to $0001 on output. All other values are invalid.

`createDate` Word input value: This parameter specifies a date that GS/OS saves as the file's creation date value. If this word is $0000, GS/OS gets the date from the system clock.


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Year (1 = 1901, 2 = 1902, . . .)
Month (1 = January, 2 = February, . . .)
Day of the month (1, 2, . . . , 31)

`createTime` Word input value: This parameter specifies the time that GS/OS saves as the file's creation time value. If this word is $0000, GS/OS gets the time from the system clock.



Comments

The `CREATE` call applies only to files on block devices.

The storage type of a file cannot be changed after it is created. For example, there is no direct way to add a resource fork to a standard file or to remove one of the forks from an extended file.

All FSTs implement standard files, but they are not required to implement extended files.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found

$47	`dupPathname`	duplicate pathname
$48	`volumeFull`	volume full
$49	`volDirFull`	volume directory full
$4B	`badStoreType`	unsupported (or incorrect) storage type
$52	`unknownVol`	unsupported volume type
$53	`paramRangeErr`	parameter out of range
$58	`notBlockDev`	not a block device
$5A	`damagedBitMap`	block number out of range

$0032 DEALLOC_INTERRUPT

Description

This function removes a specified interrupt handler from the interrupt vector table. See also the ALLOC_INTERRUPT call in this appendix.

Parameters

Offset		Size and type
$00	intNum	Word input value

intNum Word input value: Interrupt identification number of the binding that is to be undone between interrupt source and interrupt handler.

Errors

$53	paramRangeErr	parameter out of range

$0002 DESTROY

Description

This call deletes a specified standard file, extended file (both the data fork and resource fork), or subdirectory and updates the state of the file system to reflect the deletion. After a file is destroyed, no other operations on the file are possible.

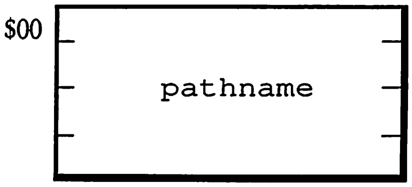
This call cannot be used to delete a volume directory; the `FORMAT` call reinitializes volume directories. Similarly, this call cannot be used to delete a character-device file.

It is not possible to delete only the data fork or only the resource fork of an extended file.

Before deleting a subdirectory file, you must empty it by deleting all the files it contains.

Parameters

Offset **Size and type**



Longword input pointer

`pathname` Longword input pointer: Points to a Pascal string that represents the pathname of the file to be deleted.

Comments

A file cannot be destroyed if it is currently open or if the access attributes do not permit destroy access.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4B	`badStoreType`	unsupported storage type
$4E	`invalidAccess`	file not destroy-enabled
$50	`fileBusy`	file open
$52	`unknownVol`	unsupported volume type
$53	`paramRangeErr`	parameter out of range
$58	`notBlockDev`	not a block device
$5A	`damagedBitMap`	block number out of range

$002C D_INFO

Description This call returns general information about a device attached to the system.

Parameters

Offset		Size and type
$00	devNum	Word input value
$02	devName	Longword input pointer

devNum Word input value: A device number. GS/OS assigns device numbers in sequence (1, 2, 3, and so on) as it loads or creates the device drivers. There is no fixed correspondence between devices and device numbers. To get information about every device in the system, make repeated calls to D_INFO with devNum values of 1, 2, 3, and so on until GS/OS returns error $53 (paramRangeErr).

devName Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string containing the device name of the device specified by device number. The maximum size of the string is 31 bytes, so the maximum size of the returned value is 33 bytes. Thus, the buffer size should be 35 bytes.

Errors

$11	invalidDevNum	invalid device number
$53	paramRangeErr	parameter out of range

$0025 ERASE_DISK

Description This call puts up a dialog box that allows the user to erase a specified volume and choose which file system is to be placed on the newly erased volume. The volume must previously have been physically formatted. The only difference between ERASE_DISK and FORMAT is that ERASE_DISK does not physically format the volume. See the FORMAT call later in this appendix.

Parameters

Offset		Size and type
$00	devName	Longword input pointer
$04	volName	Longword input pointer
$08	fileSysID	Word result value

devName Longword input pointer: Points to a Pascal string that represents the device name of the device containing the volume to be erased.

volName Longword input pointer: Points to a Pascal string that represents the volume name to be assigned to the newly erased volume.

fileSysID Word result value: If the call is successful, this field identifies the file system with which the disk was formatted. If the call is unsuccessful, this field is undefined.

$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E–$000F	Reserved
$0007	Lisa		

Errors

$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device request
$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2B	`drvrWrtProt`	write-protected disk
$53	`paramRangeErr`	parameter out of range
$5D	`osUnsupported`	file system not available
$64	`invalidFSTID`	invalid FST ID

$000E EXPAND_PATH

Description

This call converts the input pathname into the corresponding full pathname with colons (ASCII $3A) as separators. If the input is a full pathname, `EXPAND_PATH` simply converts all of the separators to colons. If the input is a partial pathname, `EXPAND_PATH` concatenates the specified prefix with the rest of the partial pathname and converts the separators to colons.

If bit 15 (MSB) of the `flags` parameter is set, the call converts all lowercase characters to uppercase (all other bits in this parameter must be cleared). This call also performs limited syntax checking. It returns an error if it encounters an illegal character, two adjacent separators, or any other syntax error.

Parameters

Offset		Size and type
$00	`inputPath`	Longword input pointer
$04	`outputPath`	Longword input pointer
$08	`flags`	Word input value

`inputPath` Longword input pointer: Points to a Pascal input string that is to be expanded.

`outputPath` Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string that contains the expanded pathname.

`flags` Word input value: If bit 15 is set to 1, this call returns the expanded pathname in uppercase characters. All other bits in this word must be 0.

Errors

$40	`badPathSyntax`	invalid pathname syntax
$4F	`buffTooSmall`	buffer too small

$0015 FLUSH

Description

This call writes to the volume all file state information that is buffered in memory but has not yet been written to the volume. The purpose of this call is to assure that the representation of the file on the volume is consistent and up to date with the latest GS/OS calls affecting the file. Thus, if a power failure occurs immediately after the FLUSH call completes, it should be possible to read all data written to the file as well as all file attributes. If such a power failure occurs, files that have not been flushed may be in inconsistent states, as may the volume as a whole.

A value of $0000 for the fileRefNum parameter indicates that all files at or above the current system file level are to be flushed.

Parameters

Offset		Size and type
$00	fileRefNum	Word input value

fileRefNum Word input value: The identifying number assigned to the file by the OPEN call. A value of $0000 indicates that all files at or above the current system file level are to be flushed.

Errors

$27	drvrIOError	I/O error
$2B	drvrWrtProt	write-protected disk
$2E	drvrDiskSwitch	disk switched
$43	invalidRefNum	invalid reference number
$48	volumeFull	volume full
$5A	damagedBitMap	block number out of range

$0024 FORMAT

Description

This call puts up a dialog box that allows the user to physically format a specified volume and choose which file system is to be placed on the newly formatted volume.

Some devices do not support physical formatting. In this case the `FORMAT` call writes only the empty file system, and in effect is just like the `ERASE_DISK` call. See the `ERASE_DISK` call earlier in this chapter.

Parameters

Offset		Size and type
$00	`devName`	Longword input pointer
$04	`volName`	Longword input pointer
$08	`fileSysID`	Word result value

`devName` Longword input pointer: Points to a Pascal string that represents the device name of the device containing the volume to be formatted.

`volName` Longword input pointer: Points to a Pascal string that represents the volume name to be assigned to the newly formatted blank volume.

`fileSysID` Word result value: If the call is successful, this field identifies the file system with which the disk was formatted. If the call is unsuccessful, this field is undefined. The file system IDs are as follows:

$0000	reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E–$000F	reserved
$0007	Lisa		

Errors

$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device number request
$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2B	`drvrWrtProt`	write-protected disk
$53	`paramRangeErr`	parameter out of range
$5D	`osUnsupported`	file system not available
$64	`invalidFSTID`	invalid FST ID

$0028 GET_BOOT_VOL

Description

This call returns the volume name of the volume from which the file GS/OS was last loaded and executed. The volume name returned by this call is equivalent to the prefix specified by */.

Parameters

Offset		Size and type
$00	dataBuffer	Longword input pointer

dataBuffer Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string containing the boot volume name.

Errors

$4F	buffTooSmall	buffer too small

$0020 GET_DEV_NUM

Description

This call returns the device number of a device identified by device name or volume name. Only block devices may be identified by volume name, and then only if the named volume is mounted. Most other device calls refer to devices by device number.

GS/OS assigns device numbers at boot time. The numbers are a series of consecutive integers beginning with 1. There is no algorithm for determining the device number for a particular device.

Because a device may hold different volumes and because volumes may be moved from one device to another, the device number returned for a particular volume name may be different at different times.

Parameters

Offset		Size and type
$00	devName	Longword input pointer
$04	devNum	Word result value

`devName` Longword input pointer: Points to a Pascal string that represents the device name or volume name (for a block device).

`devNum` Word result value: The device reference number of the specified device.

Errors

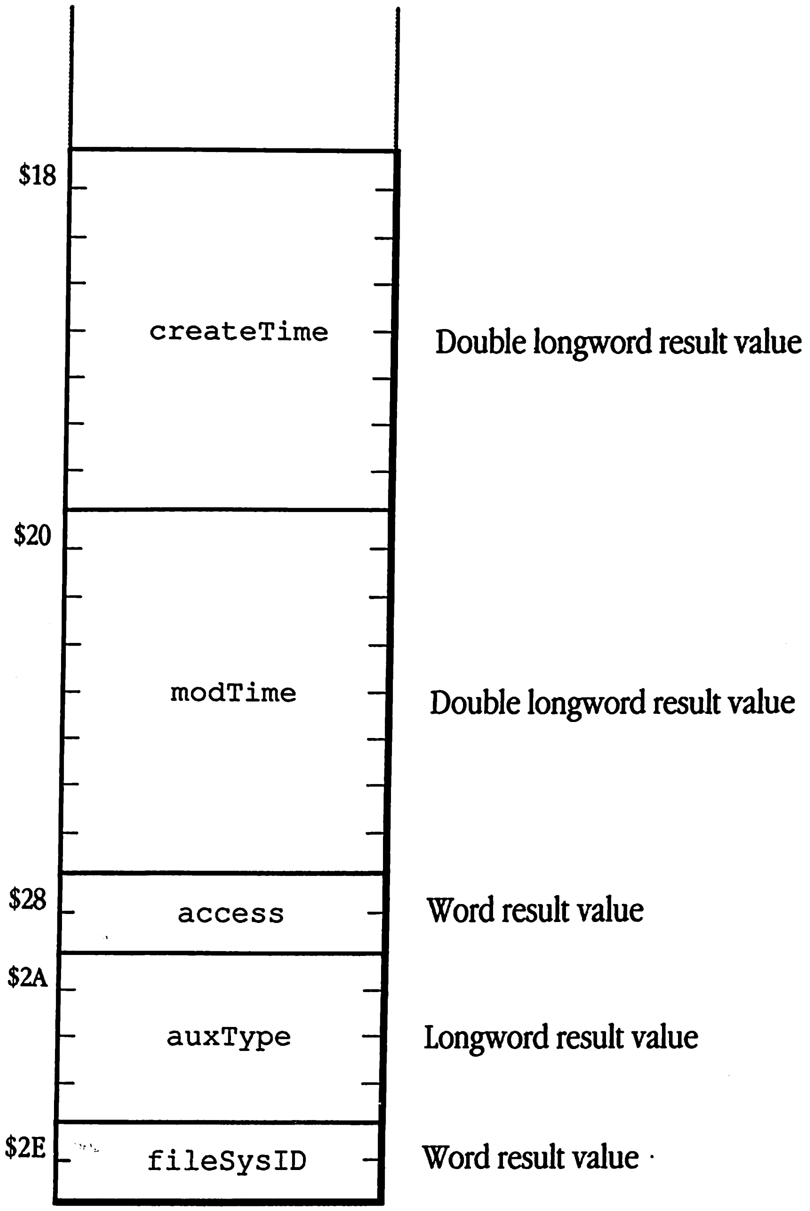
$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device number request
$40	`badPathSyntax`	invalid pathname syntax
$45	`volNotFound`	volume not found

$001C GET_DIR_ENTRY

Description This call returns information about a directory entry in the volume directory or a subdirectory. Before executing this call, the application must open the directory or subdirectory. The call allows the application to step forward or backward through file entries or to specify absolute entries by entry number.

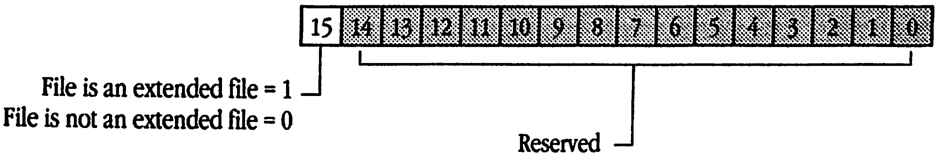
Parameters

Offset		Size and type
$00	`refNum`	Word input value
$02	`flags`	Word result value
$04	`base`	Word input value
$06	`displacement`	Word input value
$08	`nameBuffer`	Longword input pointer
$0C	`entryNum`	Word result value
$0E	`fileType`	Word result value
$10	`endOfFile`	Longword result value
$14	`blockCount`	Longword result value
$18		



`refNum` Word input value: The identifying number assigned to the directory or subdirectory by the `OPEN` call.

`flags` Word result value: Flags that indicate various attributes of the file.


15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
File is an extended file = 1
File is not an extended file = 0
Reserved

`base` Word input value: A value that tells how to interpret the displacement field, as follows:

$0000	displacement gives an absolute entry number
$0001	displacement is added to current displacement to get next entry number
$0002	displacement is subtracted from current displacement to get next entry number

`displacement` Word input value: In combination with the base parameter, the displacement specifies the directory entry whose information is to be returned. When the directory is first opened, GS/OS sets the current displacement value to $0000. The current displacement value is updated on every `GET_DIR_ENTRY` call.

If the base and displacement fields are both 0, GS/OS returns a 2-byte value in the `entryNumber` parameter that specifies the total number of active entries in the subdirectory. In this case, GS/OS also resets the current displacement to the first entry in the subdirectory.

To step through the directory entry by entry, you should set the `base` and `displacement` parameters to $0001.

`nameBuffer` Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string containing the name of the file or subdirectory represented in this directory entry.

`entryNum` Word result value: The absolute entry number of the entry whose information is being returned. This field is provided so that a program can obtain the absolute entry number even if the `base` and `displacement` parameters specify a relative entry.

`fileType` Word result value: The value of the file type field of the directory entry.

`endOfFile` Longword result value: Value of the EOF field of the directory entry.

`blockCount` Longword result value: The value of the blocks used field of the directory entry.

`createTime` Double longword result value: The value of the creation date/time field of the directory entry.

`modTime` Double longword result value: The value of the modification date/time field of the directory entry.

`access` Word result value: Value of the access attribute field of the directory entry.

`auxType` Longword result value: Value of the auxiliary type field of the directory entry.

`fileSysID` Word result value: File system identifier of the file system on the volume containing the file. Values of this field are described under the `VOLUME` call.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$4A	`badFileFormat`	version error
$4B	`badStoreType`	unsupported storage type
$4F	`buffTooSmall`	buffer too small
$52	`unknownVol`	unsupported volume type
$53	`paramRangeErr`	parameter out of range
$58	`notBlockDev`	not a block device
$61	`endOfDir`	end of directory

$0019 GET_EOF

Description

This function returns the current logical size of a specified file. See also the `SET_EOF` call in this appendix.

Parameters

Offset		Size and type
$00	`refNum`	Word input value
$02	`eof`	Longword result value

`refNum` Word input value: The identifying number assigned to the file by the `OPEN` call.

`eof` Longword result value: The current logical size of the file, in bytes.

Errors

$43	`invalidRefNum`	invalid reference number

$0006 GET_FILE_INFO

Description

This call returns certain file attributes of an existing open or closed block file.

△ **Important** A `GET_FILE_INFO` call following a `SET_FILE_INFO` call on an open file may not return the values set by the `SET_FILE_INFO` call. To guarantee recording of the attributes specified in a `SET_FILE_INFO` call, you must first close the file. △

See also the `SET_FILE_INFO` call in this appendix.

Parameters

Offset		Size and type
$00	`pathname`	Longword input pointer
$04	`fAccess`	Word result value
$06	`fileType`	Word result value
$08	`auxType`	Longword result value
$0C	`storageType`	Word result value
$0E	`createDate`	Word result value
$10	`createTime`	Word result value
$12	`modDate`	Word result value
$14	`modTime`	Word result value
$16	`blocksUsed`	Longword result value

`pathname` Longword input pointer: Points to a Pascal string representing the pathname of the file whose file information is to be retrieved.

`fAccess` Word result value: Value of the file's access attribute, which is described under the `CREATE` call.

`fileType` Word result value: Value of the file's file type attribute.

`auxType` Longword result value: Value of the file's auxiliary type attribute.

`storageType` Word result value: Value indicating the storage type of the file, as follows:

$01	standard file
$05	extended file
$0D	volume directory or subdirectory file

`createDate` Word result value: Value for the file's creation date attribute, which is described under the `CREATE` call.

`createTime` Word result value: Value for the file's creation time attribute, which is described under the `CREATE` call.

`modDate` Word result value: Value for the file's modification date attribute. The format is the same as that of the `createDate` parameter.

`modTime` Word result value: Value for the file's modification time attribute. The format is the same as that of the `createTime` parameter.

`blocksUsed` Longword result value: For a standard file, this field gives the total number of blocks used by the file. For an extended file, this field gives the number of blocks used by the file's data fork.

For a subdirectory or volume directory file, this field is undefined.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4A	`badFileFormat`	version error
$4B	`badStoreType`	unsupported storage type
$52	`unknownVol`	unsupported volume type
$53	`paramRangeErr`	parameter out of range
$58	`notBlockDev`	not a block device

$0021 GET_LAST_DEV

Description

This call returns the device number of the last accessed device. The last accessed device is defined as the last device to which any device command was directed by GS/OS as the result of a GS/OS call.

A program that uses this call must take into account that the last device value can change at any time if a device-accessing GS/OS call is made by an asynchronously executed process such as a desk accessory or interrupt handler.

To ensure that the `GET_LAST_DEV` call returns the last device accessed by the given program, the program must

1. disable interrupts
2. make the GS/OS call that accesses the device (for example, `OPEN`, `READ`)
3. make the `GET_LAST_DEV` call
4. restore the interrupt state that was current before step 1

Unfortunately, this sequence locks out interrupts for more than the maximum recommended interrupt disable time. Therefore, system integrity cannot be guaranteed, especially in a networked environment, where rapid interrupt handling is crucial.

△ **Important** Because of this danger to system integrity, use this call with caution, if at all. △

Parameters

Offset		Size and type
$00	`devNum`	Word result value

`devNum` Word result value: Device number of the last accessed device.

