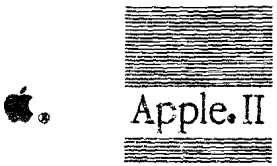


GS/OS™ Reference

Includes System Loader

Volume 1:
Applications and GS/OS

APDA Draft

August 31, 1988

Apple Computer, Inc.

20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Simultaneously published in the United States and Canada.

2/21/88

Contents

Figures and Tables

Preface

The *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, file access to character devices, 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.

In two volumes, the *GS/OS Reference* describes how GS/OS gives applications access to the the full range of Apple IIGS features, and shows how to create device drivers to work with GS/OS.

About this book

The *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 *GS/OS Reference* consists of two volumes plus one disk: the GS/OS Exerciser, a program included on a disk accompanying Volume 1.

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 will 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 Disk*, versions 4.0 and later. Apple IIGS system disks are available from Apple dealers and from APDA (Apple Programmer's and Developer's Association).

Note: System disks earlier than version 4.0 use ProDOS 16 as the operating system. ProDOS 16 is described in the *Apple IIGS ProDOS 16 Reference*.

How to use this book

This book is primarily a reference tool, although parts of each volume are explanatory.

Volume 1 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 to Volume 1 describes GS/OS in general.
- Part I of Volume 1 describes how applications interact with GS/OS, and documents all application-level GS/OS calls.
- Part II of Volume 1 documents the file system translators (FSTs), the software modules that allow your program to access files from many different file systems. For each FST, Part II lists the application calls it supports and documents any differences in call handling from the standard descriptions in Part I.

Volume 2 describes the device interface, the low-level parts of GS/OS that interact with device drivers to control hardware such as disk drives, communication ports, and the console.

- Part I of Volume 2 documents how your program can use GS/OS calls to access a wide variety of devices, both block and character devices, and describes the principal device drivers that are supplied with GS/OS.
- Part II of Volume 2 documents how device drivers interface with GS/OS, and shows you how to write a GS/OS device driver.

The principal descriptions of all application-level GS/OS calls (other than device calls) are in Part I of Volume 1. Call descriptions elsewhere in the book consist mainly of differences from the standard descriptions. The principal descriptions of application-level device calls are in Part I of Volume 2. Driver calls (low-level device calls used by device drivers) are described in Part II of Volume 2.

If you are writing a typical application, the information in Volume 1 is probably all you will need. If you need to access devices directly, or if you are writing a device driver, interrupt handler, message handler, shell, or a large, segmented application, you will need Volume 2 also.

This manual does not explain 65C816 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*.

What it contains

GS/OS is described in two volumes. Here is a brief list of the contents of each chapter and appendix in Volume 1:

Volume 1. The Operating System: What your applications can do with GS/OS.

Introduction. What is GS/OS? An overview of GS/OS.

Part I. The Application Level: The uppermost level of GS/OS.

Chapter 1. Applications and GS/OS: A brief overview.

Chapter 2. GS/OS and Its Environment: How GS/OS affects your program.

Chapter 3. Making GS/OS Calls: The basics of making calls.

Chapter 4. Accessing GS/OS Files: Accessing block files and character files.

Chapter 5. Working with Volumes and Pathnames: Bypassing files; formatting.

Chapter 6. Working with System Information: Communicating with system software.

Chapter 7. GS/OS Call Reference: Documentation of all application-level standard GS/OS calls.

Part II. The File System Level: The middle level of GS/OS.

Chapter 8. File System Translators: How the FST concept works.

Chapter 9. The ProDOS FST: Details about accessing ProDOS files

Chapter 10. The High Sierra FST: Details about accessing files on CD-ROM.

Chapter 11. The Character FST: Details about accessing character devices as files.

Appendixes

Appendix A. GS/OS ProDOS 16 Calls: Making ProDOS 16 calls under GS/OS.

Appendix B. ProDOS 16 Calls and FSTs: How each FST handles ProDOS 16 calls

Appendix C. The GS/OS Exerciser: How to practice GS/OS calls.

Appendix D. GS/OS System Disks and Startup: The major components of a system disk.

Appendix E. Apple Extensions to ISO 9660: Additions to the CD-ROM file format.

Appendix F. GS/OS Error Codes and Constants: A complete listing and description.

Here is a brief list of the general contents of Volume 2:

Volume 2. The Device Interface: How GS/OS provides access to devices.

The Device Level in GS/OS An overview of the lower level of GS/OS.

Part I. Using Device Drivers: How to make calls to GS/OS drivers.

Part II. Writing a Device Driver: How to write a device driver for GS/OS.

Appendixes: Device driver sample code, description of the System Loader.

Other materials you'll need

In order to write Apple IIGS programs that run under GS/OS, you'll need an Apple IIGS computer and development-environment software. Furthermore, you will 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, described in Appendix C of Volume 1, can be useful for practicing GS/OS calls.

Visual cues

Certain 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 the first time it is used. This lets you know that the term has not been defined earlier and that there is an entry for it in the glossary.

Special messages of note 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,* slightly differently than what you are used to. Please note:

Apple II: A general reference to the Apple II family of computers, especially those that may use ProDOS 8 or ProDOS 16 as an operating system. It includes the 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 previous members of the Apple II family share many characteristics, it is useful to distinguish them as a group from the Apple IIGS. A standard Apple II may also be called an 8-bit Apple II, because of the 8-bit registers in its 6502 or 65C02 microprocessor.

ProDOS: A general term describing the family of operating systems developed for Apple II computers. It includes both ProDOS 8 and ProDOS 16; it does not include DOS 3.3 or SOS. ProDOS is also a file system developed to operate with the ProDOS operating systems.

ProDOS 8: The 8-bit ProDOS operating system, through version 1.2, originally developed for standard Apple II computers but compatible with the Apple IIGS. In previous Apple II documentation, ProDOS 8 is called simply ProDOS.