Errors

$59 `invalidLevel` invalid file level

$001B GET_LEVEL

Description This function returns the current value of the system file level. See also the SET_LEVEL call in this appendix.

Parameters

Offset		Size and type
$00	level	Word result value

level Word result value: The value of the system file level.

Errors

$59	invalidLevel	invalid file level

$0017 GET_MARK

Description This function returns the current mark for the specified file. See also the SET_MARK call in this appendix.

Parameters

Offset		Size and type
$00	markRefNum	Word input value
$02	position	Longword result value

markRefNum Word input value: The identifying number assigned to the file by the OPEN call.

position Longword result value: The current value of the mark, in bytes, relative to the beginning of the file.

Errors

$43	invalidRefNum	invalid reference number

$0027 GET_NAME

Description

Returns the filename (not the complete pathname) of the currently running application program.

To get the complete pathname of the current application, concatenate prefix `1/` with the filename returned by this call. Do this before making any change in prefix `1/`.

Parameters

Offset		Size and type
$00	`dataBuffer`	Longword input pointer

`dataBuffer` Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string containing the filename.

Errors

$4F	`buffTooSmall`	buffer too small

$000A GET_PREFIX

Description

This function returns the current value of any one of the numbered prefixes. The returned prefix string always starts and ends with a separator. If the requested prefix is null, it is returned as a string with the length field set to 0. This call should not be used to get the boot volume prefix (`*/`). See also the `SET_PREFIX` call in this appendix.

Parameters

Offset		Size and type
$00	`prefixNum`	Word input value
$02	`prefix`	Longword input pointer

`prefixNum` Word input value: Binary value of the prefix number for the prefix to be returned.

`prefix` Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string containing the prefix value.

Errors

$4F	`buffTooSmall`	buffer too small
$53	`paramRangeErr`	parameter out of range

$002A GET_VERSION

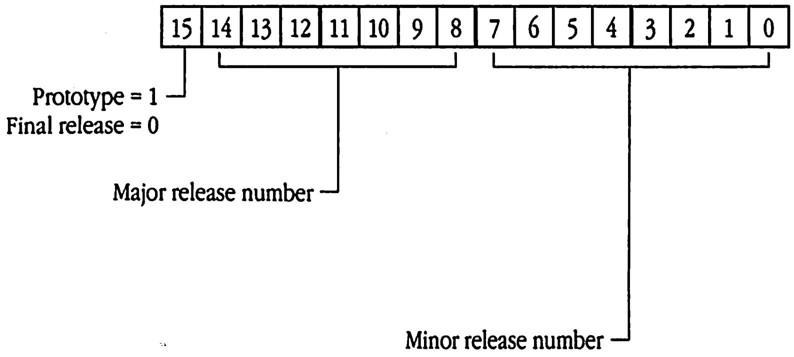
Description

This call returns the version number of the GS/OS operating system. This value can be used by application programs to condition version-dependent operations.

Parameters

Offset		Size and type
$00	`version`	Word result value

`version` Word result value: Version number of the operating system, in the following format:



Errors

(none except general GS/OS errors)

$0011 NEWLINE

Description

This function enables or disables the newline read mode for an open file and, when enabling newline read mode, specifies the newline enable mask and newline character or characters.

When newline mode is disabled, a `READ` call terminates only after it reads the requested number of characters or encounters the end of file. When newline mode is enabled, the read also terminates if it encounters one of the specified newline characters.

When a `READ` call is made while newline mode is enabled and another character is in the file, GS/OS performs the following operations:

1. Transfers the next character to the user's buffer.
2. Performs a logical AND between the character and the low-order byte of the newline mask specified in the last `NEWLINE` call for the open file.
3. Compares the resulting byte with the newline character or characters.
4. If there is a match, terminates the read; otherwise, continues from step 1.

Parameters

Offset		Size and type
$00	`newLRefNum`	Word input value
$02	`enableMask`	Word input value
$04	`newlineChar`	Word input value

`newLRefNum` Word input value: The identifying number assigned to the file access path by the `OPEN` call.

`enableMask` Word input value: If the value of this field is $0000, newline mode is disabled. If the value is greater than $0000, the low-order byte becomes the newline mask. GS/OS performs a logical AND of each input character with the newline mask before comparing it to the newline characters.

`newlineChar` Word input value: The low-order byte of this field is the newline character. When disabling newline mode (`enableMask` = $0000), this parameter is ignored.

Errors

$43 `invalidRefNum` invalid reference number

$0010 OPEN

Description

This call causes GS/OS to establish an access path to a file. Once an access path is established, the user may perform file read and write operations and other related operations on the file.

If you use this call, you allow files to be opened by multiple users. You do not fully prevent one user from changing data that another user is reading, but you do allow multiple users to read a file without changing existing code. The file access is established as follows:

1. An attempt is made to open the file as read/write, deny write.
2. If this fails, an attempt is made to open the file as read-only, deny nothing.
3. If this fails, an attempt is made to open the file as write-only, deny write.
4. If this also fails, error $4E (`invalidAccess`) is returned.

Parameters

Offset		Size and type
$00	`openRefNum`	Word result value
$02	`openPathname`	Longword input pointer
$06	`ioBuffer`	Reserved

`openRefNum` Word result value: A reference number assigned by GS/OS to the access path. All other file calls (`READ`, `WRITE`, `CLOSE`, and so on) refer to the access path by this number.

`openPathname` Longword input pointer: Points to a Pascal string that represents the pathname of the file to be opened.

`ioBuffer` This field is reserved and must be set to $00000000.

Errors

$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2E	`drvrDiskSwitch`	disk switched
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4A	`badFileFormat`	version error
$4B	`badStoreType`	unsupported storage type
$4E	`invalidAccess`	file not destroy-enabled
$4F	`buffTooSmall`	buffer too small
$50	`fileBusy`	file open
$52	`unknownVol`	unsupported volume type
$58	`notBlockDev`	not a block device

$0029 QUIT

Description

This call terminates the running application. It also closes all open files, sets the system file level to 0, initializes certain components of the Apple IIGS and the operating system, and then launches the next application.

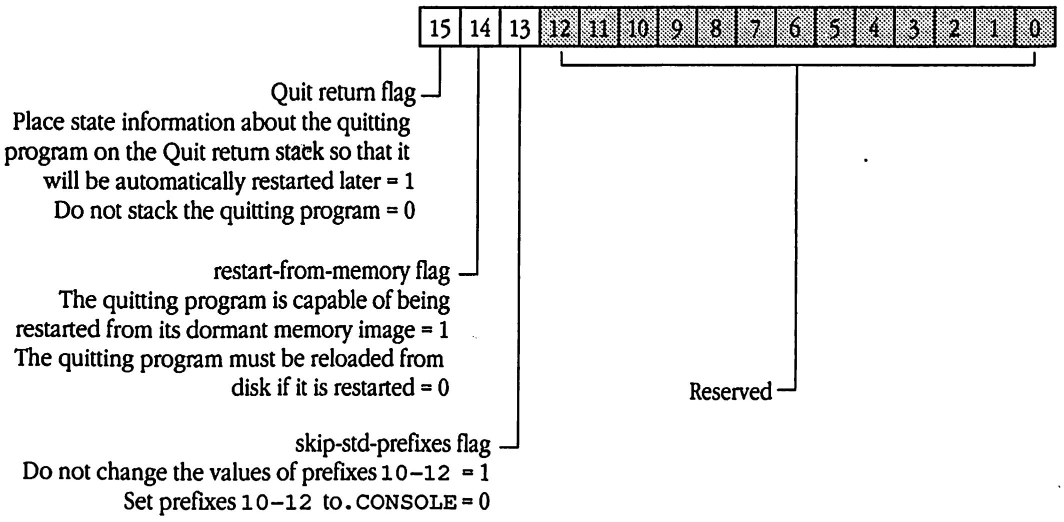
For more information about quitting applications, see Chapter 2, "GS/OS and Its Environment."

Parameters

Offset		Size and type
$02	`quitPathname`	Longword input pointer
$06	`flags`	Word input value

`quitPathname` Longword input pointer: Points to a Pascal string that represents the pathname of the program to run next. If the `quitPathname` parameter is null or the pathname itself has length 0, GS/OS chooses the next application, as described in Chapter 2.

`flags` Word input value: Two Boolean flags that give information about how to handle the program executing the `QUIT` call, as follows:



Quit return flag
Place state information about the quitting program on the Quit return stack so that it will be automatically restarted later = 1
Do not stack the quitting program = 0

restart-from-memory flag
The quitting program is capable of being restarted from its dormant memory image = 1
The quitting program must be reloaded from disk if it is restarted = 0

skip-std-prefixes flag
Do not change the values of prefixes `10-12` = 1
Set prefixes `10-12` to `.CONSOLE` = 0

Reserved

Comments

Only one error condition causes the `QUIT` call to return to the caller: error $07 (GS/OS busy). All other errors are managed within the GS/OS Program Dispatcher.

Errors

(none except general GS/OS errors)

$0012 READ

Description

This function attempts to transfer the number of bytes given by the `requestCount` parameter, starting at the current mark, from the file specified by the `refNum` parameter into the buffer pointed to by the `dataBuffer` parameter. The function updates the file mark to reflect the new file position after the read.

Because of two situations that can cause the `READ` function to transfer fewer than the requested number of bytes, the function returns the actual number of bytes transferred in `transferCount`. If GS/OS reaches the end of file before transferring the number of bytes specified in `requestCount`, it stops reading and sets `transferCount` to the number of bytes actually read.

If newline mode is enabled and a newline character is encountered before the requested number of bytes have been read, GS/OS stops the transfer and sets `transferCount` to the number of bytes actually read, including the newline character.

Parameters

Offset		Size and type
$00	`fileRefNum`	Word input value
$02	`dataBuffer`	Longword input pointer
$06	`requestCount`	Longword input value
$0A	`transferCount`	Longword result value

`fileRefNum` Word input value: The identifying number assigned to the file by the `OPEN` call.

`dataBuffer` Longword input pointer: Points to a memory area large enough to hold the requested data.

`requestCount` Longword input value: The number of bytes to be read.

`transferCount` Longword result value: The number of bytes actually read.

Errors

$27	`drvrIOError`	I/O error
$2E	`drvrDiskSwitch`	disk switched
$43	`invalidRefNum`	invalid reference number
$4C	`eofEncountered`	end-of-file encountered
$4E	`invalidAccess`	access not allowed

$0022 READ_BLOCK

Description

This call reads one 512-byte block of information to a disk specified by a device number.

Normally, you should use D_READ and D_WRITE for all direct device I/O. READ_BLOCK deals only with 512-byte blocks and devices with a maximum of 65,536 blocks, is valid only for the ProDOS FST, and exists only for compatibility with ProDOS 16.

Parameters

Offset		Size and type
$00	blockDevNum	Word input value
$02	blockDataBuffer	Longword input pointer
$06	blockNum	Longword input value

blockDevNum Word input value: The reference number assigned to the device.

blockDataBuffer Longword input pointer: Points to a data buffer large enough to hold the data to be read.

blockNum Longword input value: The number of the block to be read.

Errors

$11	invalidDevNum	invalid device request
$27	drvrIOError	I/O error
$28	drvrNoDevice	no device connected
$2B	drvrWrtProt	write-protected disk
$53	paramRangeErr	parameter out of range

$0018 SET_EOF

Description

This call sets the logical size of an open file to a specified value, which may be either larger or smaller than the current file size. The EOF value cannot be changed unless the file is write-enabled. If the specified EOF is less than the current EOF, the system may—but need not—free blocks that are no longer needed to represent the file. See also the `GET_EOF` call description.

Parameters

Offset		Size and type
$00	`eofRefNum`	Word input value
$02	`eofPosition`	Longword input value

`eofRefNum` Word input value: The identifying number assigned to the file by the `OPEN` call.

`eofPosition` Longword input value: The new logical size of the file, in bytes.

Errors

$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$43	`invalidRefNum`	invalid reference number
$4D	`outOfRange`	position out of range
$4E	`invalidAccess`	file not write-enabled
$5A	`damagedBitMap`	block number out of range

$0005 SET_FILE_INFO

Description

This call sets certain file attributes of an existing open or closed block file. This call immediately modifies the file information in the file's directory entry whether the file is open or closed. It does not affect the file information seen by previously opened access paths to the same file.

> △ **Important** A GET_FILE_INFO call following a SET_FILE_INFO call on an open file may not return the values set by the SET_FILE_INFO call. To guarantee recording of the attributes specified in a SET_FILE_INFO call, you must first close the file. △

See also the GET_FILE_INFO call in this appendix.

Parameters

Offset		Size and type
$00	pathname	Longword input pointer
$04	fAccess	Word input value
$06	fileType	Word input value
$08	auxType	Longword input value
$0C	<null>	Word input value
$0E	createDate	Word input value
$10	createTime	Word input value
$12	modDate	Word input value
$14	modTime	Word input value

`pathname` Longword input pointer: Points to a Pascal string that represents the pathname of the file whose file information is to be set.

`fAccess` Word input value: Value for the file's access attribute, which is described under the `CREATE` call.

`fileType` Word input value: Value for the file's file type attribute.

`auxType` Longword input value: Value of the file's auxiliary type attribute.

`<null>` Word input value: This field is unused and must be set to 0.

`createDate` Word input value: Value for the file's creation date attribute, which is described under the `CREATE` call. If the value of this field is 0, GS/OS does not change the creation date.

`createTime` Word input value: Value for the file's creation time attribute, which is described under the `CREATE` call. If the value of this field is 0, GS/OS does not change the creation time.

`modDate` Word input value: Value for the file's modification date attribute. Format is the same as for the `createDate` parameter. If the value of this field is zero, GS/OS supplies the date from the system clock.

`modTime` Word input value: Value for the file's modification time attribute. Format is the same as for the `createTime` parameter. If the value of this field is zero, GS/OS supplies the time from the system clock.

Errors

$10	`devNotFound`	device not found
$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$40	`badPathSyntax`	invalid pathname syntax
$44	`pathNotFound`	path not found
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$4A	`badFileFormat`	version error
$4B	`badStoreType`	unsupported storage type
$4E	`invalidAccess`	file not destroy-enabled
$52	`unknownVol`	unsupported volume type
$53	`paramRangeErr`	parameter out of range
$58	`notBlockDev`	not a block device

$001A SET_LEVEL

Description

This function sets the current value of the system file level.

Whenever a file is opened, GS/OS assigns it a file level equal to the current system file level. A `CLOSE` call with a `refNum` parameter of $0000 closes all files with file level values at or above the current system file level. Similarly, a `FLUSH` call with a `refNum` parameter of $0000 flushes all files with file level values at or above the current system file level. See also the `GET_LEVEL` call in this appendix.

Parameters

Offset		Size and type
$00	`level`	Word input value

`level` Word input value: The new value of the system file level. Must be in the range $0000–$00FF.

Errors

$59	`invalidLevel`	invalid file level

$0016 SET_MARK

Description

This call sets the mark (the position from which the next byte will be read or to which the next byte will be written) to a specified value. The value can never exceed EOF, the current size of the file. See also the GET_MARK call in this appendix.

Parameters

Offset		Size and type
$00	markRefNum	Word input value
$02	position	Longword input value

markRefNum Word input value: The identifying number assigned to the file by the OPEN call.

position Longword input value: The value assigned to the mark. It is the position (in bytes) relative to the beginning of the file at which the next read or write will begin.

Errors

$27	drvrIOError	I/O error
$43	invalidRefNum	invalid reference number
$4D	outOfRange	position out of range
$5A	damagedBitMap	block number out of range

$0009 SET_PREFIX

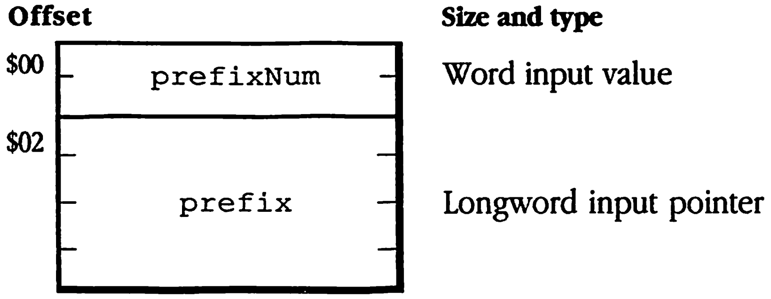
Description

This call sets one of the numbered pathname prefixes to a specified value. The input to this call can be any of the following pathnames:

- a full pathname
- a partial pathname beginning with a numeric prefix designator
- a partial pathname beginning with the special prefix designator `*/`
- a partial pathname without an initial prefix designator

The `SET_PREFIX` call is unusual in the way it treats partial pathnames without initial prefix designators. Normally, GS/OS uses the prefix `0/` in the absence of an explicit designator. However, only in the `SET_PREFIX` call, it uses the prefix *n*/ where *n* is the value of the `prefixNum` field described below. See also the `GET_PREFIX` call in this appendix.

Parameters



`prefixNum` Word input value: A prefix number that specifies the prefix to be set.

`prefix` Longword input pointer: Points to a Pascal string representing the pathname to which the prefix is to be set. If this field is not given, the prefix is set to the NULL string.

Comments

Specifying a pathname with length 0 or whose syntax is illegal sets the designated prefix to NULL.

GS/OS does not verify that the designated prefix corresponds to an existing subdirectory or file.

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

Errors

$40	`badPathSyntax`	invalid pathname syntax
$53	`paramRangeErr`	parameter out of range

$0008 VOLUME

Description Given the name of a block device, this call returns the name of the volume mounted in the device along with other information about the volume.

Parameters

Offset		Size and type
$00	`devName`	Longword input pointer
$04	`volName`	Longword input pointer
$08	`totalBlocks`	Longword result value
$0C	`freeBlocks`	Longword result value
$10	`fileSysID`	Word result value

`devName` Longword input pointer: Points to a Pascal string that contains the name of a block device.

`volName` Longword input pointer: Points to a buffer in which GS/OS places a Pascal string containing the volume name of the volume mounted in the device.

`totalBlocks` Longword result value: Total number of blocks contained in the volume.

`freeBlocks` Longword result value: The number of free (unallocated) blocks in the volume.

`fileSysID` Word result value: Identifies the file system contained in the volume, as follows:

$0000	Reserved	$0008	Apple CP/M
$0001	ProDOS/SOS	$0009	Reserved
$0002	DOS 3.3	$000A	MS/DOS
$0003	DOS 3.2 or 3.1	$000B	High Sierra
$0004	Apple II Pascal	$000C	ISO 9660
$0005	Macintosh (MFS)	$000D	AppleShare
$0006	Macintosh (HFS)	$000E–$000F	Reserved
$0007	Lisa		

Errors

$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device request
$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2E	`drvrDiskSwitch`	disk switched
$45	`volNotFound`	volume not found
$4A	`badFileFormat`	version error
$52	`unknownVol`	unsupported volume type
$53	`paramRangeErr`	parameter out of range
$57	`dupVolume`	duplicate volume
$58	`notBlockDev`	not a block device

$0013 WRITE

Description

This call attempts to transfer the number of bytes specified by the `requestCount` parameter from the application's buffer to the file specified by the `fileRefNum` parameter, starting at the current mark.

The call returns the number of bytes actually transferred. It also updates the mark to indicate the new file position and extends the EOF, if necessary, to accommodate the new data.

Parameters

Offset		Size and type
$00	`refNum`	Word input value
$02	`dataBuffer`	Longword input pointer
$06	`requestCount`	Longword input value
$0A	`transferCount`	Longword result value

`refNum` Word input value: The identifying number assigned to the file by the `OPEN` call.

`dataBuffer` Longword input pointer: Points to the area of memory containing the data to be written to the file.

`requestCount` Longword input value: The number of bytes to write.

`transferCount` Longword result value: The number of bytes actually written.

Errors

$27	`drvrIOError`	I/O error
$2B	`drvrWrtProt`	write-protected disk
$2E	`drvrDiskSwitch`	disk switched
$43	`invalidRefNum`	invalid reference number
$48	`volumeFull`	volume full
$4E	`invalidAccess`	file not destroy-enabled
$5A	`damagedBitMap`	block number out of range

$0023 WRITE_BLOCK

Description

This call writes one 512-byte block of information to a disk specified by a device number.

Normally, you should use `D_READ` and `D_WRITE` for all direct device I/O. `WRITE_BLOCK` deals only with 512-byte blocks and devices with a maximum of 65,536 blocks, is valid only for the ProDOS FST, and exists only for compatibility with ProDOS 16.

Parameters

Offset		Size and type
$00	blockDevNum	Word input value
$02	blockDataBuffer	Longword input pointer
$06	blockNum	Longword input value

`blockDevNum` Word input value: The reference number assigned to the device.

`blockDataBuffer` Longword input pointer: Points to a data buffer that holds the data to be written.

`blockNum` Longword input value: The block number of the destination disk block.

Errors

$11	`invalidDevNum`	invalid device request
$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$2B	`drvrWrtProt`	write-protected disk
$53	`paramRangeErr`	parameter out of range

Appendix B ProDOS 16 Calls and FSTs

This appendix discusses how individual GS/OS file system translators handle ProDOS 16 (GS/OS class 0) calls. It shows only those aspects of each FST's call handling that are different from the descriptions in Appendix A, "GS/OS ProDOS 16 Calls." See that appendix for the standard way to make ProDOS 16 calls to GS/OS.

The ProDOS FST

The ProDOS FST translates ProDOS 16 calls to the format used by the ProDOS file system. Actually, because that is already the file system that ProDOS 16 calls are designed to access, no translation is necessary. All GS/OS ProDOS 16 calls that pass through the ProDOS FST function exactly as described in Appendix A.

See Chapter 12 for more information on the ProDOS FST. For further information on ProDOS 16, see the *Apple IIGS ProDOS 16 Reference.*

The High Sierra FST

The main difference between the High Sierra FST and other FSTs is that High Sierra does not support writing to a file. CD-ROM is a read-only medium.

Table B-1 lists the ProDOS 16 calls, both meaningful and not meaningful, that the High Sierra FST supports. A description of each call's differences from its standard meaning (described in Appendix A) follows.

See Chapter 13 of this book for more information on the High Sierra FST.

- **Table B-1** High Sierra FST ProDOS 16 calls

Meaningful		Not meaningful	
$0006	GET_FILE_INFO	$0001	CREATE
$0008	VOLUME	$0002	DESTROY
$0010	OPEN	$0004	CHANGE_PATH
$0012	READ	$0005	SET_FILE_INFO
$0014	CLOSE	$0013	WRITE
$0016	SET_MARK	$0015	FLUSH
$0017	GET_MARK	$0018	SET_EOF
$0019	GET_EOF	$000B	CLEAR_BACKUP_BIT
001C	GET_DIR_ENTRY	$0025	ERASE_DISK
0020	GET_DEV_NUM	$0024	FORMAT

With the exception of the FLUSH call, all calls on the right side of Table B-1 do nothing and return error $2B (write-protected). The FLUSH call also does nothing, but it returns no error (the carry flag is cleared).

The following sections describe how the High Sierra FST handles some of the meaningful calls differently from standard ProDOS 16 practice. Meaningful calls not described in the following sections are handled exactly as documented in Appendix A.

GET_FILE_INFO ($0006)

The GET_FILE_INFO call returns certain attributes of an existing block file. The file may be open or closed.

Parameters

fileType This word output value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file type mapping table. See the FSTSpecific call description in Chapter 13, "The High Sierra FST."

modDate This word output value always has the same value as createDate.

modTime This word output value always has the same value as createTime.

blocksUsed This longword output value is always the same as the totalBlocks parameter returned from a VOLUME call.

VOLUME ($0008)

Given the name of a block device, the VOLUME call returns the name of the volume mounted in that device and other information about the volume.

Parameters

freeBlocks This longword output value is aways $0000.

GET_DIR_ENTRY ($001C)

The `GET_DIR_ENTRY` call returns information about a directory entry in the volume directory or a subdirectory. Before executing this call, the application must open the directory or subdirectory. The call allows the application to step forward or backward through file entries or to specify absolute entries by entry number.

The High Sierra FST does not allow `READ` calls and `GET_DIR_ENTRY` calls to the same reference number. If an open file has previously been accessed by `GET_DIR_ENTRY`, and a `READ` call is made with the same reference number, the High Sierra FST returns error $4E (invalid access). To avoid the error, open the directory twice.

Parameters

`fileType` This word output value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file type mapping table. See the FSTSpecific call description in Chapter 13, "The High Sierra FST."

`modDateTime` This double longword output value always has the same value as `createDateTime`.

`auxType` This longword output value is always $0000.

`fileSysID` This word output value is always $000B for High Sierra or $000C for ISO 9660.

The Character FST

The Character file system translator (Character FST) provides a file-system-like interface to character devices such as the console, printers, and modems.

Because the Character FST handles ProDOS 16 calls, all ProDOS 16 applications automatically have the ability to access character devices as files when running under GS/OS. ProDOS 16 itself does not provide that capability to ProDOS 16 applications. Table B-2 shows the GS/OS ProDOS 16 calls that the Character FST supports.

Attempting to send any other GS/OS ProDOS 16 call to a character device results in error $58 (`notBlockDev`).

See Chapter 14 for a general descripticn of the Character FST.

■ **Table B-2** GS/OS ProDOS 16 calls supported by the Character FST

Call number	Call name	Call number	Call name
$0010	OPEN	$0014	CLOSE
$0012	READ	$0015	FLUSH
$0013	WRITE	$0011	NEWLINE

OPEN ($0010)

OPEN establishes an access path to the character file.

Parameters pathname This longword input pointer must point to a character device name.

Errors In addition to the standard ProDOS 16 OPEN errors, the Character FST can return these errors from an OPEN call:

$26	drvrNoResrc	resources not available
$2F	drvrOffLine	device off line or no media present

READ ($0012)

The READ call attempts to transfer the requested number of bytes from the specified character file into the application's data buffer.

Errors In addition to the standard ProDOS 16 READ errors, the Character FST can return these errors from a READ call:

$23	drvrNotOpen	character device not open
$2F	drvrOffLine	device off line or no media present

WRITE ($0013)

The WRITE call attempts to transfer the requested number of bytes from the application's data buffer to the specified character file.

Errors In addition to the standard ProDOS 16 WRITE errors, the Character FST can return these errors from a WRITE call:

$23	drvrNotOpen	character device not open
$2F	drvrOffLine	device off line or no media present

CLOSE ($0014)

The CLOSE call terminates access to the specified character file. CLOSE also involves flushing the file (see the FLUSH call) to ensure that all data has been transferred before a character file is closed.

Errors In addition to the standard ProDOS 16 CLOSE errors, the Character FST can return these errors from a CLOSE call:

$23	drvrNotOpen	character device not open
$2F	drvrOffLine	device off line or no media present

FLUSH ($0015)

The FLUSH routine completes any pending data transfer to the character file specified by refNum. If the character device is synchronous, all data transfer is by definition completed when the WRITE call returns, so the FLUSH routine simply returns with no error. If the device is asynchronous (for example, if it is interrupt-driven or has direct memory access), the FLUSH routine waits until all data has been transferred and then returns. If the file is multiply opened, all output access paths to the character file (not just the one with the specified refNum) are flushed.

Errors In addition to the standard ProDOS 16 FLUSH errors, the Character FST can return these errors from a FLUSH call:

$23	drvrNotOpen	character device not open
$2F	drvrOffLine	device off line or no media present

ProDOS 16 device calls

The only ProDOS 16 device call is D_INFO, which is handled only by the Device Manager—no FST can accept this call. Therefore, the standard description of D_INFO in Appendix A is the complete specification.

See the *GS/OS Device Driver Reference* for more general information on the Device Manager and GS/OS device calls.

Appendix C **Apple Extensions to ISO 9660**

This appendix describes a protocol through which file type information can be added to CD-ROM files or other files in the ISO 9660 format (which does not recognize file typing). With this protocol, ProDOS and Macintosh files can be stored on compact discs—as valid ISO 9660 files—while retaining all information related to file type.

You may need to read this appendix if you are

- an Apple Developer working with ISO 9660
- a publisher of authoring tools for ISO 9660 discs
- a publisher of ISO 9660 discs
- a publisher of ISO 9660 receiving system software

◆ *Note:* ISO 9660 is the international file system standard for CD-ROM; it is based on the original High Sierra format, but is not identical to it. The protocol described in this appendix applies to the ISO 9660 file system; however, the High Sierra FST (see Chapter 13) supports the protocol for High Sierra–formatted files also.

What the Apple extensions do

Creating an ISO 9660 CD-ROM disc containing ProDOS files or Macintosh hierarchical file system (HFS) files can have great advantages. The large storage capacity of compact discs means cost savings and greater convenience when distributing large amounts of data, and the position of ISO 9660 as an international standard means that the files will be accessible on a large variety of machines. Unfortunately, both the HFS and ProDOS file systems require information that the ISO 9660 file system does not support. ProDOS requires a file type and an auxiliary file type, and HFS requires a file type, a file creator, and file attributes.

This appendix defines a protocol that extends the ISO 9660 specification. The protocol is designed both to solve existing compatibility problems and to allow for future expansion. At present, it has two principal features:

- It permits inclusion of HFS-specific or ProDOS-specific information in files, without corrupting the ISO 9660 structures. Discs created using the protocol are valid ISO 9660 discs and should function normally on non-Apple receiving systems.
- It defines a mechanism for preserving filenames across translations between ProDOS and ISO 9660, and gives suggestions for optimum translations of Macintosh filenames.

The protocol uses the `SystemIdentifier` field in the Primary Volume Descriptor for global information, and the `SystemUse` field in the directory record for file-specific information.

The protocol identifier

Discs that have been formatted with the Apple extensions to ISO 9660 are identified by their *protocol identifier*, which has the following characteristics:

Location The `SystemIdentifier` field in the Primary Volume Descriptor.

Size 32 bytes. It is the entire contents of the `SystemIdentifier` field.

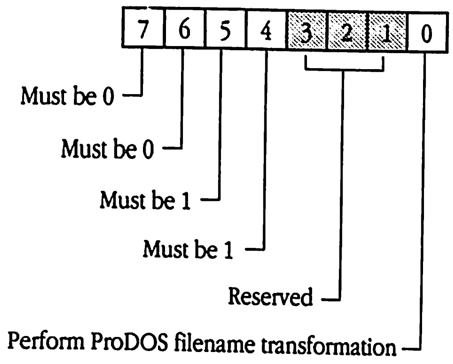
Contents `"APPLE COMPUTER, INC., TYPE: "` followed by the *protocol flags*. In hexadecimal, the protocol identifier looks like this:

41 50 50 4C 45 20 43 4F 4D 50 55 54 45 52 2C 20
49 4E 43 2E 2C 20 54 59 50 45 3A 20 3x 3x 3x 3x

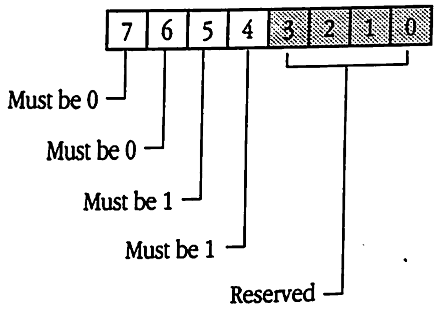
The protocol identifier is considered valid if its first 28 bytes match the first 28 characters above.

Protocol flags Four bytes of nibble-encoded information (represented as *3x* in the previous example). Nibble encoding is necessary in order to guarantee that the bytes represent legal ISO 9660 *a-characters* (printable characters). The flag bytes are numbered 0–3; flag byte 0 is the byte following the space ($20). The bits of each flag byte are numbered 0–7, 0 being the least significant. The flag bytes are presently defined as follows:

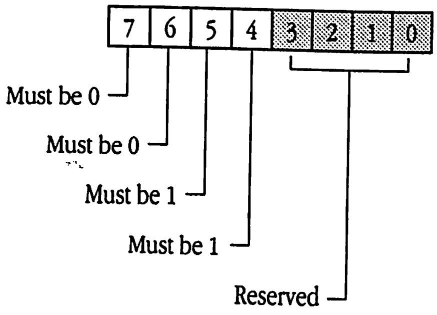
flag byte 0:



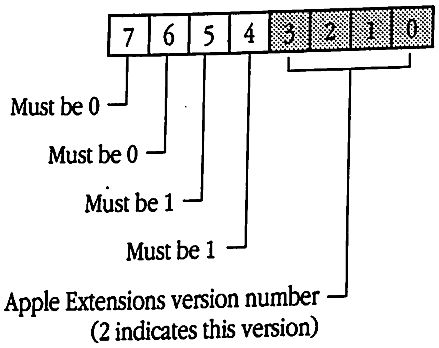
flag byte 1:



flag byte 2:



flag byte 3:



The directory record SystemUse field

Directory records in the ISO 9660 specification have the following format:

```
byte        DirectoryRcdLength
byte        XARlength
struct      ExtentLocation
struct      DataLength
struct      RecordingDateTime
byte        FileFlags
byte        FileUnitSize
byte        InterleaveGapSize
long        VolumeSequenceNum
byte        FileNameLength
char        FileName[FileNameLength]
byte        RecordPad
char        SystemUse[SystemUseLength]
```

The `RecordPad` field is present only if needed to make `DirectoryRcdLength` an even number. If `RecordPad` is present, its value must be 0 ($00).

The `SystemUse` field is an optional field. If it is present, it must begin with a signature word, followed by a length byte, followed by a 1-byte `SystemUseID` field, followed by file-specific information. This structure may be repeated multiple times for multiple file systems, and each structure may be an even or odd number; however, the total length of the `SystemUse` area (`SystemUseLength`) must be an even number. The signature word allows a receiving system to ensure that it can interpret the following data correctly, and `SystemUseID` determines the type and format of the information that follows.

△ **Important** CD-ROM XA discs (that is, discs that follow the Phillips, Microsoft, and Sony standard for interleaved audio) must be handled as a special case. The signature word, if present, begins at byte 14 of the `SystemUse` field rather than at byte 0. △

There are two Apple signature words (`AppleSignature`). The preferred `AppleSignature` word is defined as `AA` ($41 41). The old `AppleSignature` word, used for a previous version, was defined as `BA` ($42 41). Discs pressed using the old format are still supported, but new discs should be pressed using only the current format.

Receiving systems must perform a simple calculation to determine if the `SystemUse` field is present in any given directory record. It is present if

```
DirectoryRcdLength - FileNameLength > 34
```

Receiving systems should first verify that the `SystemUse` field is present, then check for `AppleSignature` before interpreting `SystemUseID`.

SystemUseID

`SystemUseID` can have the values shown in Table C-1 and C-2, depending upon the value of the `AppleSignature` word.

- **Table C-1** Defined values for `SystemUseID` for `AA` signature

Value	Meaning
$00	(Reserved)
$01	ProDOS file type and auxiliary type follow
$02	HFS file type, file creator, and Finder flags follow
$03–FF	(Reserved)

- **Table C-2** Defined values for `SystemUseID` for `BA` signature

Value	Meaning
$00	(Reserved)
$01	ProDOS file type and auxiliary type follow
$02	HFS file type and file creator follow
$03	HFS file type and file creator follow (bundle bit set)
$04	HFS file type, file creator, and 'ICN#' resource (128-byte icon) follow
$05	HFS file type, file creator, and 'ICN#' resource follow (bundle bit set)
$06	HFS file type, file creator, and Finder flags follow
$07–FF	(Reserved)

Tables C-3 and C-4 define the contents of the `SystemUse` field for each defined value of `SystemUseID` for each signature.

■ **Table C-3** Contents of `SystemUse` field for each value of `SystemUseID` for `AA` signature

`SystemUseID` = 01 (ProDOS)	
Offset	**Contents**
$00–01	$41 41 (`AppleSignature`)
$02	`SystemUse` Extension Length ($07 for this ID, including signature bytes)
$03	01 (`SystemUseID`)
$04	ProDOS file type
$05–$06	ProDOS auxiliary type (LSB–MSB)*

`SystemUseID` = 02 (HF, Finder Flags†)	
Offset	**Contents**
$00–01	$41 41 (`AppleSignature`)
$02	`SystemUse` Extension Length ($0E for this ID, including signature bytes)
$03	$02 (`SystemUseID`)
$04–07	HFS file type (MSB–LSB)*
$08–0B	HFS file creator (MSB–LSB)*
$0C–0D	HFS Finder flags (MSB–LSB)*

*(MSB–LSB) = the most significant byte occupies the lowest address; the least significant byte, the highest address; (LSB–MSB) = the least significant byte occupies the lowest address; the most significant byte, the highest address.

†To fill the Finder flags field here, premastering software can simply copy the Finder flags as retrieved by the HFS call PBGetFInfo. Only bits 5 (always switch-launch), 12 (system file), 13 (bundle bit), and 15 (locked) are used. All other bits are either ignored or always set by the FST. See Macintosh Technical Note #40 for more details about the Finder flags.