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

GS/OS: A native-code, 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 monospace 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
```

This includes assembly language labels, entry points, and file names that appear in text passages. GS/OS call names and the names of other system software functions, however, are printed in normal font in uppercase and lowercase letters (for example, GetEntry and LoadSegmentNum). 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_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 descripions of the manuals follow.

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.

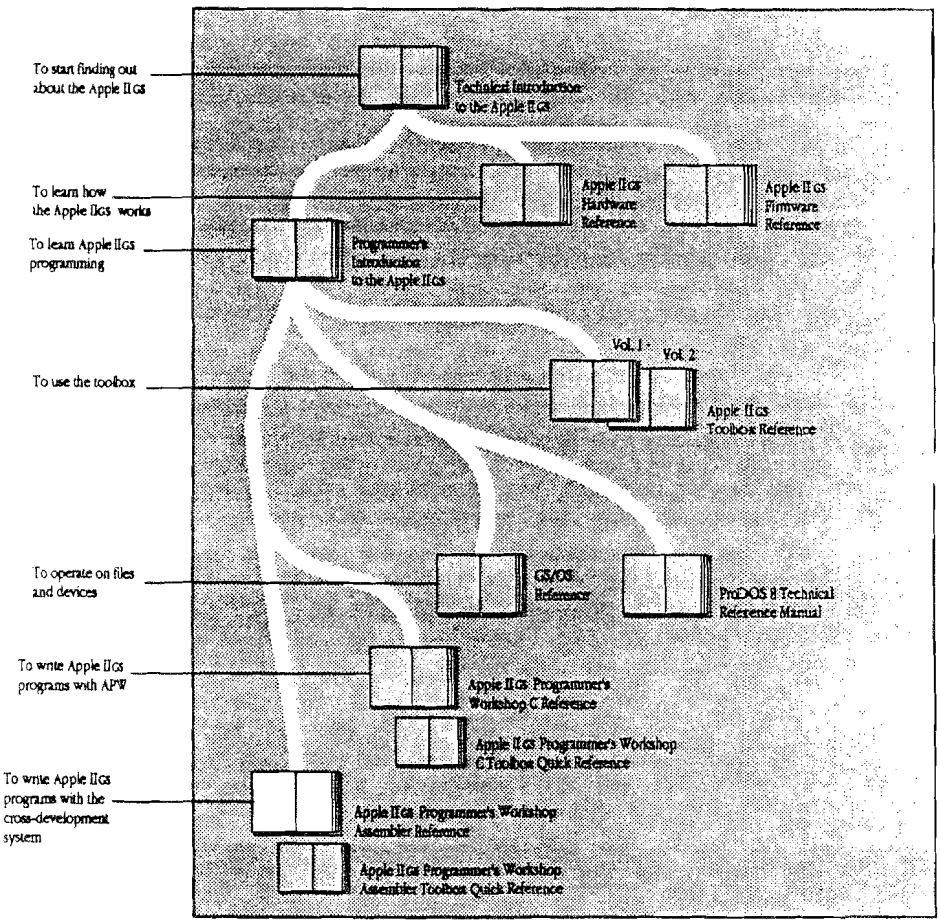
- **The Technical Introduction:** The *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals about the Apple IIGS. It describes all aspects of the Apple IIGS, including its features and general design, the program environments, the toolbox, and the development 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.

- **The Programmer's Introduction:** 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.

Figure P-1. Roadmap to Apple IIGS technical manuals



- **The firmware reference:** 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 I/O subroutines for the serial ports, the disk port, and for 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. The *Apple IIGS Toolbox Reference*, Volume 1, introduces concepts and terminology and tells how to use some of the tools. The Apple IIGS Toolbox Reference, Volume 2, contains information about the rest of the tools. Volume 2 also tells how to write and install your own tool set.

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 manual indispensable.

Apple IIGS operating-system manuals

The Apple IIGS two preferred 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. The *GS/OS Reference* describes GS/OS and includes information about the System Loader, which works closely with GS/OS to load programs into memory.

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 8-bit Apple II computers as well as on the Apple IIGS. ProDOS 8 is described in the *ProDOS 8 Technical Reference Manual*.

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*.

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* 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 accurate 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. See Figure P–1. 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 Programmer's Workshop manual:** 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.
- **Assembler:** 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.
- **C compiler:** The *Apple IIGS Programmer's Workshop C Reference* includes the specifications of the APW C implementaion 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 are available through the Apple Programmer's and Developer's Association (APDA).

The MPW IIGS manuals

Macintosh Programmer's Workshop (MPW) is one of the two development environments 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*. That book is the only Macintosh manual you need when writing programs using MPW IIGS.

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 appplications under MPW. It describes the linker, 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 the Apple Programmer's and Developer's Association (APDA).

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.

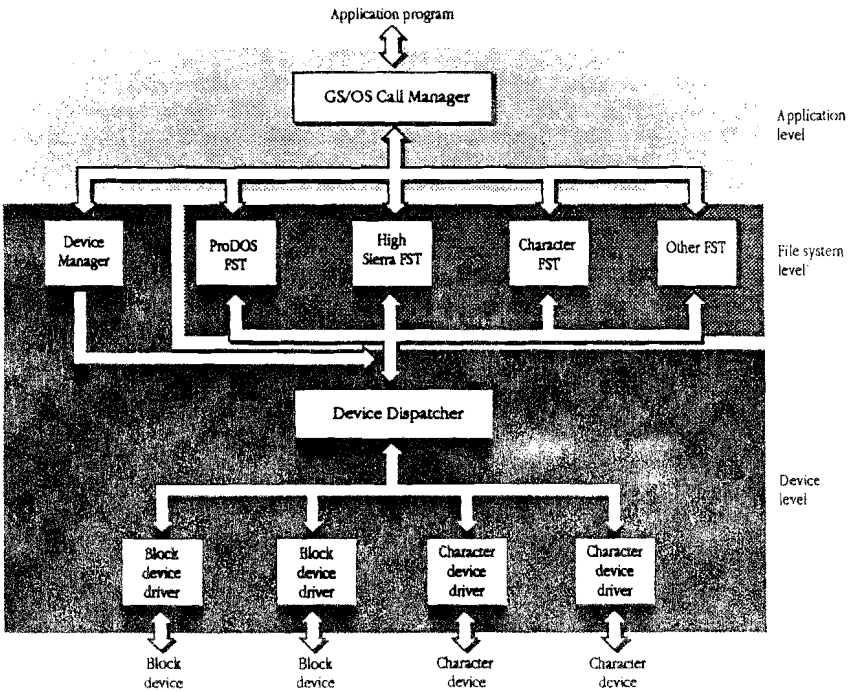
Even more important, GS/OS is file-system independent: by making GS/OS calls, your application can read and write files transparently among many different and normally incompatible file systems. GS/OS accomplishes this by defining a generic GS/OS file interface, the abstract file system. Your application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

This chapter 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 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 1-1 Interface levels in GS/OS



- **Application level:** Applications interact with GS/OS mostly at the application level. The application level 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 application level mediates between an individual application and the file system level. The application level is described in Part I of this volume.

- **File system level:** The file system level consists of **file system translators** (FSTs), which take 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.

  Note that 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 level mediates between the application level and the device level. The file system level is described in Part II of this volume.

- **Device level:** The device level communicates with all device drivers connected to the system. In handling a typical GS/OS call, the device level mediates between the file system level and an individual device driver.

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

  Device calls are described in Part I of Volume II; if your application needs direct access to devices, look there to find out how to do it. Driver calls are described in Part II of Volume II; 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 all other 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 when loading programs. The System Loader and its calls are documented in Volume 2. For most applications, however, its functioning is totally automatic; only specialized programs such as shells need 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 very easily added 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 the Macintosh uses). 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 volume. FSTs are described in Part II of this volume.

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 volume. Device drivers are described in Part II of Volume 2.

Speed enhancements

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

Disk caching is described in Volume 2.

Eliminated 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 2^{32} bytes (rather than only 16 MB for files and 32 MB for volumes).

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

ProDOS 16 compatibility

GS/OS includes a complete set of ProDOS 16 calls and implements them just as ProDOS 16 does. All well-behaved ProDOS 16 applications can run without modification under GS/OS. An added benefit is that 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, there are several categories of calls that programs can make 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 Volume 1; most low-level calls are described in Volume 2. Within these broad divisions, there are several subcategories of calls and call-related descriptions; each subcategory is described in a different place in the two volumes. The categories are as follows:

In Volume 1:

- **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.
- **FST-specific information on GS/OS calls:** Because different file systems have different characteristics, 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, 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.

In Volume 2:

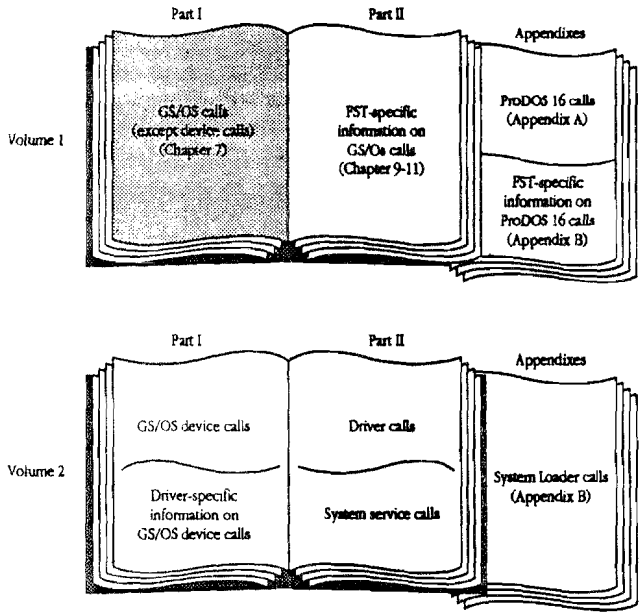
- **GS/OS device calls:** These are a subset of the application-level, standard GS/OS calls described in Volume 1, but they are special because they bypass the file level altogether and access devices directly.
- **Driver-specific information on GS/OS device calls:** Because different devices have different characteristics, not all device drivers 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.
- **System Loader calls:** These are calls a program can make to load other programs or program segments into memory. Although the typical application makes no System Loader calls, they are described in this book so that shells and system-level programs can make use of them.

Figure 1-2 shows you where to look in each volume 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 Part I of Volume 1 (Chapter 7); the descriptions of driver calls are in Part II of Volume 2 (Chapter 9).

Note: Figure 1-2 is reproduced in each Part opening in this book, highlighted in each case to show the calls described in that part.

Figure 1-2 Where to find call descriptions in this book.

Most applications make only the calls described in Part I of Volume 1 (shaded area).



GS/OS system requirements

GS/OS will not run on a standard Apple II computer. It requires an Apple IIGS with a ROM revision of 1.0 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.

Background to the development of GS/OS

To summarize this overview of GS/OS, this chapter ends with a brief discussion of how GS/OS evolved from previous Apple II operating systems.

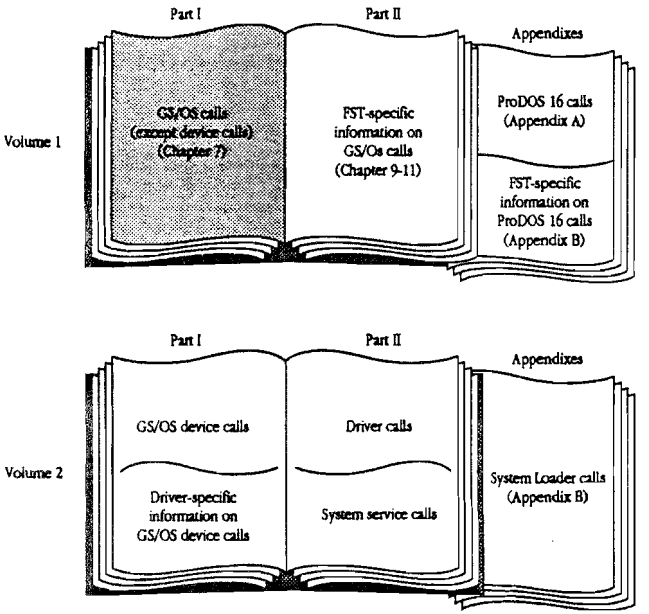
Apple has created several operating systems for the Apple II family of computers. GS/OS is the latest in that line; it is related to several earlier systems, but has far greater capabilities than any of them. Here are thumbnail sketches of the other systems:

- **DOS:** DOS (for *Disk Operating System*) was Apple's first operating system. It provided the Apple 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 an Apple 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'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 (for standard Apple II computers) is that it requires at least 256 Kb 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 64K of memory space. ProDOS supports exactly the same hierarchical file system as SOS, but 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—although there are a few important additions, it has essentially the same features as ProDOS 8 and supports exactly the same hierarchical file system. ProDOS 16's main advantage 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 ProDOS 16's I/O performance and eliminates its restrictions on number and size of open files, volumes, and devices. GS/OS supports character devices as files, it handles devices uniformly, and it 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 lineage, 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



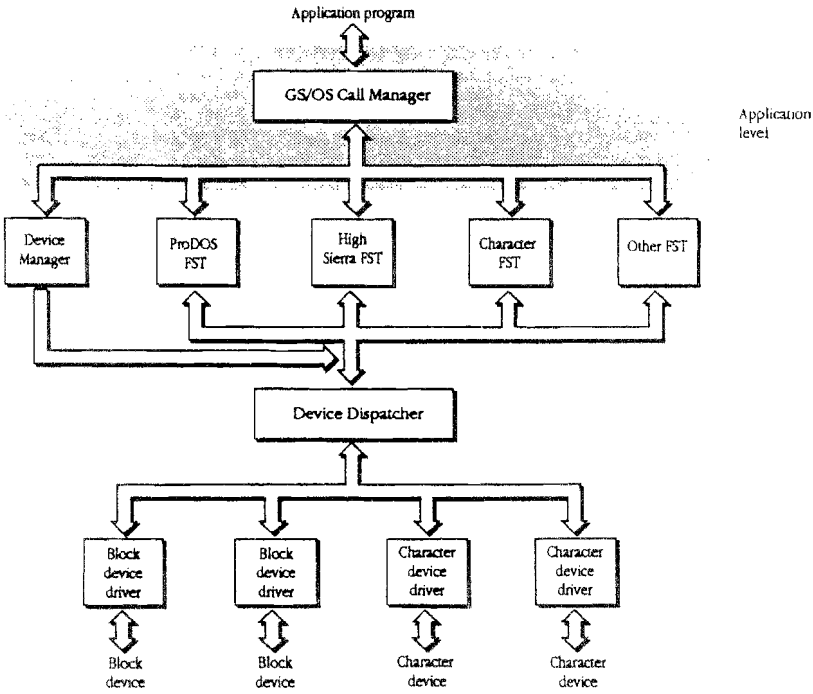
Chapter 1 The GS/OS Abstract File System

Two key features of GS/OS are its ability to insulate applications from the details of (1) the hardware devices connected to the system, and (2) the file systems used to store applications and their data. This chapter shows how GS/OS implements these features. It also lists, by category, the GS/OS calls that an application can make.

A high-level interface

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

Figure 1-1 Application level in GS/OS



Creation and modification date and time

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 subdirectories—including the volume directory—must be updated.

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

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. And, unlike with block files, 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 11, "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 and pathname calls (described in Chapter 5)
- System information calls (described in Chapter 6)

In addition to these groups of calls, the Quit call is used when an application quits, and is described in Chapter 2.

Finally, GS/OS provides calls that directly access devices and install interrupt and signal handlers. For more detail on those calls, refer to Volume 2. Table 1-3 lists the groups of GS/OS calls.

Table 1-3 GS/OS call groups

File access calls	Volume and pathname calls	System information calls	Device calls
Create ($2001)	ChangePath ($2004)	SetSysPrefs ($200C)	DControl ($202E)
Destroy ($2002)	Volume ($2008)	GetSysPrefs ($200F)	DInfo ($202C)
SetFileInfo ($2005)	SetPrefix ($2009)	GetName ($2027)	DRead ($202F)
GetFileInfo ($2006)	GetPrefix ($200A)	GetVersion ($202A)	DStatus ($202D)
GetFileInfo ($2006)	ExpandPath ($200E)	GetFSTInfo ($202B)	DWrite ($2030)
ClearBackup ($200B)	Format ($2024)		
Open ($2010)	EraseDisk ($2025)		
Newline ($2011)	GetBootVol ($2028)		
Read ($2012)			
Write ($2013)			
Close ($2014)			
Flush ($2015)			
SetMark ($2016)			
GetMark ($2017)			
SetEof ($2018)			
GetEof ($2019)			
SetLevel ($201A)			
GetLevel ($201B)			
GetDirEntry ($201C)			
BeginSession ($201D)			
EndSession ($201E)			
SessionStatus ($201F)			
ResetCache ($2026)			

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 of this volume.

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 with write access precludes any other opens 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 the Mark by using the SetMark and GetMark calls, and set or get the end of the file by using the SetEOF and GetEOF
- Enable or disable newline mode by using the Newline call
- 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, GetFileLevel, and SetFileLevel 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, ClearBackup, 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 5..

Two other 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 the calls allow you to change the creation and modification dates for a file.

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

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

- physically 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 and GetPrefix calls
- 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
- 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

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.

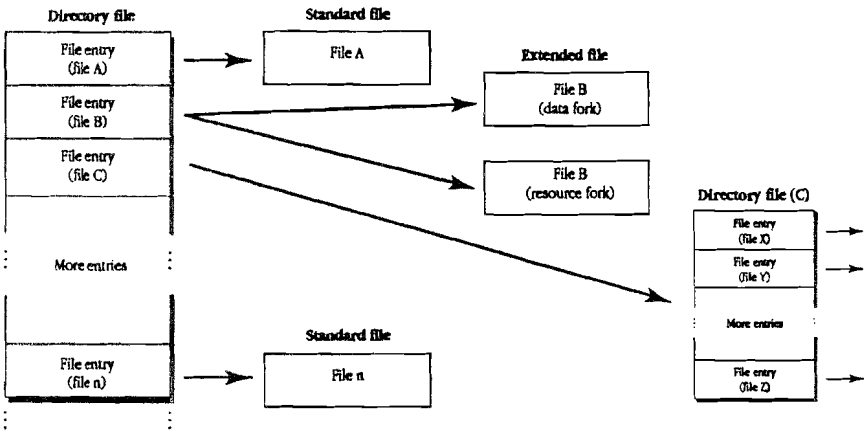
Device calls

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 (that 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

A brief summary of the individual calls is listed alphabetically by name in Chapter 7, and information device calls are completely described in Volume 2.

Figure 1-3 Directory file format



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

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.

You usually only need to examine directory files when you are creating catalog-type applications; 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.

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 GS/OS makes no assumptions about their internal formats; 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 as part of the filename. 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 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 (pathname syntax error). FSTs are also responsible for indicating whether filenames should be case-sensitive or not, and whether the high-order bit of each character is turned off. See Part II of this volume for more information about FSTs.

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

```
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 file names starting with the name of the volume directory name and ending with the name of the file. These 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.

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 colon or slash in the input string determines the separator. When the colon is the separator, the constituent filenames must not contain colons, but can contain slashes. When the slash is the separator, the constituent filenames must not contain slashes or colons. Thus, colons are never allowed in filenames.

Examples of legal full pathnames are as follows:

```
/aloysius/beelzebub/cat
:a:b:c
/x
:x
.d1/a/b
```

Examples of illegal full pathnames are as follows:

`/:::/::/:`	a ":" must not appear in a filename
`/a/b/c`	assuming that the first filename is supposed to be "a/b"

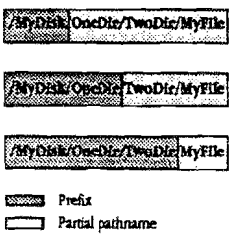
`/a/b/c/`	cannot have a separator after the last filename
`a/b/c/`	must start with a volume or device name

All calls that require you to name a file will accept either a full pathname or a partial pathname.

Prefixes and partial pathnames

A full pathname can be broken down into a prefix and a partial pathname. In essence, the prefix starts at the beginning of the pathname (that is, at the volume or device name) and can continue down through the last directory name in the path. In contrast, the partial pathname starts at the end of the pathname and can continue up to, but not 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 prefix and partial pathname portions of a single full pathname.

Figure 1-4 Prefixes and partial pathnames



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

- 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 specifies the volume from which GS/OS was last booted.
- Nothing. This is identical to prefix 0 (that is, equal to "0:" or "00000/").

Table 1-1 shows some examples of prefix use. They assume that 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 Examples of prefix use

- Full pathname provided:

  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

- Explicit use of prefix /0:

  as supplied: 0/SYSTEM/FINDER
  as expanded by GS/OS: /VOLUME1/SYSTEM/FINDER

- Use of prefix 5/:

  as supplied: 5/CHAP.12
  as expanded by GS/OS: /VOLUME1/TEXT.FILES/CHAP.12

File information

GS/OS files are marked as having several characteristics, including those that follow:

- Access permissions to the file
- File type and auxiliary type of the file
- The size of the file and the current reading-writing position in the file
- Creation and modification date and time

Your application can access and modify this information, as 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, but do indicate to GS/OS and other 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.