■ **Table C-4** Contents of `SystemUse` field for each value of `SystemUseID` for `BA` signature

`SystemUseID` = 01 (ProDOS)

Offset	Contents
$00–01	$42 41 (`AppleSignature`)
$02	$01 (`SystemUseID`)
$03	ProDOS file type
$04–05	ProDOS auxiliary type (LSB–MSB)[*]

`SystemUseID` = 02 (HFS)

Offset	Contents
$00–01	$42 41 (`AppleSignature`)
$02	$02 (`SystemUseID`)
$03–06	HFS file type (MSB–LSB)
$07–0A	HFS file creator (MSB–LSB)[*]
$0B	(Padding for even length)

`SystemUseID` = 03 (HFS, bundle bit set)

Offset	Contents
$00–01	$42 41 (`AppleSignature`)
$02	$03 (`SystemUseID`)
$03–06	HFS file type (MSB–LSB)[*]
$07–0A	HFS file creator (MSB–LSB)[*]
$0B	(Padding for even length)

`SystemUseID` = 04 (HFS, icon)

Offset	Contents
$00–01	$42 41 (`AppleSignature`)
$02	$04 (`SystemUseID`)
$03–06	HFS file type (MSB–LSB)[*]
$07–0A	HFS file creator (MSB–LSB)[*]
$0B–8A	HFS 'ICN#' resource (MSB–LSB)[*]
$8B	(Padding for even length)

[*] (MSB–LSB) = the most significant byte occupies the lowest address; the least significant byte, the highest address;
(LSB–MSB) = the least significant byte occupies the lowest address; the most significant byte, the highest address.

(continued)

■ **Table C-4** Contents of `SystemUse` field for each value of `SystemUseID` for `BA` signature (Continued)

`SystemUseID` = 05 (HFS, icon, bundle bit set)	
Offset	**Contents**
$00–01	$42 41 (`AppleSignature`)
$02	$05 (`SystemUseID`)
$03–06	HFS file type (MSB—LSB)[*]
$07–0A	HFS file creator (MSB-LSB)[*]
$0B–8A	HFS 'ICN#' resource (MSB–LSB)[*]
$8B	(Padding for even length)

`SystemUseID` = 06 (HFS, Finder flags)[†]	
Offset	**Contents**
$00–01	$42 41 (`AppleSignature`)
$02	$05 (`SystemUseID`)
$03–06	HFS file type (MSB–LSB)[*]
$07–0A	HFS file creator (MSB-LSB)[*]
$0B–0C	HFS Finder flags (MSB–LSB)[*]

[*] (MSB–LSB) = the most significant byte occupies the lowest address; the least significant byte, the highest address; (LSB–MSB) = the least significant byte occupies the lowest address; the most significant byte, the highest address.

[†] To fill the Finder flags field here, premastering software can simply copy the finder flags as retrieved by the HFS call GetFInfo. Only bits 5 (always switch-launch), 12 (system file), 13 (bundle bit), and 15 (locked) are used. All other bits are either ignored or always set by the FST. See Macintosh Technical Note #40 for more details about the Finder flags.

The Extension to ISO 9660

This section describes, more or less in the style of *ISO 9660,* first edition, 1988-04-15, the extension to ISO 9660 that allows multiple users of the `SystemUse` field. All references are to that document. This section is redundant with the other material in this appendix, but is offered to you in case you are used to reading the ISO specification.

Section 9.1.13, System Use [PB (LEN_DR–LEN_SU + 1) to LEN_DR], shall be replaced as follows:

This field shall be optional. If present, this field shall be reserved for system use. If necessary, so that the Directory Record comprises an even number of bytes, a ($00) byte shall be added to terminate this field.

If this field is present, it must be broken up into a series of System Use Extensions. There can be more than one System Use Extension for a given directory record, subject only to the limitation that the total length of a directory record must be able to be recorded in the 8-bit field defined in section 9.1.1. A System Use Extension must have the following format:

BP 1 Byte 1 of the signature. This field contains an 8-bit number, and must be recorded according to 7.1.1.

BP 2 Byte 2 of the signature. This field contains an 8-bit number, and must be recorded according to 7.1.1.

BP 3 (LEN_SE) This field contains an 8-bit number that specifies the length in bytes of this System Use Extension, including the length of the two signature bytes preceding this byte. This field shall be recorded according to 7.1.1.

BP 4 to LEN_SE This field shall be reserved for system use. Its content is not specified by this international standard.

△ **Important** For CD-ROM XA discs, the byte positions listed above must be adjusted upward by 14 to account for XA's 14-byte fixed-length `SystemUse` field. △

Filename transformations

The rules governing permissible filenames are different under ISO 9660 than under either ProDOS or Macintosh HFS. Therefore, one problem with putting ProDOS or HFS files on an ISO 9660 disc is how to rename them. Ideally, there should be a simple, reversible transformation that can be applied to a filename to make it a legal ISO 9660 name, and reversed to restore the original ProDOS or HFS name.

Such a transformation exists for ProDOS and is given here. There is none for HFS, but guidelines to minimize changes during transformation are listed.

ProDOS

Legal ProDOS filenames differ from legal filenames under ISO 9660 in these ways:

- ProDOS filenames allow multiple periods; ISO 9660 filenames do not.
- ISO 9660 requires that both the period (`.`) and the semicolon (`;`) occur as separators in each filename, and that the semicolon be followed by a version number. (This requirement is for nondirectory files only.)

The following steps constitute a reversible transformation that preserves ProDOS filename syntax. An authoring tool can apply the transformation to any ProDOS file to get a legal ISO 9660 filename, and a receiving system can reverse the transformation to hide from an application the fact that a transformation has occurred. A user can therefore access the file using its original ProDOS filename.

When creating an ISO 9660 disc from ProDOS source files, the authoring tool must perform the following transformation on *all* filenames:

1. Replace all periods in the ProDOS filename with underscores. If the file is a directory file, that completes the transformation.
2. If the file is not a directory file, append the characters `.;1` to the filename. It is now a valid ISO 9660 filename.

◆ *Note:* The ProDOS volume name becomes the ISO 9660 Volume Identifier in the Primary Volume Descriptor. It is a filename and, therefore, must be transformed like other ProDOS filenames. It must be transformed as a directory name (periods replaced with underscores).

After all filenames have been transformed, the authoring tool must set the ProDOS transformation bit in the protocol identifier, described earlier in this appendix.

Table C-5 shows some examples of the transformation.

■ **Table C-5** ProDOS-to-ISO 9660 filename transformations

ProDOS filename	Kind of file	ISO 9660 filename
`PRODOS`	standard	`PRODOS.;1`
`BASIC.SYSTEM`	standard	`BASIC_SYSTEM.;1`
`SYSTEM`	directory	`SYSTEM`
`DESK.ACCS`	directory	`DESK_ACCS`
`START.GS.OS`	standard	`START_GS_OS.;1`

The receiving system can inspect the ProDOS transformation bit in the protocol identifier and make the conversions necessary so that the original ProDOS filenames can be used to refer to all files and directories on the volume. The receiving system performs this transformation on user-supplied filenames before searching for them on disc, and reverses the transformation before presenting filenames to the user.

Remember that this transformation cannot be done on a file-by-file basis; it must be applied to every file and directory on a disc.

Macintosh HFS

Because HFS file-naming rules are very flexible, most HFS filenames are illegal in the ISO 9660 specification. Furthermore, no reversible transformation is possible without degrading performance; unlike with ProDOS, there is no simple conversion from all valid Macintosh HFS filenames to valid ISO 9660 filenames. To make the transformations as consistent as possible, however, Apple Computer, Inc., recommends that authoring tools and receiving systems follow these guidelines when performing HFS-to-ISO 9660 transformations:

1. Convert all lowercase characters to uppercase.
2. Replace all illegal characters, including periods, with underscores.
3. If the filename needs to be shortened, truncate the rightmost characters.
4. If the file is not a directory file, append the characters `.;1` to the filename.

Such a transformation is not reversible, but its results will at least be consistent across all files and discs.

ISO 9660 associated files

An associated file under ISO 9660 is analogous to the resource fork of an HFS file. The format of associated files is defined in the ISO 9660 specification; the Apple extensions do not change the format in any way. For clarity, however, this section restates the definition and gives an example.

An associated file has these characteristics:

- It is one of two identically named files in a directory; the associated file has exactly the same file identifier as its counterpart.
- It resides immediately before its counterpart in the directory.
- It has the associated bit set in the file flags byte of the directory record.

The associated file is equivalent to the resource fork of an HFS file; its counterpart is equivalent to the data fork of the same HFS file.

For example, if the file `ANYFILE.;1` has an associated file, two adjacent directory records will be named `ANYFILE.;1`. The first one (the resource fork) will have the associated bit set; the second one (the data fork) will have the associated bit clear.

Appendix D Delta Guide to GS/OS System Software Version 5.0 Changes

GS/OS system software version 5.0 includes many enhancements and performance improvements. The tables in this chapter summarize the changes between version 4.0 and version 5.0. More information about the enhancements can be found in appropriate chapters of this manual.

New features for the application programmer

Table D-1 summarizes the new features in GS/OS system software version 5.0 that you might want to use for your application program.

■ **Table D-1** New features in GS/OS version 5.0

New feature	Description	Reference
New loader	The new ExpressLoad loader loads applications and load files more quickly than the System Loader. To use ExpressLoad, you must store your application in ExpressLoad format.	Chapter 8, "Loading Program Files"
AppleShare support	This allows your application to be launched from a server. Be careful about the access privileges you assign when you create a file.	Chapter 4, "Accessing GS/OS Files"
New AppleShare FST	This FST supports AppleShare and includes several FSTSpecific calls.	Chapter 15, "The AppleShare FST"
New @ prefix	GS/OS sets this prefix to a pathname specified by the AppleShare FST when an application being launched resides on an AppleShare volume.	Chapter 4, "Accessing GS/OS Files"
New general notification queue	This queue allows your application to be notified when certain GS/OS events occur. The new system calls AddNotifyProc and DelNotifyProc have been added to support the new queue.	Chapter 6, "Working With System Information," and the descriptions of AddNotifyProc and DelNotifyProc in Chapter 7
Additional control of Volume Mount and error dialog boxes	Two new preference bits have been defined for the SetSysPrefs and GetSysPrefs system calls. These bits control the appearance of the Volume Mount dialog and error dialog boxes.	Descriptions of SetSysPrefs and GetSysPrefs in Chapter 7
New system information calls	Three new calls—GetStdRefNum, GetRefNum, and GetRefInfo—get information about the reference number, access attributes, and full pathname of an open file.	Chapter 6, "Working With System Information," and the descriptions of GetStdRefNum, GetRefNum, and GetRefInfo in Chapter 7

■ **Table D-1** New features in GS/OS version 5.0 (continued)

New feature	Description	Reference
New cache feature	A new feature added to the Cache Manager allows better use of a small cache when sessions are enabled. The Cache Manager places itself in the out-of-memory queue. If the Memory Manager cannot allocate the requested memory, the Cache Manager purges the GS/OS Cache.	(None)
New and faster device calls	The Device Manager has added the call DRename to allow your application to rename a device. In addition, the DControl, DInfo, DStatus, DRead, and DWrite calls are now faster.	The description of DRename in Chapter 7
New Console Driver mechanism	Two new calls, AddTrap and ResetTrap, allow you to install and remove your own console driver trap vector.	Chapter 9, "Using the Console Driver"
Redirection support	Prefixes `10`, `11`, and `12` are supported as standard input, standard output, and standard error, respectively.	Chapter 2, "GS/OS and Its Environment"

Enhanced features for the application programmer

Table D-2 summarizes the enhanced features in GS/OS system software version 5.0, including performance enhancements that might make you reconsider using a feature in your application.

- **Table D-2** Enhancements in GS/OS version 5.0

Enhancement	Description	Reference
Enhanced Console Driver performance	This provides faster screen scrolling and faster write operations (for example, about 11 times faster for single character writes). Also, the Console Driver is now restartable and does not need to be loaded from disk when GS/OS is switched back in after a ProDOS 8 application is run.	Chapter 9, "Using the Console Driver"
Quicker switching from ProDOS 8 to GS/OS	When a ProDOS 8 application is launched from a GS/OS native environment (such as the Finder), GS/OS is stored in memory and thus can be quickly restarted when the ProDOS 8 application exits.	(None)
Enhanced partial pathname handling	If you supply a partial pathname that doesn't contain a prefix designator to GS/OS, and prefix designator `0` is null, GS/OS automatically creates the full pathname by adding `8/` in front of the partial pathname.	Chapter 1, "The GS/OS Abstract File System"
Enhanced interrupt and signal management	A signal dispatching queue offers a method for handling signals sent to GS/OS by the hardware, the firmware, or the application.	Chapter 10, "Handling Interrupts and Signals"
Faster device calls	The DControl, DInfo, DStatus, DRead, and DWrite calls are now faster.	(None)

■ **Table D-2** Enhancements in GS/OS version 5.0 (continued)

Enhancement	Description	Reference
ProDOS FST enhancements	The ProDOS FST has been enhanced as follows: ■ It writes all dirty blocks to disk when the last file that has been modified is closed. Thus, files opened for read access do not affect when the bitmap and directory blocks are updated. ■ It allows you to add a resource fork to an existing standard file. ■ It uses the Cache Manager more often, which means better performance. ■ It uses the `optionList` parameter. ■ It supports notification of volume change events.	Chapter 12, "The ProDOS FST"
High Sierra FST enhancements	This FST has added support for version 2.2 of the Apple Extensions to ISO 9660.	Chapter 13, "The High Sierra FST," and Appendix C, "Apple Extensions to ISO 9660"

New and enhanced features for the device driver writer

Table D-3 summarizes the new and enhanced features in GS/OS system software version 5.0 for the device driver writer. The details of these features are provided in the *GS/OS Device Driver Reference,* but the features are summarized here for your convenience.

■ **Table D-3** New and enhanced device features in GS/OS version 5.0

New or enhanced feature	Description
New SCSI Manager	The SCSI Manager allows future SCSI drivers to be quickly defined and added and also allows future SCSI peripherals to be easily developed.
New SCSI drivers	New HD and CD-ROM drivers take full advantage of the GS/OS caching mechanism and also support both single- and multiple-block I/O, which decreases the hardware overhead.
Enhanced device access	Device access is significantly faster, and the following enhancements have been added: ■ support for notification of disk-switched events through the OS Notification Manager ■ support for keeping GS/OS in memory while running ProDOS 8 applications ■ alert boxes for devices that require a driver not presently installed on the system disk
Enhanced AppleDisk 3.5 and UniDisk 3.5 Drivers	Each of these drivers can now be restarted and does not need to be loaded from disk when GS/OS is switched back in after a ProDOS 8 application is run.

Appendix E GS/OS Error Codes and Constants

This appendix lists and describes the errors that an application can receive as a result of making a GS/OS call.

Column 1 in Table E-1 lists the GS/OS error codes that an application can receive. Column 2 lists the predefined constants whose values are equal to the error codes; the constants are defined in the GS/OS interface files supplied with development systems. Column 3 gives a brief description of what each error means.

■ **Table E-1** GS/OS errors

Code	Constant	Description
$01	`badSystemCall`	bad GS/OS call number
$04	`invalidPcount`	parameter count out of range
$07	`gsosActive`	GS/OS is busy
$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device number (request)
$20	`drvrBadReq`	invalid request
$21	`drvrBadCode`	invalid control or status code
$22	`drvrBadParm`	bad call parameter
$23	`drvrNotOpen`	character device not open
$24	`drvrPriorOpen`	character device already open
$25	`irqTableFull`	interrupt table full
$26	`drvrNoResrc`	resources not available
$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$29	`drvrBusy`	driver is busy
$2B	`drvrWrtProt`	device is write-protected
$2C	`drvrBadCount`	invalid byte count
$2D	`drvrBadBlock`	invalid block address
$2E	`drvrDiskSwitch`	disk has been switched
$2F	`drvrOffLine`	device off line or no media present
$40	`badPathSyntax`	invalid pathname syntax
$43	`invalidRefNum`	invalid reference number
$44	`pathNotFound`	subdirectory does not exist
$45	`volNotFound`	volume not found
$46	`fileNotFound`	file not found
$47	`dupPathname`	create or rename with existing name
$48	`volumeFull`	volume full

■ **Table E-1** GS/OS errors (Continued)

Code	Constant	Description
$49	volDirFull	volume directory full
$4A	badFileFormat	version error (incompatible file format)
$4B	badStoreType	unsupported (or incorrect) storage type
$4C	eofEncountered	end-of-file encountered
$4D	outOfRange	position out of range
$4E	invalidAccess	access not allowed
$4F	buffTooSmall	buffer too small
$50	fileBusy	file is already open
$51	dirError	directory error
$52	unknownVol	unknown volume type
$53	paramRangeErr	parameter out of range
$54	outOfMem	out of memory
$57	dupVolume	duplicate volume name
$58	notBlockDev	not a block device
$59	invalidLevel	specified level outside legal range
$5A	damagedBitMap	block number too large
$5B	badPathNames	invalid path names for ChangePath
$5C	notSystemFile	not an executable file
$5D	osUnsupported	Operating system not supported
$5F	stackOverflow	too many applications on stack
$60	dataUnavail	data unavailable
$61	endOfDir	end of directory has been reached
$62	invalidClass	invalid FST call class
$63	resForkNotFound	file does not contain required resource
$64	invalidFSTID	FST ID is invalid
$67	devNameErr	device exists with same name as replacement name
$70	resExistsErr	cannot expand file, resource fork already exists
$71	resAddErr	cannot add resource fork to this type of file

Appendix F **Object Module Format**

Object module format (OMF) is the general file format followed by all object files, library files, and executable load files that run on the Apple IIGS computer under ProDOS 16 or GS/OS. It is a general format that allows dynamic loading and unloading of file segments, both at startup and while a program is running.

Most application writers need not be concerned with the details of OMF. If, however, you are writing a compiler or other program that must create or modify executable files, or if you want to understand the details of how the System Loader functions, you need to understand OMF.

△ **Important** This appendix describes Version 2.1 of the Apple IIGS object module format (OMF). △

What files are OMF files?

The Apple IIGS object module format (OMF) supports language, linker, library, and loader requirements, and it is extremely flexible, easy to generate, and fast to load.

Under ProDOS 8 on the Apple IIe and Apple IIc, there is only one loadable file format, called the binary file format. This format consists of one absolute memory image along with its destination address. ProDOS 8 does not have a relocating loader, so that even if you write relocatable code, you must specify the memory location at which the file is to be loaded.

The Apple IIGS uses a more general format that allows dynamic loading and unloading of file segments while a program is running and that supports the various needs of many languages and assemblers. Apple IIGS linkers (supplied with development environments) and the System Loader fully support relocatable code; in general, you do not specify a load address for an Apple IIGS program, but let the loader and Memory Manager determine where to load the program.

Four kinds of files use object module format: object files, library files, load files, and run-time library files.

- **Object files** are the output from an assembler or compiler and the input to a linker. Object files must be fast to process, easy to create, independent of the source language, and able to support libraries in a convenient way. In some development environments object files also support segmentation of code. They support both absolute and relocatable program segments.

  Apple IIGS object files contain both machine-language code and relocation information for use by the linker. Object files cannot be loaded directly into memory; they must first be processed by the linker to create load files.
- **Library files** contain general object segments that a linker can find and extract to resolve references unresolved in the object files. Only the code needed during the link process is extracted from the library file.
- **Load files,** which are the output of a linker, contain memory images that a loader loads into memory. Load files must be very fast to process. Apple IIGS load files contain load segments that can be relocatable, movable, dynamically loadable, or have any combination of these attributes. Shell applications are load files that can be run from a shell program without requiring the shell to shut down. Startup load files are load files that GS/OS loads during its startup.