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

Table 1-2 shows the valid table types. In Table 1-2, the descriptions under the Auxiliary type column have the following meanings:

- *Application specific* means that the auxiliary type specifies which application created the file
- *Way the xxxx is stored* means the auxiliary type differentiates between various storage methods
- *Upper/lower case in filename* means that AppleWorks uses 15 bits of the auxiliary type word (it's a word on disk, instead of a long word, for the ProDOS file system) to flag whether to display that letter of the filename in lowercase
- *Not loaded if bit 15 is set* means that GS/OS won't load or execute files like DAs and Setup files if bit 15 of the auxiliary type is set
- *APW language type* is the language designation for APW source files
- *Load address in bank for BASIC.SYSTEM* is the default load address for ProDOS 8 executable binary files (file type $06)

- *Random-access record length* specifies the record length for an ASCII text file (file type $04)

Table 1-2 GS/OS file types and auxiliary types

File type	Description	Auxiliary type
$00	Uncategorized file	
$01	Bad blocks file	
$04	ASCII text file	Random-access record-length (0=Sequential file)
$06	Binary file	Load address in bank for BASIC.SYSTEM
$08	Double Hi-Res file	
$0F	Directory file	
$19	AppleWorks database file	Upper/lower case in file name
$1A	AppleWorks word processor file	Upper/lower case in file name
$1B	AppleWorks spreadsheet file	Upper/lower case in file name
$50	Word processor file	Application specific
$51	Spreadsheet file	Application specific
$52	Database file	Application specific
$53	Object-oriented graphics file	Application specific
$54	Desktop publishing file	Application specific
$55	Hypermedia file	Application specific
$56	Educational data file	Application specific
$57	Stationery file	Application specific
$58	Help file	Application specific
$59	Communications file	Application specific
$5A	Application configuration file	Application specific
$AB	GS BASIC program file	
$AC	GS BASIC Toolbox definition file	
$AD	GS BASIC data file	
$B0	APW source file	APW Language type
$B1	APW object file	
$B2	APW library file	
$B3	GS/OS application	
$B4	GS/OS Run-time library file	
$B5	GS/OS Shell application file	
$B6	GS/OS permanent initialization file	Not loaded if high bit set
$B7	Apple IIGS temporary initialization file	Not loaded if high bit set
$B8	New Desk Accessory	Not loaded if high bit set
$B9	Classic Desk Accessory	Not loaded if high bit set
$BA	Tool file	
$BB	Apple IIGS device driver file	Not loaded if bit 15 set
$BC	Generic load file	

$BD	GS/OS file system translator	Not loaded if bit 15 set
$BF	Apple IIGS sound file	
$C0	Apple IIGS Super Hi-Res screen image	Way the image is stored
$C1	Apple IIGS Super Hi-Res picture file	Way the picture is stored
$C8	Apple IIGS font file	
$C9	Apple IIGS Finder data file	
$CA	Apple IIGS Finder icon file	
$D5	Music sequence file	Application-specific
$D6	Instrument file	Application-specific
$D7	MIDI file	
$E0	Telecommunications Library file	Application-specific
$E2	AppleTalk File	
$EF	Pascal area on partitioned disk	
$F0	BASIC.SYSTEM Command File	
$F1	User-defined file type #1	
$F2	User-defined file type #2	
$F3	User-defined file type #3	
$F4	User-defined file type #4	
$F5	User-defined file type #5	
$F6	User-defined file type #6	
$F7	User-defined file type #7	
$F8	User-defined file type #8	
$F9	GS/OS System file	
$FA	Integer BASIC program file	
$FB	Integer BASIC variable file	
$FC	AppleSoft BASIC program file	
$FD	AppleSoft BASIC variable file	
$FE	EDASM relocatable code file	
$FF	ProDOS 8 application	

EOF and mark

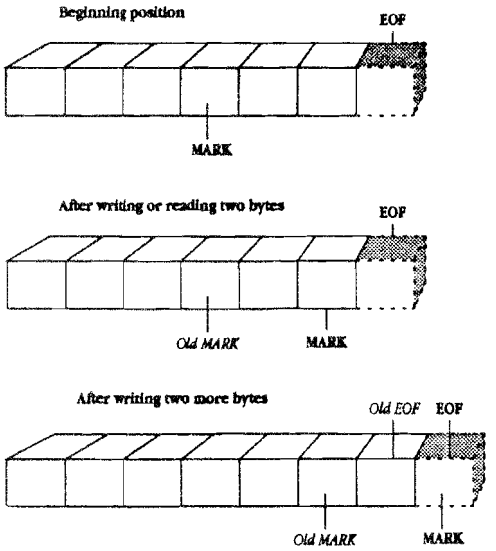
To aid reading from and writing to files, each open standard file and each fork of an open extended file has a byte count indicating the size of the file in bytes (EOF), and another defining the current position in the file (the mark). GS/OS moves both EOF and mark automatically when data is added to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

EOF is the number of readable bytes in the file. Since the first byte in a file has number 0, 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 to write new data. It cannot exceed EOF.

If the mark meets 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 EOF to accommodate the new information. Figure 1-5 illustrates the relationship between the mark and EOF.

Figure 1-5 Automatic movement of EOF and mark



An application can place 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 EOF. These two functions can be accomplished using the SetEOF and Setmark calls. The current values of EOF and the mark can be determined using the GetEOF and Getmark calls.

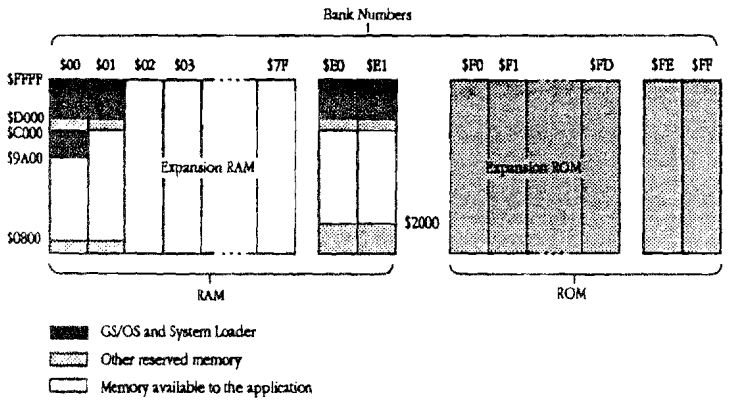
Chapter 2 GS/OS and Its Environment

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



GS/OS and the System Loader together occupy nearly all addresses from $D000 through $FFFF in banks $00, $01, $E0, and $E1. In addition, GS/OS reserves (through the Memory Manager) approximately 9.5 KB just below $C000 in bank $00 for GS/OS system code and data. None of these reserved memory areas is available for use by applications.

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.

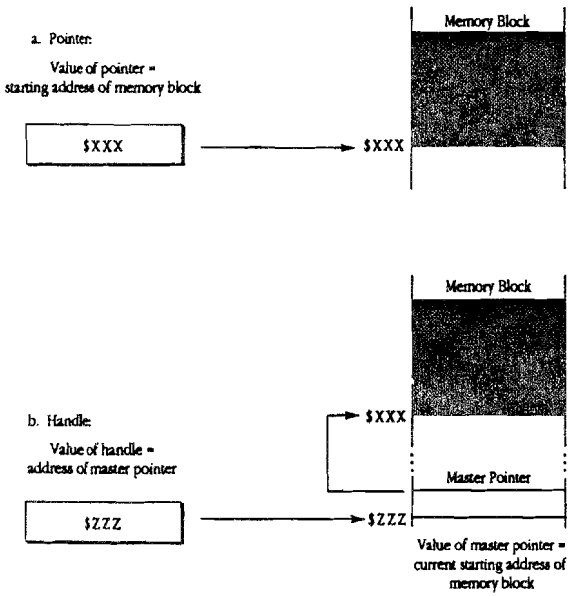
Other reserved memory includes the ROM space in banks $FE and $FF; they contain firmware and ROM-based tools. In addition, banks $F0 through $FD are reserved for future ROM expansion.

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

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 $C000 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

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" 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

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, it does not call the Memory Manager to allocate a direct-page/stack space, and it returns zeros as the base address and size of the space. 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.

System startup considerations

The startup sequence for the Apple IIGS is is invisible to applications and relatively complex, so further discussion of the sequence is presented in Appendix D, "GS/OS System Disks and Startup." That appendix describes the structure of a valid system disk.

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. The GS/OS program dispatcher 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 indicate to GS/OS whether your application can be restarted from memory. You can also specify the next application to be launched, and whether your application should be placed on the quit return stack so that it will be restarted when the other program quits. 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 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 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 would pass automatically back to the shell; if the shell then quits without specifying a pathname, control would pass 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 and not 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:

- 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:

- 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 2-2.

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

Item	State
Reserved memory	Addresses above $9A00 in bank zero 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. The only other space that GS/OS requires in RAM is the language-card areas in banks $00, $01, $E0, and $E1.
Hardware registers	
accumulator	Contains the user ID assigned to the application.

X- and Y-registers	Contain zero ($0000).
e-, m-, and x-flags in the processor status register	All set to zero; processor in full native mode.
stack register	Contains the address of the top of the direct-page/stack space.
direct register	Contains the 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.