  Load files are created by the linker from object files and library files. Load files can be loaded into memory by the System Loader; they cannot be used as input to the linker.

- **Run-time library files** are load files containing general routines that can be shared between applications. The routines are contained in file segments that can be loaded as needed by the System Loader and then purged from memory when they are no longer needed. The run-time library files are also input to the linker, which scans them for unresolved references. However, segments that satisfy references are not included in the link.

All four types of files consist of individual components called segments. Each file type uses a subset of the full object module format. Each compiler or assembler uses a subset of the format depending on the requirements and complexity of the language.

Some common GS/OS file types related to program files are listed in Table F-1.

The rest of this appendix defines object module format. First, the general format specification for all OMF files is described. Then, the unique characteristics of each of the following file types are discussed:

- object files
- library files
- load files
- run-time library files
- shell applications

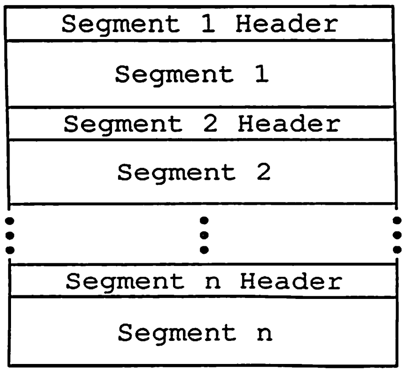
- **Table F-1** GS/OS program-file types

Hexadecimal	Decimal	Mnemonic	Meaning
$B0	176	SRC	Source
$B1	177	OBJ	Object
$B2	178	LIB	Library
$B3	179	S16	GS/OS or ProDOS 16 application
$B4	180	RTL	Run-time library
$B5	181	EXE	Shell application
$B6	182	PIF	Permanent initialization
$B7	183	TIF	Temporary initialization
$B8	184	NDA	New desk accessory
$B9	185	CDA	Classic desk accessory
$BA	186	TOL	Tool set file
$BB	187	DVR	Apple IIgs Device Driver File
$BC	188	LDF	Generic load file (application-specific)
$BD	189	FST	GS/OS file system translator

General format for OMF files

Each OMF file contains one or more segments. Each segment consists of a segment header and a segment body. The segment header contains general information about the segment, such as its name and length. The segment body is a sequence of records; each record consists of either program code or information used by a linker or by the System Loader. Figure F-1 represents the structure of an OMF file.

- **Figure F-1** The structure of an OMF file



Each segment in an OMF file contains a set of records that provide relocation information or contain code or data. If the file is an object file, each segment includes the information the linker needs to generate a relocatable load segment; the linker processes each record and generates a load file containing load segments. If the file is a load file, each segment consists of a memory image followed by a relocation dictionary; the System Loader loads the memory image and then processes the information in the relocation dictionary. (Load file segments on the Apple IIGS are usually relocatable.) Relocation dictionaries are discussed in the section "Load Files," later in this appendix.

Segments in object files can be combined by the linker into one or more segments in the load file. (See the discussion of the LOADNAME field in the section "Segment Header," later in this appendix.) For instance, each subroutine in a program can be compiled independently into a separate (object) code segment; then the linker can be told to place all the code segments into one load segment.

Segment types and attributes

Each OMF segment has a segment type and can have several attributes. The following segment types are defined by OMF:

- code segment
- data segment
- jump-table segment
- pathname segment
- library dictionary segment
- initialization segment
- direct-page/stack segment

The following segment attributes are defined by the object module format:

- reloadable or not reloadable
- absolute-bank or not restricted to a particular bank
- loadable in special memory or not loadable in special memory
- position-independent or position-dependent
- private or public
- static or dynamic
- bank-relative or not bank-relative
- skipped or not skipped

Code and data segments are object segments provided to support languages (such as assembly language) that distinguish program code from data. If a programmer specifies a segment by using a PROC or START assembler directive, the linker flags it as a code segment; if the programmer uses a RECORD or DATA directive instead, the linker flags it as a data segment.

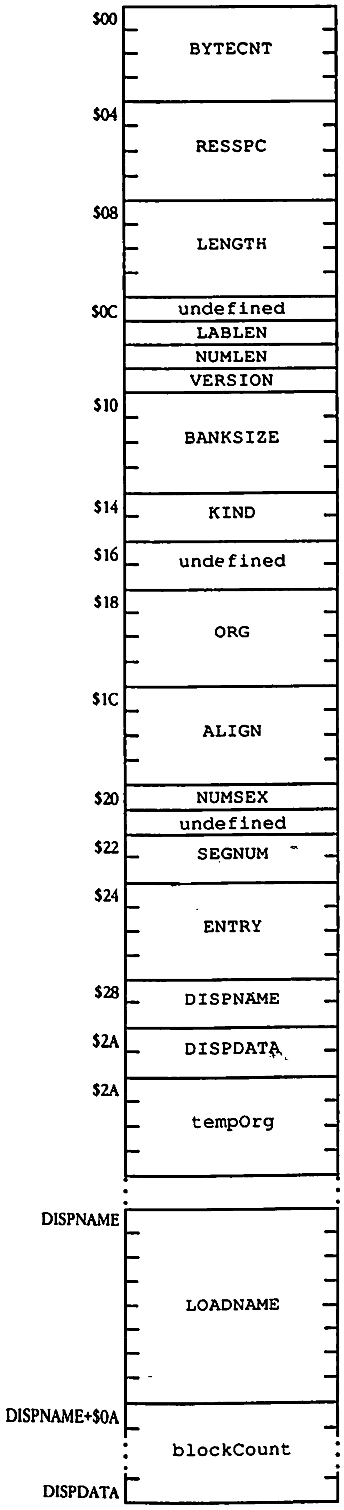
- **Jump-table segments** and pathname segments are load segments that facilitate the dynamic loading of segments; they are described in the section "Load Files" later in this appendix.
- Library dictionary segments allow the linker to scan library files quickly for needed segments; they are described in the section "Library Files" later in this appendix.
- Initialization segments are optional parts of load files that are used to perform any initialization required by the application during an initial load. If used, they are loaded and executed immediately as the System Loader encounters them and are re-executed any time the program is restarted from memory. Initialization segments are described in the section "Load Files" later in this appendix.

- Direct-page/stack segments are load segments used to preset the location and contents of the direct page and stack for an application. See the section "Direct-Page/Stack Segments" later in this appendix for more information.
- Reload segments are load segments that the loader must reload even if the program is restartable and is restarted from memory. They usually contain data that must be restored to its initial values before a program can be restarted.
- Absolute-bank segments are load segments that are restricted to a specified bank but that can be relocated within that bank. The ORG field in the segment header specifies the bank to which the segment is restricted.
- Loadable in special memory means that a segment can be loaded in banks $00, $01, $E0, and $E1. Because these are the banks used by programs running under ProDOS 8 in standard–Apple II emulation mode, you may prevent your program from being loaded in these banks so that it can remain in memory while programs are run under ProDOS 8.
- Position-independent segments can be moved by the Memory Manager during program execution if they have been unlocked by the program.
- A private code segment is a code segment whose name is available only to other code segments within the same object file. (The labels within a code segment are local to that segment.)
- A private data segment is a data segment whose labels are available only to code segments in the same object file.
- Static segments are load segments that are loaded at program execution time and are not unloaded during execution; dynamic segments are loaded and unloaded during program execution as needed.
- Bank-relative segments must be loaded at a specified address within any bank. The ORG field in the segment header specifies the bank-relative address (the address must be less than $10000).
- Skip segments will not be linked by the linker or loaded by the System Loader. However, all references to global definitions in a Skip object segment will be processed by a linker as if the object segment.
- A segment can have only one segment *type* but can have any combination of *attributes*. The segment types and attributes are specified in the segment header by the KIND segment-header field, described in the next section.

Segment header

Each segment in an OMF file has a header that contains general information about the segment, such as its name and length. Segment headers make it easy for the linker to scan an object file for the desired segments, and they allow the System Loader to load individual load segments. The format of the segment header is illustrated in Figure F-2. A detailed description of each of the fields in the segment header follows.

- **Figure F-2** The format of a segment header



△ **Important** In future versions of the OMF, additional fields may be added to the segment header between the DISPDATA and LOADNAME fields. To ensure that future expansion of the segment header does not affect your program, always use DISPNAME and DISPDATA instead of absolute offsets when referencing LOADNAME, SEGNAME, and the start of the segment body, and always be sure that all undefined fields are set to 0. △

BYTECNT A 4-byte field indicating the number of bytes in the file that the segment requires. This number includes the segment header, so you can calculate the starting Mark of the next segment from the starting Mark of this segment plus BYTECNT. Segments need not be aligned to block boundaries.

RESSPC A 4-byte field specifying the number of bytes of 0's to add to the end of the segment. This field can be used in an object segment instead of a large block of zeros at the end of the segment. This field duplicates the effect of a DS record at the end of the segment.

LENGTH A 4-byte field specifying the memory size that the segment will require when loaded. It includes the extra memory specified by RESSPC.

LENGTH is followed by one undefined byte, reserved for future changes to the segment header specification.

LABLEN A 1-byte field indicating the length, in bytes, of each name or label record in the segment body. If LABLEN is 0, the length of each name or label is specified in the first byte of the record (that is, the first byte of the record specifies how many bytes follow). LABLEN also specifies the length of the SEGNAME field of the segment header, or, if LABLEN is 0, the first byte of SEGNAME specifies how many bytes follow. (The LOADNAME field always has a length of 10 bytes.) Fixed-length labels are always left justified and padded with spaces.

NUMLEN A 1-byte field indicating the length, in bytes, of each number field in the segment body. This field is 4 for the Apple IIGS.

VERSION A 1-byte field indicating the version number of the object module format with which the segment is compatible. At the time of publication, this field is set to 2 for the current object module format.

REVISION

A 1-byte field indicating the revision number of the object module format with which the segment is compatible. Together with the VERSION field, REVISION specifies the OMF compatibility level of this segment. At the time of publication, this field is set to 1 for the current object module format.

BANKSIZE

A 4-byte binary number indicating the maximum memory-bank size for the segment. If the segment is in an object file, the linker ensures that the segment is not larger than this value. (The linker returns an error if the segment is too large.) If the segment is in a load file, the loader ensures that the segment is loaded into a memory block that does not cross this boundary. For Apple IIGS code segments, this field must be $00010000, indicating a 64K bank size. A value of 0 in this field indicates that the segment can cross bank boundaries. Apple IIGS data segments can use any number from $00 to $00010000 for BANKSIZE.

KIND

A 2-byte field specifying the type and attributes of the segment. The bits are defined as shown in Table F-2, on the next page. The column labeled "Where described" indicates the section in this appendix where the particular segment type or attribute is discussed.

A segment can have only one *type* but any combination of *attributes*. For example, a position-independent dynamic data segment has KIND = ($A001).

△ **Important** If segment KINDs are specified in the source file, and the KINDs of the object segments placed in a given load segment are not all the same, the segment KIND of the first object segment determines the segment KIND of the entire load segment. △

KIND is followed by two undefined bytes, reserved for future changes to the segment header specification.

ORG

A 4-byte field indicating the absolute address at which this segment is to be loaded in memory, or, for an absolute-bank segment, the bank number. A value of 0 indicates that this segment is relocatable and can be loaded anywhere in memory. A value of 0 is normal for the Apple IIGS.

■ **Table F-2** KIND field definition

Bit(s)	Values	Meaning	Where described
0–4		*Segment Type subfield*	
	$00	Code	Segment Types and Attributes
	$01	Data	Segment Types and Attributes
	$02	Jump-table segment	Load Files
	$04	Pathname segment	Segment Types and Attributes
	$08	Library dictionary segment	Library Files
	$10	Initialization segment	Load Files
	$12	Direct-page/stack segment	Direct-Page/Stack Segments
8–15		*Segment Attributes bits*	
8	if = 1	Bank-relative segment	Segment Types and Attributes
9	if = 1	Skip segment	Segment Types and Attributes
10	if = 1	Reload segment	Segment Types and Attributes
11	if = 1	Absolute-bank segment	Segment Types and Attributes
12	if = 0	Can be loaded in special memory	Segment Types and Attributes
13	if = 1	Position independent	Segment Types and Attributes
14	if = 1	Private	Segment Types and Attributes
15	if = 0	Static; otherwise dynamic	Segment Types and Attributes

ALIGN

A 4-byte binary number indicating the boundary on which this segment must be aligned. For example, if the segment is to be aligned on a page boundary, this field is $00000100; if the segment is to be aligned on a bank boundary, this field is $00010000. A value of 0 indicates that no alignment is needed. For the Apple IIGS, this field must be a power of 2, less than or equal to $00010000. Currently, the loader supports only values of 0, $00000100, and $00010000; for any other value, the loader uses the next higher supported value.

NUMSEX

A 1-byte field indicating the order of the bytes in a number field. If this field is 0, the least significant byte is first. If this field is 1, the most significant byte is first. This field is set to 0 for the Apple IIGS.

NUMSEX is followed by one undefined byte, reserved for future changes to the segment header specification.

SEGNUM

A 2-byte field specifying the segment number. The segment number corresponds to the relative position of the segment in the file (starting with 1). This field is used by the System Loader to search for a specific segment in a load file.

ENTRY

A 4-byte field indicating the offset into the segment that corresponds to the entry point of the segment.

DISPNAME

A 2-byte field indicating the displacement of the LOADNAME field within the segment header. Currently, DISPNAME = 44. DISPNAME is provided to allow for future additions to the segment header; any new fields will be added between DISPDATA and LOADNAME. DISPNAME allows you to reference LOADNAME and SEGNAME no matter what the actual size of the header.

DISPDATA

A 2-byte field indicating the displacement from the start of the segment header to the start of the segment body. DISPDATA is provided to allow for future additions to the segment header; any new fields will be added between DISPDATA and LOADNAME. DISPDATA allows you to reference the start of the segment body no matter what the actual size of the header.

tempORG

A 4-byte field indicating the temporary origin of the Object segment. A nonzero value indicates that all references to globals within this segment will be interpreted as if the Object segment started at that location. However, the actual load address of the Object segment is still determined by the ORG field.

LOADNAME

A 10-byte field specifying the name of the load segment that will contain the code generated by the linker for this segment. More than one segment in an object file can be merged by the linker into a single segment in the load file. This field is unused in a load segment. The position of LOADNAME may change in future revisions of the OMF; therefore, you should always use DISPNAME to reference LOADNAME.

SEGNAME

A field that is LABLEN bytes long, and that specifies the name of the segment. The position of SEGNAME may change in future revisions of the OMF; therefore, you should always use DISPNAME to reference SEGNAME.

Segment body

The body of each segment is composed of sequential records, each of which starts with a 1-byte operation code. Each record contains either program code or information for the linker or System Loader. All names and labels included in these records are LABLEN bytes long, and all numbers and addresses are NUMLEN bytes long (unless otherwise specified in the following definitions). For the Apple IIGS, the least significant byte of each number field is first, as specified by NUMSEX.

Several of the OMF records contain expressions that have to be evaluated by the linker. The operation and syntax of expressions are described in the next section, "Expressions." If the description of the record type does not explicitly state that the opcode is followed by an expression, then an expression cannot be used. Expressions are never used in load segments.

The operation codes and segment records are described in this section, listed in order of the opcodes. Table F-3 provides an alphabetical cross-reference between segment record types and opcodes. Library files consist of object segments and so can use any record type that can be used in an object segment. Table F-3 also lists the segment types in which each record type can be used.

■ **Table F-3** Segment-body record types

Record type	Opcode	Found in what segment types
ALIGN	$E0	object
BEXPR	$ED	object
cINTERSEG	$F6	load
CONST	$01–$DF	object
cRELOC	$F5	load
DS	$F1	all
END	$00	all
ENTRY	$F4	run-time library dictionary
EQU	$F0	object
EXPR	$EB	object
GEQU	$E7	object
GLOBAL	$E6	object
INTERSEG	$E3	load
LCONST	$F2	all
LEXPR	$F3	object
LOCAL	$EF	object

■ **Table F-3** Segment-body record types (Continued)

Record type	Opcode	Found in what segment types
MEM	$E8	object
ORG	$E1	object
RELEXPR	$EE	object
RELOC	$E2	load
STRONG	$E5	object
SUPER	$F7	load
USING	$E4	object
ZEXPR	$EC	object

The rest of this section defines each of the segment-body record types. The record types are listed in order of their opcodes.

Record type	Opcode	Explanation
END	$00	This record indicates the end of the segment.
CONST	$01–$DF	This record contains absolute data that needs no relocation. The operation code specifies how many bytes of data follow.
ALIGN	$E0	This record contains a number that indicates an alignment factor. The linker inserts as many 0 bytes as necessary to move to the memory boundary indicated by this factor. The value of this factor is in the same format as the ALIGN field in the segment header and cannot have a value greater than that in the ALIGN field. ALIGN must equal a power of 2.
ORG	$E1	This record contains a number that is used to increment or decrement the location counter. If the location counter is incremented (ORG is positive), 0's are inserted to get to the new address. If the location counter is decremented (ORG is a complement negative number of 2), subsequent code overwrites the old code.

(continued)

Record type **Opcode** **Explanation**

RELOC $E2

This is a relocation record, which is used in the relocation dictionary of a load segment. It is used to patch an address in a load segment with a reference to another address in the same load segment. It contains two 1-byte counts followed by two offsets. The first count is the number of bytes to be relocated. The second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, the number is shifted to the left, filling vacated bit positions with 0's (arithmetic shift left). If the bit-shift operator is (two's complement) negative, the number is shifted right (logical shift right) and 0-filled.

The first offset gives the location (relative to the start of the segment) of the first byte of the number that is to be patched (relocated). The second offset is the location of the reference relative to the start of the segment; that is, it is the value that the number would have if the segment containing it started at address $000000. For example, suppose the segment includes the following lines:

```
35   LABEL . . .
       .
       .
       .
400  LDA   LABEL+4
```

The RELOC record contains a patch to the operand of the LDA instruction. The value of the patch is LABEL+4, so the value of the last field in the RELOC record is $39—the value the patch would have if the segment started at address $000000. LABEL+4 is two bytes long; that is, the number of bytes to be relocated is 2. No bit-shift operation is needed. The location of the patch is 1025 ($401) bytes after the start of the segment (immediately after the LDA, which is one byte).