At all times during execution, GetName returns the filename of the current application (regardless of whether prefix 1/ has been changed), and GetBootVol returns the boot volume name, equal to the value of prefix */ (regardless of whether prefix 0/ has been changed).

Table 2-4 Prefix values when GS/OS application launched at boot time

Prefix	Description
*	boot volume name
0	boot volume name
1	full pathname of directory containing current application
2	*/SYSTEM/LIBS
3-8	null strings
9	equal to prefix 1
10-31	null strings

Table 2-5 Prefix values—GS/OS application launched after GS/OS application quits

Prefix	Description
*	unchanged from previous application
0	unchanged from previous application
1	full pathname of directory containing current application
2	unchanged from previous application
3-8	unchanged from previous application
9	equal to prefix 1
10-31	unchanged from previous application

Table 2-6 Prefix values—GS/OS application launched after ProDOS 8 application quits

Prefix	Description
*	boot volume name
0	unchanged from the ProDOS 8 system prefix under previous application
1	full pathname of the directory containing the current application
2	*/SYSTEM/LIBS
3-8	null strings
9	equal to prefix 1
10-31	null strings

Pathname prefixes at ProDOS 8 application launch

Table 2-7 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 2-7 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.

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:

- Macro technique—uses macros defined by Apple to generate inline calls. Macro calls are the simplest and the easiest to read.
- Inline call technique—similar to ProDOS 8
- Stack call technique—consistent with the way compilers generate code

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

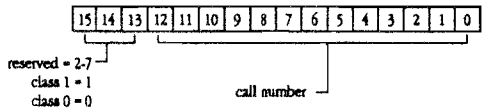
To make a GS/OS assembly language call, your application must provide

- a 2-byte call number or the macro name of the call
- If you don't use the macro name, a Jump to Subroutine Long (JSL) instruction to the appropriate GS/OS entry point
- a 4-byte pointer to the 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 document, 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 would use the macro name or a `callnum` value of $2010; to make a ProDOS 16-compatible (class 0) OPEN call, the caller would use a `callnum` value of $0010.

Making a GS/OS call using macros

To make a standard GS/OS call using the macro technique, perform the following 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 inline GS/OS call

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

1. Perform a JSL to $E100A8, the GS/OS inline 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 inline call:

```
inline_entry   gequ   $E100A8          ;address of GS/OS inline entry point
;
               jsl    inline_entry     ;long jump to GS/OS inline 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.SYSCALLS and M16.SYSCALLS	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:

SYSCALLS.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 or 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, code, or buffer space in which GS/OS can receive or place data; that is, the pointer may point to a location that contains an input, or point to space that will receive a result, or point to 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 will accommodate large devices using file system translators.

All parameter fields contain an even number of bytes, for ease of manipulation by the16-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.

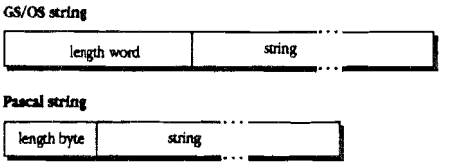
Pointers in the parameter block must be written with the low-order byte of the low-order word at the lowest address.

Important The range of theoretically possible values as 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 the permissible values for a field depends upon its file system.

GS/OS string format

GS/OS strings resemble Pascal-style strings. A **Pascal-style 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 instead of a byte. See Figure 3-1.

Figure 3-1 GS/OS and Pascal strings



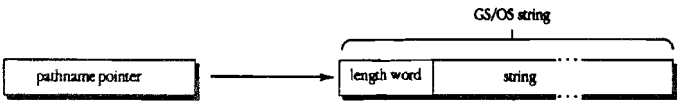
String parameters consist of a pointer parameter in the call's parameter block that points to a data structure containing the string. For standard GS/OS calls, that data structure varies depending on whether the string parameter is an input to or output from the call.

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

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.

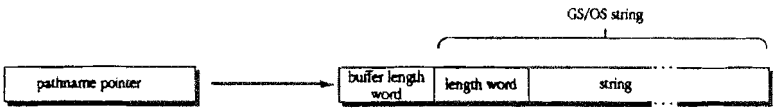
Figure 3-2 GS/OS input string structure



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 length word that specifies the total length of the buffer, including the buffer length 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 length word) to hold the output string. If there is enough space, the result is placed in the buffer starting just after the buffer length word.

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

If the area is too small to contain the string, GS/OS returns a "buffer too small" error and sets the string length field to the actual string length. In this case, the string field is undefined. The caller must add four to the returned string length to determine the total area size needed to hold the string and the two length fields.

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 such a parameter block will be.

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

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

2. Use Memory Manager and System Loader calls to place the block in memory, as follows:
   a. 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.
   b. Obtain a pointer to the block, by dereferencing the memory handle returned by the Memory Manager (that is, read the contents of the location pointed to by the handle, and use that value as a pointer to the block).
   c. Set up your parameter block, starting at the address pointed to by the pointer obtained in step (b).

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 exit from GS/OS

Register	Description
A	zero 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 exit 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

Note: The n flag is undefined here; under ProDOS 8, it is set according to the value in the accumulator.

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. If the error is unrecoverable and requires that the system be rebooted, the GS/OS Error Manager calls the System Failure Manager, a part of the Apple IIGS Toolbox. The System Failure Manager is described in the chapter "Miscellaneous Tool Set" in the *Apple IIGS Toolbox Reference* .

The errors that specifically apply to a particular call are listed as part of the call description in Chapter 7. Other errors can occur for almost any of the calls. For example, almost any call can return error $54 (out of memory), and perhaps you would want to invoke a special error handler for that condition.

Chapter 4 Accessing GS/OS Files

The most common use of GS/OS is to access files that contain data on a storage medium. 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, as an 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 SFPutFile or SFGetFile (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 automatically construct the full pathname to the file.

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

The simplest access method

To use this method, perform the following steps:

1. If you are creating a new file, call the tool set routine SFPutFile 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.

   If the user is opening an existing file, call the tool set routine SFGetFile to get a pointer to the pathname of the file that the user wishes to open. Save the pointer, and use it in a GS/OS Open call to open the file.
2. If the user is opening an existing file, call the tool set routine SFGetFile to get a pointer to the pathname of the file the user wishes to open. Save the pointer, and use it in a GS/OS Open call to open the file.
3. While the file is open, 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 end-of-file (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.
4. 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 SFPutFile, you only have to save the pathname pointer it returns and supply that pointer to GS/OS. If you want to build the pathname yourself, see Chapter 5.
- The file access, which determines whether or not the file can be written to, read from, destroyed, or renamed, and whether the file is invisible.
- A file type and auxiliary type, which indicate to other applications the type of information to be stored in the file. It does not affect, in any way, the contents of the file.
- A storage type, which determines the physical format of the file on the disk. There are three different formats: one is used for directory files, the other two for nondirectory files. Once a file has been created, you can't change its storage type.
- The size of the file and the size of the resource of the file, which are used to preallocate disk storage for the file to be created. 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**). A created 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 GS/OS returns an error. As already mentioned, you can query the user for the filename by using the SFGetFile 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 open files at a time limited only by the amount of total available memory and number of available reference numbers. In practice, there is no limit to the number of open files.a practical limit, . 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.

Your application can also further limit the read-write access to a file when it makes a GS/OS Open call; for example, if the file was created with read-write access, you could change that access to read-only.

You should 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.

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:

- reference number of the file (assigned when the file was opened)
- location in memory of a buffer that contains, or is to contain, the transferred data
- number of bytes to be transferred
- 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, 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.

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 called GetDirEntry; for more details, see GetDirEntry in Chapter 7.

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 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**. 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 command:

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 backup the file.

If you want to change the state information about the backup, and in effect indicate to GS/OS that the file does not need to be backed up, you can use the ClearBackup call. This resets the backup status so that it looks to GS/OS as if the file had not been altered. For example, you could use this technique in a word-processing application if the user deleted something from the file but then decided to undo the change; issuing the ClearBackup call would prevent 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 11, "Character FST," for a detailed discussion of that kind of file.

Setting or getting file characteristics

Certain characteristics about 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.

Those 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 manual for more information about FSTs)

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

Changing the creation and modification date and time

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 4-1.

Table 4-1 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

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

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

Finally, when a file is copied, a utility program must be sure to give the copy the same creation and modification date and time as the original file, and not the date and time at which the copy was created. See the section "Copying Files" in this chapter for more information.

Copying files

GS/OS provides several techniques that help your application copy files. This section details those techniques.

Copying single files

To copy single files, 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 the source and destination files. Use Read and Write calls to copy the source to the destination. Close both files.
4. 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.

Copying multiple files

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.
2. Copy the files .
3. 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.

If your application gets error $54 (out of memory) when sessions are active, it should make an EndSession call, make a BeginSession call, and try the operation again. If the operation still fails, more EndSession and BeginSession calls will not help.

Chapter 5 Working with Volumes and Pathnames