The RELOC record for the number to be loaded into the A register by this statement would therefore look like this (note that the values are stored low byte first, as specified by NUMSEX):

E2020001 04000039 000000

Record type **Opcode** **Explanation**

This sequence corresponds to the following values:

$E2	operation code
$02	number of bytes to be relocated
$00	bit-shift operator
$00000401	offset of value from start of segment
$00000039	value if segment started at $000000

◆ *Note:* Certain types of arithmetic expressions are illegal in a relocatable segment; specifically, any expression that the assembler cannot evaluate (relative to the start of the segment) cannot be used. The expression LAB|4 can be evaluated, for example, since the RELOC record includes a bit-shift operator. The expression LAB|4+4 cannot be used, however, because the assembler would have to know the absolute value of LAB to perform the bit-shift operation before adding 4 to it. Similarly, the value of LAB*4 depends on the absolute value of LAB and cannot be evaluated relative to the start of the segment, so multiplication is illegal in expressions in relocatable segments.

INTERSEG $E3

This record is used in the relocation dictionary of a load segment. It contains a patch to a long call to an external reference; that is, the INTERSEG record is used to patch an address in a load segment with a reference to another address in a different load segment. It contains two 1-byte counts followed by an offset, a 2-byte file number, a 2-byte segment number, and a second offset. The first count is the number of bytes to be relocated, and the second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, the number is shifted to the left, filling vacated bit positions with 0's (arithmetic shift left). If the bit-shift operator is (two's complement) negative, the number is shifted right (logical shift right) and 0-filled.

(continued)

Record type **Opcode** **Explanation**

The first offset is the location (relative to the start of the segment) of the (first byte of the) number that is to be relocated. If the reference is to a static segment, the file number, segment number, and second offset correspond to the subroutine referenced. (The linker assigns a file number to each load file in a program. This feature is provided primarily to support run-time libraries. In the normal case of a program having one load file, the file number is 1. The load segments in a load file are numbered by their relative locations in the load file, where the first load segment is number 1.) If the reference is to a dynamic segment, the file and segment numbers correspond to the jump-table segment, and the second offset corresponds to the call to the System Loader for that reference.

For example, suppose the segment includes an instruction such as

```
JSL EXT
```

The label `EXT` is an external reference to a location in a static segment.

If this instruction is at relative address $720 within its segment and `EXT` is at relative address $345 in segment $000A in file $0001, the linker creates an INTERSEG record in the relocation dictionary that looks like this (note that the values are stored low byte first, as specified by NUMSEX):
E3030021 07000001 000A0045 030000

This sequence corresponds to the following values:

$E3	operation code
$03	number of bytes to be relocated
$00	bit-shift operator
$00000721	offset of instruction's operand
$0001	file number
$000A	segment number
$00000345	offset of subroutine referenced

When the loader processes the relocation dictionary, it uses the first offset to find the JSL and patches in the address corresponding to the file number, segment number, and offset of the referenced subroutine.

Record type	Opcode	Explanation
		If the JSL is to an external reference in a dynamic segment, the INTERSEG records refer to the file number, segment number, and offset of the call to the System Loader in the jump-table segment. If the jump-table segment is in segment 6 of file 1, and the call to the System Loader is at relative location $2A45 in the jump-table segment, then the INTERSEG record looks like this (note that the values are stored low byte first, as specified by NUMSEX): E3030021 07000001 00060045 2A0000 This sequence corresponds to the following values: $E3 operation code $03 number of bytes to be relocated $00 bit-shift operator $00000721 offset of instruction's operand $0001 file number of jump-table segment $0006 segment number of jump-table segment $00002A45 offset of call to System Loader The jump-table segment entry that corresponds to the external reference EXT contains the following values: **User ID** $0001 file number $0005 segment number $00000200 offset of instruction call to System Loader INTERSEG records are used for any long-address reference to a static segment. See the section "Jump-Table Segment" later in this appendix for a discussion of the function of the jump-table segment.
USING	$E4	This record contains the name of a data segment. After this record is encountered, local labels from that data segment can be used in the current segment.
STRONG	$E5	This record contains the name of a segment that must be included during linking, even if no external references have been made to it. If you are using the APW assembler, the following statement generates a STRONG record: `DC      R'xxxx'` where *xxxx* is label.

(continued)

Record type	Opcode	Explanation
GLOBAL	$E6	This record contains the name of a global label followed by three attribute fields. The label is assigned the current value of the location counter. The first attribute field is two bytes long and gives the number of bytes generated by the line that defined the label. If this field is $FFFF, it indicates that the actual length is unknown but that it is greater than or equal to $FFFF. The second attribute field is one byte long and specifies the type of operation in the line that defined the label. The following type attributes are defined (uppercase ASCII characters with the high bit off): A address-type DC statement B Boolean-type DC statement C character-type DC statement D double-precision floating-point-type DC statement F floating-point-type DC statement G EQU or GEQU statement H hexadecimal-type DC statement I integer-type DC statement K reference-address-type DC statement L soft-reference-type DC statement M instruction N assembler directive O ORG statement P ALIGN statement S DS statement X arithmetic symbolic parameter Y Boolean symbolic parameter Z character symbolic parameter The third attribute field is one byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private. (See the section "Segment Types and Attributes" earlier in this appendix for a definition of private segments.)
GEQU	$E7	This record contains the name of a global label followed by three attribute fields and an expression. The label is given the value of the expression. The first attribute field is 2 bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is 1 byte long and specifies the type of operation in the line that defined the label, as listed in the

Record type	Opcode	Explanation
		discussion of the GLOBAL record. The third attribute field is 1 byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private. (See the section "Segment Types and Attributes" earlier in this appendix for a definition of private segments.)
MEM	$E8	This record contains two numbers that represent the starting and ending addresses of a range of memory that must be reserved. If the size of the numbers is not specified, the length of the numbers is defined by the NUMLEN field in the segment header.
EXPR	$EB	This record contains a 1-byte count followed by an expression. The expression is evaluated, and its value is truncated to the number of bytes specified in the count. The order of the truncation is from most significant to least significant.
ZEXPR	$EC	This record contains a 1-byte count followed by an expression. ZEXPR is identical to EXPR, except that any bytes truncated must be all 0's. If the bytes are not 0's, the record is flagged as an error.
BEXPR	$ED	This record contains a 1-byte count followed by an expression. BEXPR is identical to EXPR, except that any bytes truncated must match the corresponding bytes of the location counter. If the bytes don't match, the record is flagged as an error. This record allows the linker to make sure that an expression evaluates to an address in the current memory bank.
RELEXPR	$EE	This record contains a 1-byte length followed by an offset and an expression. The offset is NUMLEN bytes long. RELEXPR is used to generate a relative branch value that involves an external location. The length indicates how many bytes to generate for the instruction, the offset indicates where the origin of the branch is relative to the current location counter, and the expression is evaluated to yield the destination of the branch. For example, a BNE LOC instruction, where LOC is external, generates this record. For the 6502 and 65816 microprocessors, the offset is 1.

(continued)

Record type	Opcode	Explanation
LOCAL	$EF	This record contains the name of a local label followed by three attribute fields. The label is assigned the value of the current location counter. The first attribute field is two bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is one byte long and specifies the type of operation in the line that defined the label, as listed in the discussion of the GLOBAL record. The third attribute field is one byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private. (See the section "Segment Types and Attributes," earlier in this appendix, for a definition of private segments.) Some linkers (such as the APW Linker) ignore local labels from code segments and recognize local labels from other data segments only if a USING record was processed. See the preceding discussion of the USING statement.
EQU	$F0	This record contains the name of a local label followed by three attribute fields and an expression. The label is given the value of the expression. The first attribute field is two bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is one byte long and specifies the type of operation in the line that defined the label, as listed in the discussion of the GLOBAL record. The third attribute field is one byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private. (See the section "Segment Types and Attributes," earlier in this appendix, for a definition of private segments.)
DS	$F1	This record contains a number indicating how many bytes of 0's to insert at the current location counter.
LCONST	$F2	This record contains a 4-byte count followed by absolute code or data. The count indicates the number of bytes of data. The LCONST record is similar to CONST except that it allows for a much greater number of data bytes. Each relocatable load segment consists of LCONST records, DS records, and a relocation dictionary. See the discussions on INTERSEG records, RELOC records, and the relocation dictionary for more information.

Record type **Opcode** **Explanation**

LEXPR $F3 This record contains a 1-byte count followed by an expression. The expression is evaluated, and its value is truncated to the number of bytes specified in the count. The order of the truncation is from most significant to least significant.

Because the LEXPR record generates an intersegment reference, only simple expressions are allowed in the expression field, as follows:

```
LABEL ± const
LABEL| ± const
(LABEL ± const)| ± const
```

In addition, if the expression evaluates to a single label with a fixed, constant offset, and if the label is in another segment and that segment is a dynamic code segment, then the linker creates an entry for that label in the jump-table segment. (The jump-table segment provides a mechanism to allow dynamic loading of segments as they are needed—see the section "Load Files," later in this appendix.)

ENTRY $F4 This record is used in the run-time library entry dictionary; it contains a 2-byte number and an offset followed by a label. The number is the segment number. The label is a code-segment name or entry, and the offset is the relative location within the load segment of the label. Run-time library entry dictionaries are described in the section "Run-Time Library Files," later in this appendix.

cRELOC $F5 This record is the compressed version of the RELOC record. It is identical to the RELOC record, except that the offsets are two bytes long rather than four bytes. The cRELOC record can be used only if both offsets are less than $10000 (65,536). The following example compares a RELOC record and a cRELOC record for the same reference:

RELOC	cRELOC
$E2	$F5
$02	$02
$00	$00
$00000401	$0401
$00000039	$0039
(11 bytes)	(7 bytes)

(continued)

Record type **Opcode** **Explanation**

For an explanation of each line of these records, see the preceding discussion of the RELOC record.

cINTERSEG $F6

This record is the compressed version of the INTERSEG record. It is identical to the INTERSEG record, except that the offsets are two bytes long rather than four bytes, the segment number is one byte rather than two bytes, and this record does not include the 2-byte file number. The cINTERSEG record can be used only if both offsets are less than $10000 (65,536), the segment number is less than 256, and the file number associated with the reference is 1 (that is, the initial load file). References to segments in run-time library files must use INTERSEG records rather than cINTERSEG records.

The following example compares an INTERSEG record and a cINTERSEG record for the same reference:

INTERSEG	cINTERSEG
$E3	$F6
$03	$03
$00	$00
$00000720	$0720
$0001	
$000A	$0A
$00000345	$0345
(15 bytes)	(8 bytes)

For an explanation of each line of these records, see the preceding discussion of the INTERSEG record.

SUPER $F7

This is a supercompressed relocation-dictionary record. Each SUPER record is the equivalent of many cRELOC, cINTERSEG, and INTERSEG records. It contains a 4-byte length, a 1-byte record type, and one or more subrecords of variable size, as follows:

Opcode:	$F7
Length:	number of bytes in the rest of the record (4 bytes)
Type:	0–37 (1 byte)
Subrecords:	(variable size)

When SUPER records are used, some of the relocation information is stored in the LCONST record at the address to be patched.

Record type **Opcode** **Explanation**

The length field indicates the number of bytes in the rest of the SUPER record (that is, the number of bytes exclusive of the opcode and the length field).

The type byte indicates the type of SUPER record. There are 38 types of SUPER record:

Value	SUPER record type
0	SUPER RELOC2
1	SUPER RELOC3
2–37	SUPER INTERSEG1–SUPER INTERSEG36

SUPER RELOC2: This record can be used instead of cRELOC records that have a bit-shift count of zero and that relocate two bytes.

SUPER RELOC3: This record can be used instead of cRELOC records that have a bit-shift count of zero and that relocate three bytes.

SUPER INTERSEG1: This record can be used instead of cINTERSEG records that have a bit-shift count of zero and that relocate three bytes.

SUPER INTERSEG2 through SUPER INTERSEG12: The number in the name of the record refers to the file number of the file in which the record is used. For example, to relocate an address in file 6, use a SUPER INTERSEG6 record. These records can be used instead of INTERSEG records that meet the following criteria:

- Both offsets are less than $10000.
- The segment number is less than 256.
- The bit-shift count is 0.
- The record relocates 3 bytes.
- The file number is from 2 through 12.

SUPER INTERSEG13 through SUPER INTERSEG24: These records can be used instead of cINTERSEG records that have a bit-shift count of zero, that relocate two bytes, and that have a segment number of *n* minus 12, where *n* is a number from 13 to 24. For example, to replace a cINTERSEG record in segment 6, use a SUPER INTERSEG18 record.

(continued)

Record type	Opcode	Explanation

SUPER INTERSEG25 through SUPER INTERSEG36: These records can be used instead of cINTERSEG records that have a bit-shift count of $F0 (–16), that relocate two bytes, and that have a segment number of *n* minus 24, where *n* is a number from 25 to 36. For example, to replace a cINTERSEG record in segment 6, use a SUPER INTERSEG30 record.

Each subrecord consists of either a 1-byte offset count followed by a list of 1-byte offsets, or a 1-byte skip count.

Each offset count indicates how many offsets are listed in this subrecord. The offsets are one byte each. Each offset corresponds to the low byte of the first (2-byte) offset in the equivalent INTERSEG, cRELOC, or cINTERSEG record. The high byte of the offset is indicated by the location of this offset count in the SUPER record: Each subsequent offset count indicates the next 256 bytes of the load segment. Each skip count indicates the number of 256-byte pages to skip; that is, a skip count indicates that there are no offsets within a certain number of 256-byte pages of the load segment.

For example, if patches must be made at offsets 0020, 0030, 0140, and 0550 in the load segment, the subrecords would include the following fields:

2 20 30	the first 256-byte page of the load segment has two patches: one at offset 20 and one at offset 30
1 40	the second 256-byte page has one patch at offset 40
skip-3	skip the next three 256-byte pages
1 50	the sixth 256-byte page has one patch at offset 50

In the actual SUPER record, the patch count byte is the number of offsets minus one, and the skip count byte has the high bit set. A SUPER INTERSEG1 record with the offsets in the preceding example would look like this:

$F7	opcode
$00000009	number of bytes in the rest of the record
$02	INTERSEG1-type SUPER record
$01	the first 256-byte page has two patches

Record type	Opcode	Explanation
		$20 patch the load segment at offset $0020 $30 patch the segment at $0030 $00 the second page has one patch $40 patch the segment at $0140 $83 skip the next three 256-byte pages $00 the sixth page has one patch $50 patch the segment at $0550 A comparison with the RELOC record shows that a SUPER RELOC record is missing the offset of the reference. Similarly, the SUPER INTERSEG1 through SUPER INTERSEG12 records are missing the segment number and offset of the subroutine referenced. The offsets (which are two bytes long) are stored in the LCONST record at the "to be patched" location. For the SUPER INTERSEG1 through SUPER INTERSEG12 records, the segment number is stored in the third byte of the "to be patched" location. For example, if the example given in the discussion of the INTERSEG record were instead referenced through a SUPER INTERSEG1 record, the value $0345 (the offset of the subroutine referenced) would be stored at offset $0721 in the load segment (the offset of the instruction's operand). The segment number ($0A) would be stored at offset $0723, as follows: `4503 0A`
General	$FB	This record contains a 4-byte count indicating the number of bytes of data that follow. This record type is reserved for use by Apple Computer, Inc.
Experimental	$FC–$FF	These records contain a 4-byte count indicating the number of bytes of data that follow. These record types are reserved by Apple Computer for use in system development.

Expressions

Several types of OMF records contain expressions. Expressions form an extremely flexible reverse-Polish stack language that can be evaluated by the linker to yield numeric values such as addresses and labels. Each expression consists of a series of operators and operands together with the values on which they act.

An operator takes one or two values from the evaluation stack, performs some mathematical or logical operation on them, and places a new value onto the evaluation stack. The final value on the evaluation stack is used as if it were a single value in the record. Note that this evaluation stack is purely a programming concept and does not relate to any hardware stack in the computer. Each operation is stored in the object module file in postfix form; that is, the value or values come first, followed by the operator. For example, since a binary operation is stored as *Value1 Value2 Operator,* the operation *Num1 minus Num2* is stored as

```
Num1Num2-
```

The operators are as follows:

- **Binary math operators:** These operators take two numbers (as two's-complement signed integers) from the top of the evaluation stack, perform the specified operation, and place the single-integer result back on the evaluation stack. The binary math operators include

$01 addition	(+)
$02 subtraction	(-)
$03 multiplication	(*)
$04 division	(/, DIV)
$05 integer remainder	(//, MOD)
$07 bit shift	(<<, >>)

  The subtraction operator subtracts the second number from the first number. The division operator divides the first number by the second number. The integer-remainder operator divides the first number by the second number and returns the unsigned integer remainder to the stack. The bit-shift operator shifts the first number by the number of bit positions specified by the second number. If the second number is positive, the first number is shifted to the left, filling vacated bit positions with 0's (arithmetic shift left). If the second number is negative, the first number is shifted right, filling vacated bit positions with 0's (logical shift right).

- **Unary math operator:** A unary math operator takes a number as a two's-complement signed integer from the top of the evaluation stack, performs the operation on it, and places the integer result back on the evaluation stack. The only unary math operator currently available is

$06 negation	(−)

- **Comparison operators:** These operators take two numbers as two's-complement signed integers from the top of the evaluation stack, perform the comparison, and place the single-integer result back on the evaluation stack. Each operator compares the second number in the stack (TOS – 1) with the number at the top of the stack (TOS). If the comparison is TRUE, a 1 is placed on the stack; if FALSE, a 0 is placed on the stack. The comparison operators include

$0C	less than or equal to	(<=, ≤)
$0D	greater than or equal to	(>=, ≥)
$0E	not equal	(<>, ≠, !=)
$0F	less than	(<)
$10	greater than	(>)
$11	equal to	(= or ==)

- **Binary logical operators:** These operators take two numbers as Boolean values from the top of the evaluation stack, perform the operation, and place the single Boolean result back on the stack. Boolean values are defined as being FALSE for the number 0 and TRUE for any other number. Logical operators always return a 1 for TRUE. The binary logical operators include

$08	AND	(**, AND)
$09	OR	(++, OR, |)
$0A	EOR	(--, XOR)

- **Unary logical operator:** A unary logical operator takes a number as a Boolean value from the top of the evaluation stack, performs the operation on it, and places the Boolean result back on the stack. The only unary logical operator currently available is

$0B	NOT	(¬, NOT)

- **Binary bit operators:** These operators take two numbers as binary values from the top of the evaluation stack, perform the operation, and place the single binary result back on the stack. The operations are performed on a bit-by-bit basis. The binary bit operators include

$12	Bit AND	(logical AND)
$13	Bit OR	(inclusive OR)
$14	Bit EOR	(exclusive OR)

- **Unary bit operator:** This operator takes a number as a binary value from the top of the evaluation stack, performs the operation on it, and places the binary result back on the stack. The unary bit operator is

$15	Bit NOT	(complement)

- **Termination operator:** All expressions end with the termination operator $00.