If you don't want to, you can usually avoid working with volumes, pathnames, and devices in detail; GS/OS can free you from keeping track of exactly where files exist. As discussed in Chapter 4, if you use the Apple IIGS Standard File Operations Tool Set routines SFPutFile and SFGetFile, 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 SFPutFile and SFGetFile. 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 SFPutFile and SFGetFile).
- It needs to check whether particular files are in the appropriate directories; for example, if 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 you need to understand about those 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 Volume 2, "The Device Interface."

Working with volumes

Some GS/OS calls are designed to allow you to work directly with volumes, as 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:

- name of the volume
- total number of blocks on the volume
- number of free blocks on the volume
- file system contained on the volume
- 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 following 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 $53 (parameter out of range). DInfo returns the name of the device associated with that 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 directroy 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 could start up QuickDraw II and the Event Manager and then use the GetBootVol call to check if the boot volume is online. This would allow the application to put up a custom dialog box if the boot volume was offline.

Formatting a volume

GS/OS provides two format options to applications, as follows:

- The GS/OS Format call attempts to physically format the disk; this method is necessary when your application can't read the existing volume.
- The GS/OS EraseDisk call assumes that a physically formatted medium already exists in the appropriate device, and writes new boot blocks, directory, and bitmaps to the disk. EraseDisk is usually faster than Format, but requires that the disk already be physically 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.

In both of these cases, you have to provide a device name to the call, so you'll 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 graphics 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 graphics user interface, your application has to meet the following requirements:

- The IIGS Toolbox Desk Manager must be active; by implication, all of the tools sets upon which the Desk Manager depends must also be active (see the *Apple IIGS Toolbox Reference*).
- In addition, the List Manager must be active.
- For the graphics tools to run, 64 KB of free RAM must be available.
- The super hi-res screen must be currently displayed.

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

Working with pathnames

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

Setting and getting prefixes

You can use standard GS/OS calls to manually set and retrieve the prefix assignments. 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. To retrieve the name of the boot volume prefix, use the GS/OS GetBootVol call, as 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 will automatically adjust all pathnames to reflect the changed prefix.

Changing the path to a file

GS/OS allows you to change the path to a specified file. From the user's viewpoint of a file system, 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 d

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

Expanding a pathname

GS/OS allows you to expand a partial pathname into its corresponding full pathname.

To expand the pathname, use the standard GS/OS call ExpandPath. For detailed information about how to expand the path, see ExpandPath 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, 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). The DInfo call returns the name of the device associated with that device number (see Chapter 7 for details on DInfo).

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.
3. Open that volume by using the GS/OS Open call.
4. Get the directory entries for the files by using successive GetDirEntry calls.

Introducing devices

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

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

- 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 directly access blocks. The calls that directly access devices are briefly summarized in Chapter 7, and discussed in detail in Chapter 2 of Volume 2.

Note: RAM disks are software constructs that the operating system treats like devices. GS/OS supports any RAM disk that behaves like a block device in all respects just as if it were a block device.

Character devices

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

GS/OS supports character devices through both direct and file access. For more information, see Chapter 11 in this volume.

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 Volume 2, "The Device Interface," 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 Volume 2 of this manual.

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 preference word that allows your application to customize some GS/OS functions. One of the options provided is the ability of the application using pathname callsto determine whether or not it wants to handle error $45 (volume not found) itself, or whether it wants to have GS/OS handle those errors.

For information on how to set up the preferences word, and on any other options available in that word, see the description of SetSysPrefs and GetSysPrefs in Chapter 7.

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, 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 more detailed information about how to retrieve the information, see GetFSTInfo in Chapter 7. For more information about FSTs, see Part II of this volume.

Finding out the version of the operating system

If your application depends upon some 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 information, see the GetVersion call in Chapter 7.

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 an application wanted to display its own name to the user, it could use GetName to get its current name (remember, the user can rename applications).

For more detailed information about how to retrieve the information, see the GetName call in Chapter 7.

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.

$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 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: The number of parameters in this parameter block. Minimum is 0; maximum is 0.

Errors (none)

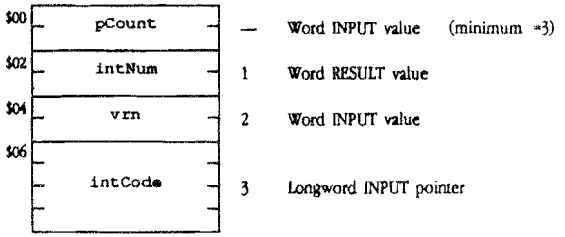
$2031 BindInt

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 Volume 2. See also the UnbindInt call in this chapter.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 3; maximum is 3.

`intNum` Word result value: An identifying number assigned by GS/OS to the 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.

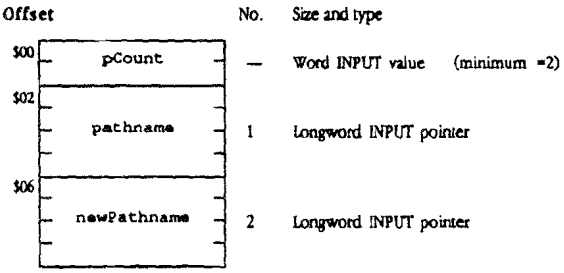
Errors

$25 interrupt vector table full
$53 parameter out of range

$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.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 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

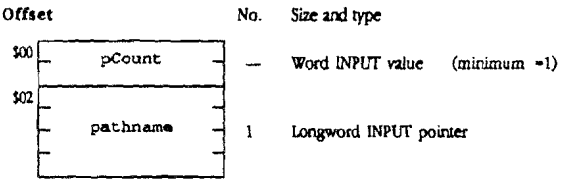
$10	device not found
$27	I/O error
$2B	write-protected disk
$40	invalid pathname syntax
$44	path not found
$45	volume not found
$46	file not found
$47	duplicate pathname
$4A	version error
$4B	unsupported storage type
$4E	access: file not destroy enabled
$50	file open
$52	unsupported volume type
$53	invalid parameter
$57	duplicate volume
$58	not a block device
$5A	block number out of range

$200B ClearBackup

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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 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	I/O error
$28	no device connected
$2B	write-protected disk
$2E	disk switched
$40	invalid pathname syntax
$44	path not found
$45	volume not found
$46	file not found
$4A	version error
$52	unsupported volume type
$58	not a block device

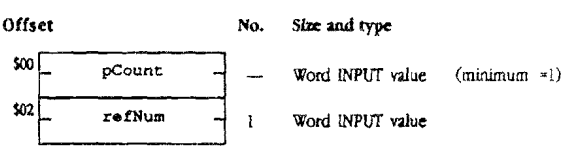
$2014 Close

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.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`refNum` 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	I/O error
$2B	write-protected disk
$2E	disk switched
$43	invalid reference number
$48	volume full
$5A	block number out of range

$2001 Create

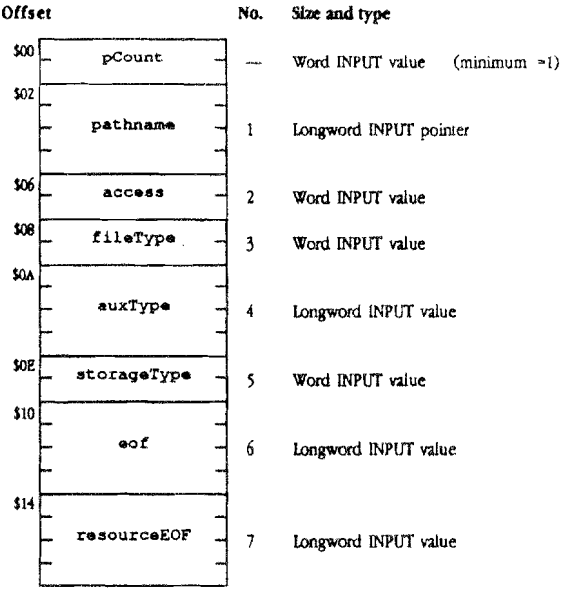
Description

This call creates either a standard file, an extended file, or a subdirectory on a volume mounted in a block device. A standard file 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; the character FST creates that special kind of file (see Chapter 11).

This call sets up file system state information for the new file and initializes the file to the empty state.

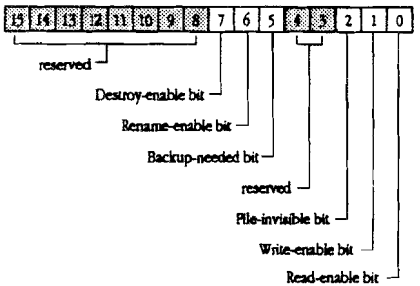
Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 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 and listed in Table 1-2 in Chapter 1 of this volume.

`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 by Apple Computer and listed in Table 1-2 in Chapter 1 of this volume.

`storageType` Word input value: The value of this parameter determines whether the file being created is a standard file, an extended file, or subdirectory file. The following values are valid:

$0000–$0003*	create a standard file
$0005	create an extended file
$000D	create a subdirectory file

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

`eof` Longword input value: The `eof` parameter 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: For an extended file, this parameter specifies the amount of space to preallocate for the resource fork. 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.

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.

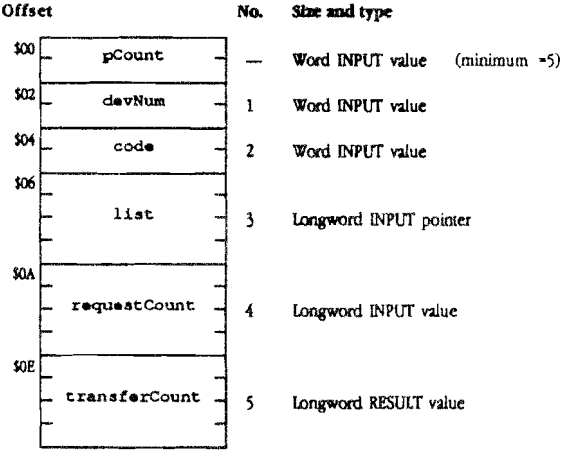
Errors

$10	device not found
$27	I/O error
$2B	write-protected disk
$40	invalid pathname syntax
$44	path not found
$45	volume not found
$46	file not found
$47	duplicate pathname
$48	volume full
$49	volume directory full
$4B	unsupported storage type
$52	unsupported volume type
$53	invalid parameter
$58	not a block device
$5A	block number out of range

$202E DControl

Description This call sends control information to a specified device. This description only provides general information about the parameter block; for more information, see Volume 2, "The Device Interface."

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 5; maximum is 5.

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

`code` Word input value: A number indicating the type of control request being made. The control requests are described completely in Chapter 1 of Volume 2. Control codes of $0000-$7FFF are standard status calls that must be supported by the device driver. Device-specific control calls may be supported by a particular device; they use status codes $8000-$FFFF. A list of standard control codes is 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)