An **operand** causes some value, such as a constant or a label, to be loaded onto the evaluation stack. The operands are as follows:

- **Location-counter operand ($80):** This operand loads the value of the current location counter onto the top of the stack. Because the location counter is loaded before the bytes from the expression are placed into the code stream, the value loaded is the value of the location counter before the expression is evaluated.
- **Constant operand ($81):** This operand is followed by a number that is loaded on the top of the stack. If the size of the number is not specified, its length is specified by the NUMLEN field in the segment header.
- **Label-reference operands ($82–$86):** Each of these operand codes is followed by the name of a label and is acted on as follows:

  $82 Weak reference (see the following note).

  $83 The value assigned to the label is placed on the top of the stack.

  $84 The length attribute of the label is placed on the top of the stack.

  $85 The type attribute of the label is placed on the top of the stack. (Type attributes are listed in the discussion of the GLOBAL record in the section "Segment Body" earlier in this appendix.)

  $86 The count attribute is placed on the top of the stack. The count attribute is 1 if the label is defined and 0 if it is not.
- **Relative offset operand ($87):** This operand is followed by a number that is treated as a displacement from the start of the segment. Its value is added to the value that the location counter had when the segment started, and the result is loaded on the top of the stack.

◆ *Note:* The operand code $82 is referred to as the weak reference. The weak reference is an instruction to the linker that asks for the value of a label, if it exists. It is not an error if the linker cannot find the label. However, the linker does not load a segment from a library if only weak references to it exist. If a label does not exist, a 0 is loaded onto the top of the stack. This operand is generally used for creating jump tables to library routines that may or may not be needed in a particular program.

Example

Assume your assembly-language program contains the following line, where MSG4 and MSG3 are global labels:

```
LDX    #MSG4-MSG3
```

This line is assembled into two OMF records:

```
CONST ($01)        A2
EXPR ($EB)         02 : MSG4MSG3-
```

In hexadecimal format, these records appear as follows:

```
01 A2          ."
EB 02 83 04 4D 53 47 34 83 04 4D 53 47 33 02 00     k...MSG4..MSG3..
```

The initial $01 is the OMF opcode for a 1-byte constant. The $A2 is the 65816 opcode for the LDX instruction. The $EB is the OMF opcode for an EXPR record, which is followed by a 1-byte count indicating the number of bytes to which the expression is to be truncated ($02 in this case). The next number, $83, is a label-reference operand for the first label in the expression, indicating that the value assigned to the label (MSG4) is to be placed on top of the evaluation stack. Next is a length byte ($04), followed by MSG4 spelled out in ASCII codes.

The next sequence of codes, starting with $83, places the value of MSG3 on the evaluation stack. Finally, the expression-operator code $02 indicates that subtraction is to be performed, and the termination operator ($00) indicates the end of the expression.

◆ *Note:* You can use the DumpObj tool provided with some development environments to examine the contents of any OMF file. DumpObj can list the header contents of each segment and can list the body of each segment in OMF format, 65816 disassembly format, or as hexademical codes. See your development-environment manuals for instructions.

Object files

Object files (file type $B1) are created from source files by a compiler or assembler. Object files can contain any of the OMF record types except INTERSEG, cINTERSEG, RELOC, cRELOC, SUPER, and ENTRY. Object files can contain unresolved references, because all references are resolved by the linker. If you are writing a compiler for the Apple IIGS, you can use the DumpObjIIGS tool to examine the contents of a variety of object files to get an idea of their content and structure.

Library files

Library files (file type $B2) contain object segments that the linker can search for external references. Usually, these files contain general routines that can be used by more than one application. The linker extracts from the library file any object segment that contains an unresolved global definition that was referenced during the link. This segment is then added to the load segment that the linker is currently creating.

Library files differ from object files in that each library file includes a segment called the *library dictionary segment* (segment kind = $08). The library dictionary segment contains the names and locations of all segments in the library file. This information allows the linker to scan the file quickly for needed segments. Library files are created from object files by a MakeLib tool (provided with a development environment). The format of the library dictionary segment is illustrated in Figure F-3.

The library dictionary segment begins with a segment header, which is identical in form to other segment headers. The BYTECNT field indicates the number of bytes in the library dictionary segment, including the header. The body of the library dictionary segment consists of three LCONST records, in this order:

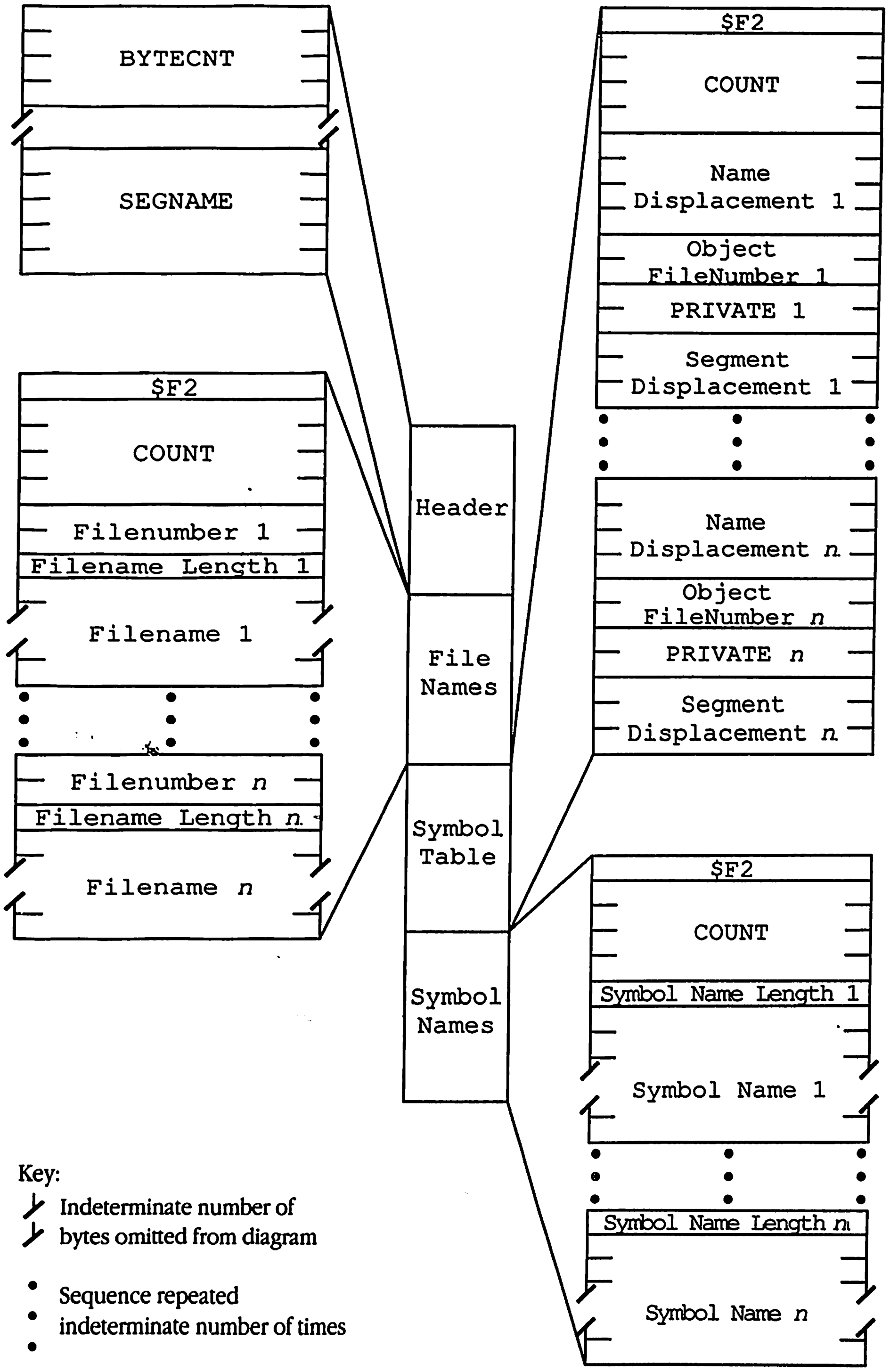
1. Filenames
2. Symbol table
3. Symbol names

The filenames record consists of one or more subrecords, each consisting of a 2-byte file number followed by a filename. The filename is in Pascal string format, that is, a length byte indicating the number of characters, followed by an ASCII string. The filenames are the full pathnames of the object files from which the segments in this library file were extracted. The file numbers are assigned by the MakeLib program and used only within the library file. These file numbers are not related to the load-file numbers in the pathname table.

The symbol table record consists of a cross-reference between the symbol names in the symbol-names record and the object segments in which the symbol names occur. For each global symbol in the library file, the symbol table record contains the following components:

1. A 4-byte displacement into the symbol names record, indicating the start of the symbol name.
2. The 2-byte file number of the file in which the name occurred. This is the file number assigned by the MakeLib utility and used in the filenames record of this library dictionary segment.

■ **Figure F-3** The format of a library dictionary segment



3. A 2-byte flag, the private flag. If this flag equals 1, the symbol name is valid only in the object file in which it occurred (that is, the symbol name was in a private segment). If this flag equals 0, the symbol name is not private.
4. A 4-byte displacement into the library file indicating the beginning of the object segment in which the symbol occurs. The displacement is to the beginning of the segment even if the symbol occurs inside the segment; the location within the segment is resolved by the linker.

The symbol names record consists of a series of symbol names; each symbol name consists of a length byte followed by up to 255 ASCII characters. All global symbols that appear in an object segment, including entry points and global equates, are placed in the library dictionary segment. Duplicate symbols are not allowed.

Load files

Load files (file types $B3 through $BE) contain the load segments that are moved into memory by the System Loader. They are created by a linker from object files and library files.

◆ *Note:* ExpressLoad does not support file type $BE.

Load files conform to the object module format but are restricted to a small subset of that format. Because the segments must be quickly relocated and loaded, they cannot contain any unresolved symbolic information.

All load files are composed of load segments. The format of each load segment is a loadable binary memory image followed by a relocation dictionary. Load files can contain any of several special segment types:

- jump-table segment
- pathname segment
- initialization segment
- direct-page/stack segment

Each of these segment types is described in the following sections.

The load segments in a load file are numbered by their relative location in the load file, where the first load segment is number 1. The segment number is used by the System Loader to find a specific segment in a load file.

Memory image and relocation dictionary

Each load segment consists of two parts, in this order:

1. A memory image comprising long-constant (LCONST) records and define-storage (DS) records. These records contain all of the code and data that do not change with load address (these records reserve space for location-dependent addresses). The DS records are inserted by the linker (in response to DS records in the object file) to reserve large blocks of space, rather than putting large blocks of 0's in the load file.
2. A relocation dictionary that provides the information necessary to patch the LCONST records at load time. The relocation dictionary contains relocation (RELOC, cRELOC, or SUPER RELOC) records and intersegment (INTERSEG, cINTERSEG, or SUPER INTERSEG) records.

When the System Loader loads the segment into memory, it loads each LCONST record or DS record in one piece; then it processes the relocation dictionary. The relocation dictionary includes only RELOC (or cRELOC or SUPER RELOC) and INTERSEG (or cINTERSEG or SUPER INTERSEG) records. The RELOC records provide the information the loader needs to recalculate the values of location-dependent local references, and the INTERSEG records provide the information it needs to transfer control to external references. For more information, see the discussions of the RELOC and INTERSEG records in the section "Segment Body," earlier in this appendix.

Jump-table segment

The jump-table segment, when used, is the segment of a load file that contains the calls to the System Loader to load dynamic segments. Each time the linker comes across a statement that references a label in a dynamic segment, it generates an entry in the jump-table segment for that label (it also creates an entry in the relocation dictionary). The entry in the jump-table segment contains the file number, segment number, and offset of the reference in the dynamic segment, plus a call to the System Loader to load the segment. The relocation dictionary entry provides the information the loader needs to patch a call to the jump-table segment into the memory image.

The segment type of the jump-table segment is KIND = $02. There is one jump-table segment per load file; it is a static segment, and it is loaded into memory at program boot time at a location determined by the Memory Manager. The System Loader maintains a list, called the jump-table directory (or just the jump table), of the jump-table segments in memory.

Each entry in the jump-table segment corresponds to a call to an external (intersegment) routine in a dynamic segment. The jump-table segment initially contains entries in the unloaded state. When the external call is encountered during program execution, a jump to the jump-table segment occurs. The code in the jump-table segment entry, in turn, jumps to the System Loader. The System Loader figures out which segment is referenced and loads it. Next, the System Loader changes the entry in the jump-table segment to the loaded state. The entry stays in the loaded state as long as the corresponding segment is in memory. If the application tells the System Loader to unload a segment, all jump-table segment entries that reference that segment are changed to their unloaded states.

Unloaded state

The unloaded state of a jump-table segment entry contains the code that calls the System Loader to load the needed segment. An entry contains the following fields:

- user ID (two bytes)
- load-file number (two bytes)
- load-segment number (two bytes)
- load-segment offset (four bytes)
- JSL to jump-table load function (four bytes)

The user ID field is reserved for the identification number assigned to the program by the UserID Manager; until initial load time, this field is 0. The load-file number, load-segment number, and load-segment offset refer to the location of the external reference. The rest of the entry is a call to the System Loader jump-table load function (an internal routine). The user ID and the address of the load function are patched by the System Loader during initial load. See Chapter 8 for information about the jump-table load function. A load-file number of 0 indicates that there are no more entries in this jump-table segment. (There may be other jump-table segments for this program, however—each load file that is part of a program has its own jump-table segment.)

Loaded state

The loaded state of a jump-table segment entry is identical to the unloaded state except that the JSL to the System Loader jump-table load function is replaced by a JML to the external reference. A loaded entry contains the following fields:

- user ID (two bytes)
- load-file number (two bytes)
- load-segment number (two bytes)
- load-segment offset (four bytes)
- JML to external reference (four bytes)

◆ *Note:* In versions 1.0 and 2.0 of the OMF, the jump-table segment starts with eight bytes of zeros. In future versions of the OMF, these 0's may be eliminated.

Pathname segment

The pathname segment is a segment in a load file that is created by the linker to help the System Loader find the load segments of run-time library files that must be loaded dynamically. It provides a cross-reference between file numbers and file pathnames. The segment type of the pathname segment is KIND = $04. When the loader processes the load file, it adds the information in the pathname segment to the pathname table that it maintains in memory. Pathname tables are described in Chapter 8.

The pathname segment contains one entry for each load file and for each run-time library file referenced in a load file. The format of each entry is as follows:

file number (two bytes)

file date and time (eight bytes)

file pathname (length byte and ASCII string)

The file number is a number assigned by the linker to a specific load file. File number 1 is reserved for the load file in which the pathname segment resides (usually the load file of the application program). A file number of 0 indicates that there are no more entries in this pathname segment.

The file date and time are directory items retrieved by the linker during the link process. The System Loader compares these values with the directory of the run-time library file at run time. If they are not the same, the System Loader does not load the requested load segment, thus ensuring that the run-time library file used at link time is the same as the one loaded at execution time.

The file pathname is the pathname of the load file. The pathname is listed as a Pascal-type string: that is, a length byte followed by an ASCII string. A pathname segment created by the linker may contain prefix designators.

Initialization segment

The initialization segment is an optional segment in a load file. When the System Loader encounters an initialization segment during the initial loading of segments, it transfers control to the initialization segment. After the initialization segment returns control to the System Loader, the loader continues the normal initial load of the remaining segments in the load file. The segment type of the initialization segment is KIND = $10.

You might use an initialization segment, for example, to initialize the graphics environment of an application and to display a "splash screen" (such as a copyright message and company logo) for the duration of the program load.

The initialization segment does not have to be the first segment loaded, there may be more than one initialization segment, and an initialization segment can make references to other segments previously loaded.

The initialization segment must obey the following rules:

- It must not reference any segments not yet loaded.
- It must exit with an RTL instruction.
- It must not change the stack pointer.
- It must not use the current direct page. To avoid writing over a portion of the direct page being used by the loader, the initialization segment must allocate its own direct page if it needs direct-page space.

◆ *Note:* Initialization segments are reexecuted during the restart of an application from memory.

Direct-page/stack segments

The Apple IIGS stack can be located anywhere in the lower 48 KB of bank $00 and can be any size up to 48 KB. The direct page is the Apple IIGS equivalent of the zero page of 8-bit Apple II computers; the direct page can also be located anywhere in the lower 48 KB of bank $00. Like the zero page, the direct page occupies 256 bytes of memory; on the Apple IIGS, however, a program can move its direct page while it is running. Consequently, a given program can use more than 256 bytes of memory for direct-page functions.

Each program running on the Apple IIGS reserves a portion of bank $00 as a combined direct-page/stack space. Because more than one application can be loaded in memory at one time on the Apple IIGS, more than one stack and one direct page could be in bank $00 at a given time. Furthermore, some applications may place some of their code in bank $00. A given program should therefore probably not use more than about 4 KB for its direct-page/stack space.

When an instruction uses one of the direct-page addressing modes, the effective address is calculated by adding the value of the operand of the instruction to the value in the direct-page register. The stack pointer, on the other hand, is decremented each time a stack-push instruction is executed. The convention used on the Apple IIGS, therefore, is for the direct page to occupy the lower part of the direct-page/stack space, whereas the stack grows downward from the top of the space.

△ **Important** GS/OS provides no mechanism for detecting stack overflow or underflow, or collision of the stack with the direct page. Your program must be carefully designed to make sure those conditions cannot occur. △

If you do not define a direct-page/stack segment in your program, GS/OS assigns 4 KB of direct-page/stack space when the System Loader InitialLoad or Restart call is executed. To specify the size and contents of the direct-page/stack space, follow the procedures outlined in Chapter 2, "GS/OS and Its Environment."

Run-time library files

Run-time library files (file type $B4) contain dynamic load segments that the System Loader can load when these segments are referenced through the jump table. Usually, run-time library files contain general routines that can be used by more than one application.

When you include a run-time library file while linking, the file is scanned by the linker during the link process. When the linker finds a referenced segment in the run-time library file, it generates an INTERSEG reference to the segment in the relocation dictionary and adds an entry to the jump-table segment for that file. The linker also adds the pathname of the run-time library file to the pathname table if it has not already done so. It does not extract the segment from the file and place it in the file that referenced it, as it does for ordinary library files. In other words, references to segments in run-time library files are treated by the linker like references to other dynamic segments, except that the run-time library file segments are in a file other than the currently executing load file.

The first load segment of the run-time library file contains all the information the linker needs to find referenced segments; it is not necessary for the linker to scan every subroutine in every segment each time a subroutine is referenced. The first segment contains a table of ENTRY records, each one corresponding to a segment name or global reference in the run-time library file.

Run-time library files are typically created from corresponding object files by specifying an option to a linker command.

Shell applications

Shell applications (file type $B5) are executable load files that are run under an Apple IIGS shell program, such as the APW Shell. The shell calls the System Loader's InitialLoad function and transfers control to the shell application by means of a JSL instruction, rather than launching the program through the GS/OS Quit function. Therefore, the shell does not shut down, and the program can use shell facilities during execution. The program returns control to the shell with an RTL instruction, or with a GS/OS Quit call if the shell intercepts and acts on GS/OS calls. (Development-environment shells might intercept GS/OS Quit calls.) Shell applications should use standard Text Tool Set calls for all nongraphics I/O. The shell program is responsible for initializing the Text Tool Set routines.

- ◆ *Note:* A load file of file type $B5 can be launched by GS/OS by way of the Quit call if it requires no support other than standard input from the keyboard and output to the screen. GS/OS initializes the Text Tool Set to use the Pascal I/O drivers (see the *Apple IIGS Toolbox Reference*) for the keyboard and 80-column screen. Only $B5 files that end in a GS/OS Quit call can be run in this way.

As soon as a shell application is launched, it should check the X and Y registers for a pointer to the shell-identifier string and input line. The X register holds the high word and the Y register holds the low word of this pointer. The shell program is responsible for loading this pointer into the index registers and for placing the following information in the area pointed to:

1. An 8-byte ASCII string containing an identifier for the shell. (The identifier for the APW Shell, for example, is BYTEWRKS.) The shell application should check this identifier to make sure that it has been launched by the correct shell, so that the environment it needs is in place. If the shell identifier is not correct, the shell application should write an error message to standard error output (normally the screen) and then exit with an RTL instruction (or a GS/OS Quit call if the shell intercepts GS/OS calls).
2. A null-terminated ASCII string containing the input line for the shell application. The shell program can strip any I/O redirection or pipeline commands from the input line, since those commands are intended for the shell itself, but must pass on all input parameters intended for the shell application.

The shell program must request a user ID for the shell application; the user ID is passed in the accumulator. The shell must set up a direct-page and stack area for the shell application. The shell places the address of the start of the direct-page/stack space in the direct-page (D) register and sets the stack pointer (S register) to point to the last byte of the block. If the shell application does not have a direct-page/stack segment, the shell should follow the same conventions used by GS/OS for default direct-page/stack allocation. See the section "Direct-Page/Stack Segments" earlier in this appendix, and Chapter 2 for more information about direct-page and stack allocation.

◆ *Note:* GS/OS does not support the identifier string or input line. If the shell application is launched by GS/OS, the X and Y registers contain 0's.

Some shell applications may launch other programs; for example, a shell nested within another shell would be a shell application. When a shell application requests a user ID for a program, the calling program is responsible for intercepting GS/OS Quit calls and system resets, so that it can remove from memory all memory buffers with that user ID before passing control to the shell.

A shell application should use the following procedure to quit:

1. If the shell application has launched any programs, it must call the System Loader's UserShutdown function to shut down those programs.
2. The shell application should release any memory buffers that it has requested and dispose of their handles.

3. The shell application must place an error code in the accumulator. If no error occurred, the error code should be $0000. The error code $FFFF is used as a general (nonspecific) error code. For a shell program you write, you can define any other error codes you want to use, and you can handle them in any way you wish.
4. The shell application should execute an RTL or a GS/OS Quit call. If the program ends in a Quit call, the shell program that launched the shell application is responsible for intercepting the Quit call, releasing all memory buffers associated with that shell application, and performing any other system tasks normally done by GS/OS in response to a Quit call.

△ **Important** When a shell launches a shell application, the address of the shell program is not pushed onto the GS/OS Quit Return stack; therefore, the shell itself must handle the shell application's Quit call, or control is not returned to the shell. To intercept the Quit call, the shell program must intercept all GS/OS calls. The shell may pass on any other operating system calls to GS/OS, but it must handle Quit calls itself. If the shell you are using does not handle GS/OS calls in this fashion, the shell application must end in an RTL instruction. △

Glossary

abstract file system: The generic file interface that GS/OS provides to applications. Individual **file system translators** convert file information in abstract format into formats meaningful to specific file systems.

AppleShare-aware program: A program that can be executed from an AppleShare file server.

AppleShare FST: The part of the GS/OS file system level that implements AppleShare capabilities for GS/OS.

Apple II: Any computer from the Apple II family, including the Apple II Plus, the Apple IIc, the Apple IIe, and the Apple IIGS.

Apple IIGS Toolbox: An extensive set of routines (in ROM and in RAM) that provide easy program access to hardware and firmware, and facilitate the writing of applications that display the **desktop interface.**

application level: One of the three **interface levels** of GS/OS. The application level accepts calls from applications, and may send them on to the file system level or the device level.

application-level calls: The calls an application makes to GS/OS to gain access to files or devices or to set or get system information. Application-level calls include **standard GS/OS calls** and **ProDOS 16–compatible calls**.

associated file: In the ISO 9660 file format, a file analogous to the resource fork of a GS/OS **extended file.**

block: (1) A unit of data storage or transfer, typically 512 bytes. (2) A contiguous, page-aligned region of computer memory of arbitrary size.

block device: A device that reads and writes information in multiples of one block of characters at a time. Disk drives are block devices.

cache: A portion of the Apple IIGS memory set aside for temporary storage of frequently accessed disk blocks. By reading blocks from the cache instead of from disk, GS/OS can greatly speed I/O.

cache controller: The part of GS/OS that sets the cache size based on the amount of system RAM installed. The minimum value is 0 KB, and the maximum is the amount of RAM in the system minus 256 KB.

call: (v.) To execute an operating system routine. (n.) The routine so executed.

caller: A program, such as an application, that makes a call to the operating system.

character device: A device that reads or writes a stream of characters in order, one at a time. The keyboard, screen, printer, and communications port are character devices.

character device driver: A driver that controls a character device.

Character FST: The part of the GS/OS file system level that makes character devices appear to application programs as if they were sequential files.

class 0 calls: See **ProDOS 16–compatible calls.**

class 1 calls: See **standard GS/OS calls.**

console: The main terminal—that is, the keyboard and screen—of the computer. GS/OS considers the console to be a single device.

console driver: A GS/OS character device driver that allows GS/OS to read data from the keyboard or write it to the screen.

controlling program: A program that loads and runs other programs, without itself relinquishing control. A controlling program is responsible for shutting down its subprograms and freeing their memory space when they are finished. A shell, for example, is a controlling program.

data fork: The part of an extended file that contains data created by an application.

desktop interface: The visual interface that an application using the Apple IIGS Toolbox presents to the user. It is characterized by menus, the mouse, icons, and windows.

device: A piece of equipment that transfers information to or from the Apple IIGS. Disk drives, printers, joysticks, and the mouse are external devices. The keyboard and screen are also a device (the **console**).

device call: See **GS/OS device call.**

device level: One of the three **interface levels** of GS/OS. The device level mediates between the **file system level** and individual device drivers.

directory: See **directory file.**

directory entry: See **file entry.**

directory file: A file that describes and points to other files on disk. Compare **standard file, extended file.**

direct page: An area of memory used for fast access by the microprocessor; the Apple IIGS equivalent to the standard Apple II **zero page.** The difference is that it need not be page zero in memory.

direct-page/stack segment: A load segment used to preset the location and contents of the direct page and stack for an application.

disk cache: See **cache.**

driver calls: A class of low-level calls in GS/OS, not accessible to applications. Driver calls are calls made from within GS/OS to device drivers.

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare **static segment.**

EOF: A count indicating the size of the file in bytes.

ExpressLoad: One of the two **GS/OS Loaders**.

extended file: A named collection of data consisting of two sequences of bytes, referred to by a single directory entry. The two different byte sequences of an extended file are called the **data fork** and the **resource fork.**

file: An ordered collection of bytes that has several attributes under GS/OS, including a name and a file type.

file entry: A component of a directory file that describes and points to another file on disk.

filename: The string of characters that identifies a particular file within its directory. Compare **pathname.**

file system level: One of the three **interface levels** of GS/OS. The file system level consists of **file system translators** (FSTs), which take calls from the **application level,** convert them to a specific file system format, and send them on to the **device level.**

file system translator: A component of GS/OS that converts application calls into a specific file system format before sending them on to device drivers. FSTs allow an application to use the same calls to read and write files for any number of file systems.

FST: See **file system translator.**

FSTSpecific: A standard GS/OS call whose function is defined individually for each FST.

full pathname: A volume name or a device name followed by a series of zero or more filenames, each preceded by the same separator, and ending with the name of a directory file, standard file, or extended file.

generated drivers: Drivers that are constructed by GS/OS itself, to provide a GS/OS interface to preexisting, usually ROM-based, peripheral-card drivers.

GS/OS: A 16-bit operating system developed for the Apple IIGS computer. GS/OS replaces ProDOS 16 as the preferred Apple IIGS operating system.

GS/OS calls: See **standard GS/OS calls.**

GS/OS device call: Any of a subset of the standard GS/OS calls that bypass the file level altogether, allowing applications to access devices directly.

GS/OS driver calls: See **driver calls.**

GS/OS Loaders: The programs that load all other programs into memory and prepare them for execution. There are two loaders: **ExpressLoad** and **System Loader**.

GS/OS string: An ASCII character string preceded by a 2-byte word whose numeric value is the number of 1-byte characters in the string. A GS/OS string can be much longer than a **Pascal string.**

hierarchical file system: A file system that contains both normal files that contain data or applications, and special files called **directories**.

High Sierra FST: The part of the GS/OS file system level that gives applications transparent access to files stored on optical compact disks (CD-ROM) in the most commonly used file formats: **High Sierra** and **ISO 9660**.

High Sierra Group format: A common file format for files on CD-ROM compact discs. Similar to the **ISO 9660** international standard format.

interface level: A conceptual division in the organization of GS/OS. GS/OS has three interface levels—the **application level,** the **file system level,** and the **device level.** The application level and the device level are external interfaces, whereas the file system level is internal to GS/OS.

interrupt: A hardware signal sent from an external or internal device to the CPU. When the CPU receives an interrupt, it suspends execution of the current program, saves the program's state, and transfers control to an **interrupt handler.**

interrupt dispatching: The process of handing control to the appropriate interrupt handler after an interrupt occurs.

interrupt handler: A program that executes in response to a hardware interrupt. Interrupts and interrupt handlers are commonly used by device drivers to operate their devices more efficiently and to make possible simple background tasks such as printer spooling. Compare **signal handler.**

interrupt source: Any device that can generate an interrupt, such as the mouse or serial ports.

ISO 9660 format (International Standards Organization's 9660): An international standard that specifies volume and file structure for CD-ROM discs. ISO 9660 is similar to the **High Sierra** format.

jump table: A mechanism whereby segments in memory can trigger the loading of other segments not yet in memory.

jump-table segment: A segment in a load file that contains all references to dynamic segments that may be called during execution of that load file. The jump-table segment is created by the linker. In memory, the loader combines all jump-table segments it encounters into the **jump table.**

length byte: A byte that specifies the length of a string in bytes.

length word: Two bytes that specify the length of a string in bytes.

library file: An object file containing program segments, each of which can be used in any number of programs. The **linker** can search through the library file for segments that have been referenced in the program source file.

linker: A program that combines files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

load file: The output of the linker. Load files contain memory images that the **System Loader** can load into memory, together with relocation dictionaries that the loader uses to relocate references.

loaded drivers: Drivers that are written to work directly with GS/OS, and that are usually loaded in from the system disk at boot time.

long prefix: A GS/OS prefix whose maximum total length is approximately 8,000 characters. Prefix designators `8/` through `31/` refer to long prefixes. Compare **short prefix.**

mark: A byte count indicating the current position in the file.

Memory Manager: An Apple IIGS tool set that controls all allocation and deallocation of memory.

object files: The files that are the output from an assembler or compiler and are the input to a linker.

object module format (OMF): The general format followed by Apple IIGS object files, library files, and load files.

OMF file: A file in object module format (an object file, library file or load file).

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

parameter count: A word-length input value to a standard GS/OS call that specifies the total number of parameters in the block.

partial pathname: This part of the pathname always contains the filename and one or more directory names up to, but not including, the volume name or device name.

Pascal string: An ASCII character string preceded by a single byte whose numeric value is the number of characters in the string. Pascal strings are limited to a maximum of 255 characters. Compare **GS/OS string.**

pathname: The complete name by which a file is specified. It is a sequence of filenames separated by **pathname separators,** starting with the filename of the volume directory and following the path through any subdirectories that a program must follow to locate the file. Compare **filename.**

pathname separator: The slash character (/) or colon (:). Pathname separators separate filenames in a pathname.

prefix: A portion of a pathname starting with a volume name and ending with a subdirectory name. A GS/OS prefix always starts with a pathname separator because a volume name always starts with a separator.

prefix designator: A number (0–31) or the asterisk character (*), followed by a pathname separator. Prefix designators are a shorthand method for referring to prefixes.

prefix number: See **prefix designator.**

ProDOS: (1) 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, SOS, or GS/OS. (2) The **ProDOS file system.**

ProDOS 8: The 8-bit ProDOS operating system, originally developed for standard Apple II computers but compatible with the Apple IIGS. In some earlier Apple II documentation, ProDOS 8 is called simply ProDOS.

ProDOS file system: The general format of files created and read by applications that run under ProDOS 8 or ProDOS 16 on Apple II computers. Some aspects of the ProDOS file system are similar to the GS/OS **abstract file system.**

ProDOS FST: The part of the GS/OS file system level that implements the ProDOS file system.

ProDOS 16: The first 16-bit operating system developed for the Apple IIGS computer. ProDOS 16 is based on ProDOS 8.

ProDOS 16–compatible calls: Also called *ProDOS 16 calls* or *class 0 calls,* a secondary set of **application-level calls** in GS/OS. They are identical to the ProDOS 16 system calls described in the *Apple IIGS ProDOS 16 Reference.* GS/OS supports these calls so that existing ProDOS 16 applications can run without modification under GS/OS. Compare **standard GS/OS calls.**

quit return flag: A flag, part of the Quit call, that notifies GS/OS whether or not control should eventually return to the program making the Quit call.

quit return stack: A stack of user IDs used to restart applications that have previously quit.

relocation: The process of modifying a file or segment at load time so that it will execute correctly at its current memory location. Relocation consists of patching the proper values onto address operands. The loader relocates load segments when it loads them into memory.

relocation dictionary: The part of every relocatable segment that the loader uses to patch the code for correct execution at its current address.

resource fork: The part of an extended file that contains specifically formatted, generally static data used by an application (such as menus, fonts, and icons).

restartable: A program is restartable if it reinitializes its variables and makes no asumptions about machine state each time it gains control. Only restartable programs can be resurrected from a dormant state in memory.

restart-from-memory flag: A flag, part of the Quit call, that lets the System Loader know whether the quitting program can be restarted from memory if it is executed again.

run-time library files: Special load files that contain general program segments to be loaded as needed by the GS/OS Loaders.

segment: A component of an OMF file, consisting of a header and a body. In object files, each segment incorporates one or more subroutines. In load files, each segment incorporates one or more object segments.

separator: See **pathname separator.**

short prefix: A GS/OS prefix whose maximum total length is 63 characters. Prefix designators `*/` and `0/` through `7/` refer to short prefixes. Compare **long prefix.**

signal: A message from one software subsystem to a second subsystem that something of interest to the second has ocurred.

signal handler: A program that executes in response to a signal. Compare **interrupt handler.**

signal source: A routine that announces the occurrence of a signal when it detects the prerequisite conditions for that signal.

span: The maximum number of characters in a filename; that is, the maximum number of characters between pathname separators, including volume names.

special memory: On an Apple IIGS, all of banks $00 and $01, and all display memory in banks $E0 and $E1.

stack: A list in which entries are added (pushed) and removed (pulled) at the beginning only, causing them to be removed in last-in, first-out (LIFO) order. The term *the stack* usually refers to the particular stack pointed to by the 65C816 stack pointer.

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.

standard file: A named collection of data consisting of a single sequence of bytes. Compare **extended file, directory file.**

standard GS/OS calls: Also called *class 1 calls* or simply *GS/OS calls,* the primary set of **application-level calls** in GS/OS. They provide the full range of GS/OS capabilities accessible to applications. Compare **ProDOS 16–compatible calls.**

static segment: A segment that is loaded only at program boot time and is not unloaded during execution. Compare **dynamic segment.**

system file: Any file of ProDOS file type $FF whose name ends with `.SYSTEM`.

system file level: Determines which files are closed or flushed whenever a Close or Flush call is made with a reference number of 0.

System Loader: The program that loads all other programs into memory and prepares them for execution.

system service calls: Low-level calls in a common format used by internal components of GS/OS (such as FSTs), and also between GS/OS and device drivers.

terminator: A character that, when read, terminates or interrupts a Read call.

terminator list: The list that the console driver users to keep track of terminators.

unclaimed interrupt: An interrupt that is not recognized and acted upon by any interrupt handlers.

volume directory: The name of the highest-level directory in GS/OS.

volume name: The name of the volume directory file on a disk or other medium. All pathnames on a volume start with the volume name. Volume names follow the same rules as other filenames, except that a volume name always starts with a pathname separator.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS computer when running a standard Apple II program). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**

Index

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and Microsoft Word software. Proof pages were created on Apple LaserWriter printers. Final pages were created on the Varityper VT600 imagesetter. Line art was created using Adobe Illustrator™. POSTSCRIPT®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Apple's corporate font, a condensed version of ITC Garamond®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writer: William H. Harris
Developmental Editors: Stella Hackell and Beverly Zegarski
Illustrator: Sandee Karr
Production Supervisor: Rex Wolf

Special thanks to Dave Bice.

> $28.95 FPT USA
> $37.95 CANADA

The Official Publication from Apple Computer, Inc.

Apple IIGS® GS/OS® Reference

Apple IIGS GS/OS Reference is the official programming guide to the standard 16-bit operating system for the Apple IIGS computer. It supersedes the *Apple IIGS ProDOS® 16 Reference.* Written by the people at Apple Computer, Inc., this guide covers important topics of interest to all Apple IIGS application developers, including how to

- access disk files and disk volumes
- use the GS/OS System Loader
- use the Console Driver for character-based screen and keyboard I/O
- handle interrupts in a GS/OS environment

The guide also explains how to use file system translators, code modules that enable GS/OS to work with a variety of file systems, including ProDOS, ISO 9660 (for CD-ROM), and the file system used by AppleShare® file servers.

A chapter in the *Apple IIGS GS/OS Reference* contains summaries of the core 53 commands used to communicate with files, volumes, and character devices. Each summary describes the parameters one must provide to a command, the results returned by the command, and all possible error conditions.

Appendixes describe the Object Module Format (OMF) used by GS/OS for applications and libraries, the Apple extensions to the ISO 9660 standard, and the ProDOS 16 commands that GS/OS supports for compatibility with the original 16-bit operating system Apple developed for the Apple IIGS.

Apple IIGS GS/OS Reference describes the features found in Apple IIGS System Disk 5.0.2. It is an indispensable reference for all programmers of Apple IIGS applications.

Printed in U.S.A.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
408-996-1010
TLX 171 576

Addison-Wesley Publishing Company, Inc.

52895>
9 780201 550207
ISBN 0-201-55020-2
55020