`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 code as described in Volume 2, "The Device Interface."

`requestCount` Longword input value: For control codes that have a control list, this parameter gives the size of the control list.

`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.

Errors

$11	invalid device number
$53	parameter out of range

$2002 Destroy

Description

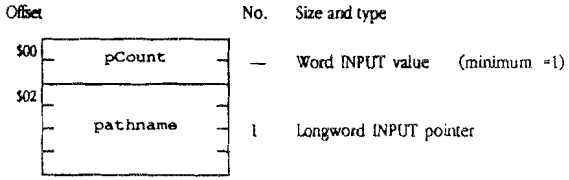
This call deletes a specified standard file, extended file (both the data fork and resource fork), or subdirectory, and updates the state of the file system to reflect the deletion. After a file is destroyed, no other operations on the file are possible.

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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 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.

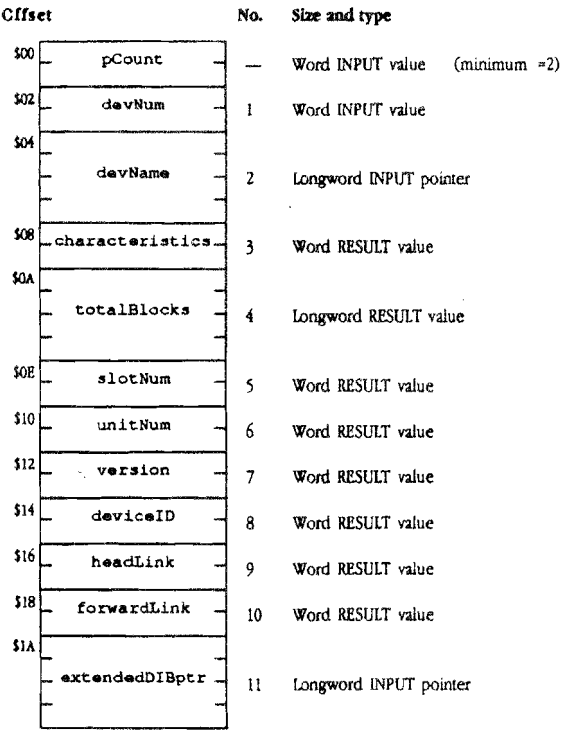
Errors

$10	device not found
$27	I/O error
$2B	write-protected disk
$40	invalid pathname syntax
$44	path not found
$45	volume not found
$46	file not found
$4B	unsupported storage type
$4E	access: file not destroy-enabled
$50	file open
$52	unsupported volume type
$53	invalid parameter
$58	not a block device
$5A	block number out of range

$202C DInfo

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

Parameters

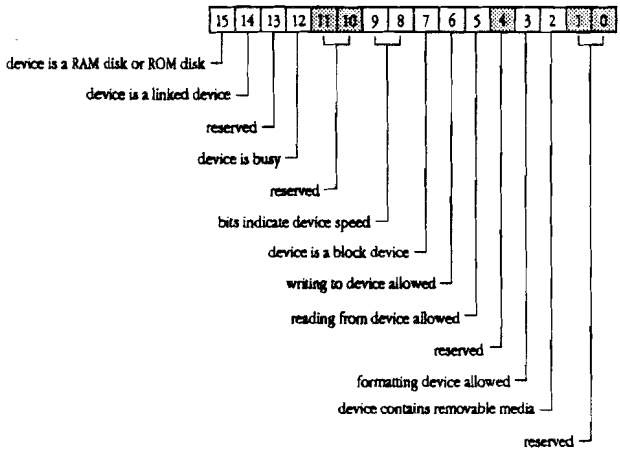


`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 11.

`devNum` Word input value: A device number. GS/OS assigns device numbers in sequence 1, 2, 3,... 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, one makes repeated calls to DInfo with `devNum` values of 1, 2, 3,... 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:

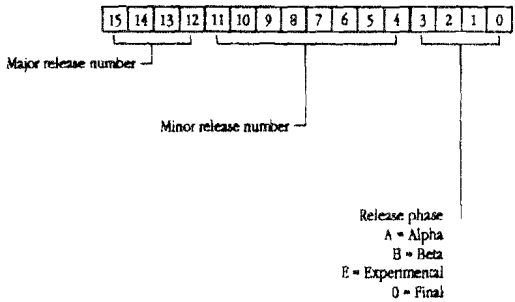


`totalBlocks` Longword result value: If the device is a block device, this parameter gives the maximum number of blocks on volumes handled by the device. For character devices, this parameter returns zero.

`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: An identifying number associated with a particular type of device.

This parameter may be useful for Finder-type 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)	$0010	File Server
		$0011	Reserved
		$0012	AppleDesktop Bus
$0001	Profile 5 MB	$0013	Hard disk (generic)
$0002	Profile 10 MB	$0014	Floppy disk (generic)
$0003	Apple 3.5 Drive (includes UniDisk 3.5 Drive)	$0015	Tape drive (generic)
		$0016	Character device driver (generic)
		$0017	MFM-encoded disk drive
		$0018	AppleTalk network (generic)
$0004	SCSI (generic)	$0019	Sequential access device
$0005	SCSI hard disk	$001A	SCSI scanner
$0006	SCSI tape drive	$001B	Other scanner
$0007	SCSI CD ROM	$001C	LaserWriter SC
$0008	SCSI printer	$001D	AppleTalk main driver
$0009	Serial modem	$001E	AppleTalk file service driver
$000A	Console driver	$001F	AppleTalk RPM driver
$000B	Serial printer		
$000C	Serial Laser Writer		
$000D	AppleTalk LaserWriter		
$000E	RAM Disk		
$000F	ROM Disk		

`headLink` Word result value: A 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 are associated with each other because they represent distinct partitions on a single disk medium. A value of 0 indicates that no link exists.

`forwardLink` Word result value: A device number that describes a link to another device. It is the device number of the next device in a linked list of devices that are associated with each other because they represent distinct partitions on a single disk. 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	invalid device number
$53	parameter out of range

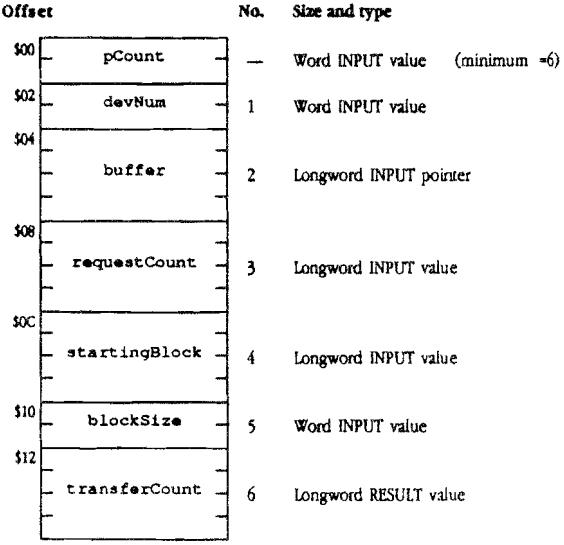
$202F DRead

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 Volume 2, "The Device Interface."

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 6; maximum is 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: The 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: The number of bytes actually transferred by the call.

Errors

$11	invalid device number
$53	parameter out of range

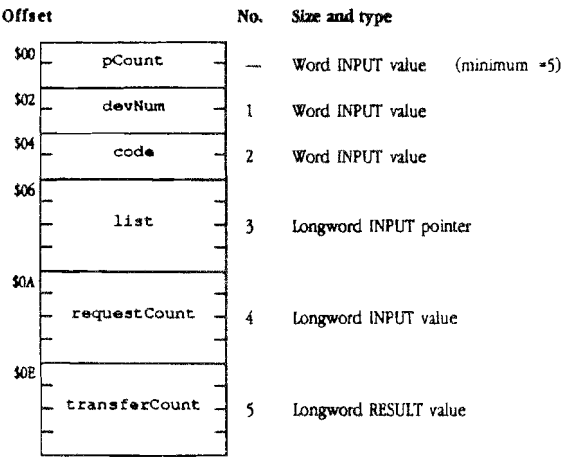
$202D DStatus

Description

Returns status information about a specified device.

This description provides only general information about the call; for more information, see Volume 2, "The Device Interface."

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 5; maximum is 5.

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

`code` Word input value: A number indicating the type of status request being made. The status requests are described completely in Volume 2, "The Device Interface." 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	GetDeviceStatus
$0001	GetConfigParameters
$0002	GetWaitStatus
$0003	GetFormatOptions
$0004	GetPartitionMap
$0005-$7FFF	(reserved)
$8000-$FFFF	(device specific subcalls)

`list` Longword input pointer: Points to a buffer in which the device returns its status information. Details about the status list are provided in Chapter 1 of Volume 2.

`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	invalid device number
$53	parameter out of range

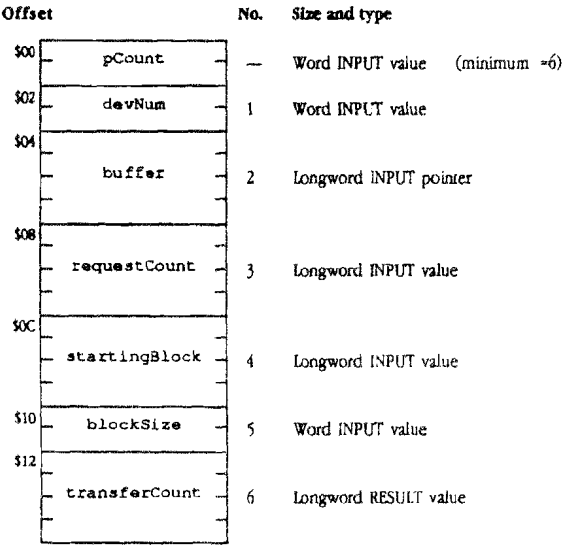
$2030 DWrite

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 Volume 2, "The Device Interface."

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 6; maximum is 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: The 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: The number of bytes actually transferred by the call.

Errors

$11	invalid device number
$53	parameter out of range

$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: The number of parameters in this parameter block. Minimum is 0; maximum is 0.

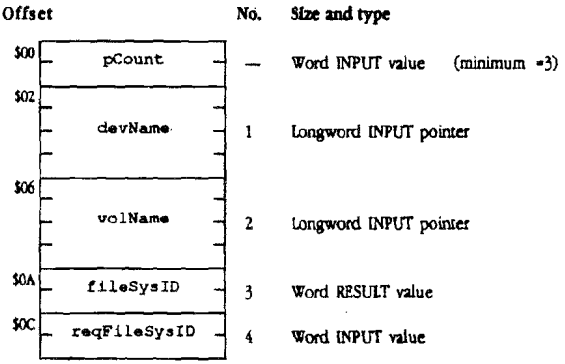
Errors (none)

$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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 3; maximum is 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	$0007	LISA
$0001	ProDOS/SOS	$0008	Apple CP/M
$0002	DOS 3.3	$0009	reserved
$0003	DOS 3.2 or 3.1	$000A	MS/DOS
$0004	Apple II Pascal	$000B	High Sierra
$0005	Macintosh (MFS)	$000C	ISO 9660
$0006	Macintosh (HFS)	$000D–$FFFF	reserved

`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 initialization dialog 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 device not found
$11 invalid device request
$27 I/O error
$28 no device connected
$2B write-protected disk
$40 invalid pathname syntax
$53 parameter out of range
$58 not a block device
$5D file system not available
$64 invalid FST ID

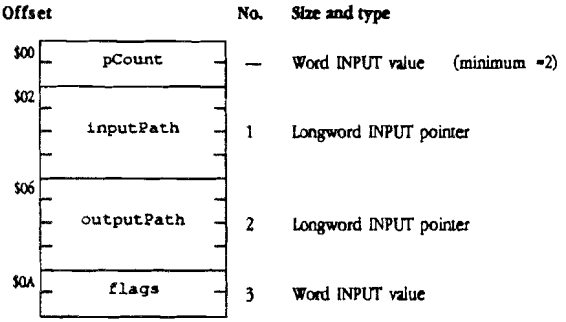
$200E ExpandPath

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 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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 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	invalid pathname syntax
$4F	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 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. 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: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`refNum` 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	I/O error
$2B	disk write protected
$2E	disk switched
$43	invalid reference number
$48	volume full
$5A	block number out of range

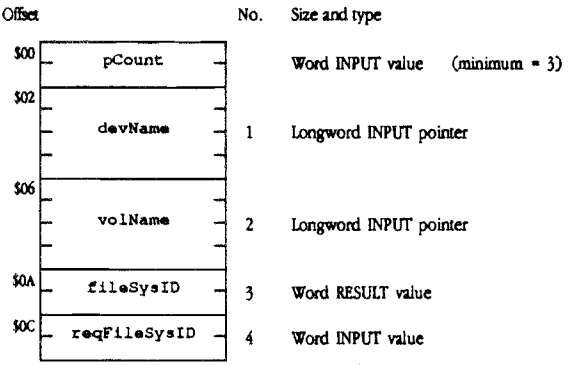
$2024 Format

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 which 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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 3; maximum is 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	$0007	LISA
$0001	ProDOS/SOS	$0008	Apple CP/M
$0002	DOS 3.3	$0009	reserved
$0003	DOS 3.2 or 3.1	$000A	MS/DOS
$0004	Apple II Pascal	$000B	High Sierra
$0005	Macintosh (MFS)	$000C	ISO 9660
$0006	Macintosh (HFS)	$000D–$FFFF	reserved

`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 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 FST.

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

$10	device not found
$11	invalid device request
$27	I/O error
$28	no device connected
$2B	disk is write protected
$40	invalid pathname syntax
$53	parameter out of range
$58	not a block device
$5D	file system not available
$64	invalid FST ID

$2020 GetDevNumber

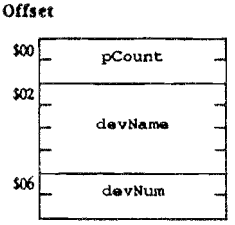
Description

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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 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: The device number of the specified device.

Errors

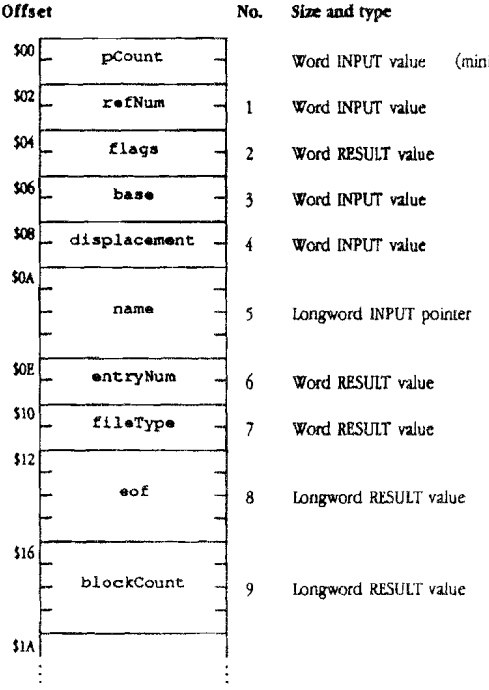
$10	device not found
$11	invalid device request
$40	invalid device or volume name syntax
$45	volume not found

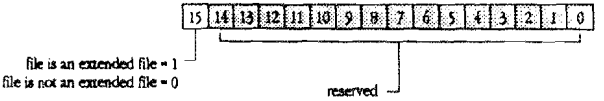
$201C GetDirEntry

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





`base` Word input value: A value that tells how to interpret the displacement parameter, 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: The 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: The value of the file type of the directory entry.

`eof` Longword result value: For a standard file, this parameter gives 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, this parameter gives 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: The value of the creation date and time of the directory entry. The format of the date and time is shown in Table 4-1 in Chapter 4.

`modDateTime` Double longword result value: The value of the modification date and time of the directory entry. The format of the date and time is shown in Table 4-1 in Chapter 4.

`access` Word result value: Value of the access attribute of the directory entry.

`auxType` Longword result value: Value of the 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 which 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.

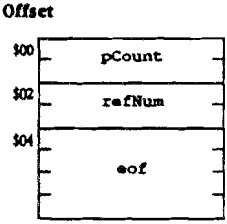
`resourceEOF` Longword result value: If the specified file is an extended file, this parameter gives the number of bytes that can be read from the file's resource fork. Otherwise, the parameter is undefined.

`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.

$2019 GetEOF

Description This function returns the current logical size of a specified file. See also the SetEOF call.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 2.

`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 invalid reference number

Errors

$10	device not found
$27	I/O error
$4A	version error
$4B	unsupported storage type
$4F	buffer too small
$52	unsupported volume type
$53	invalid parameter
$58	not a block device
$61	end of directory

$2006 GetFileInfo

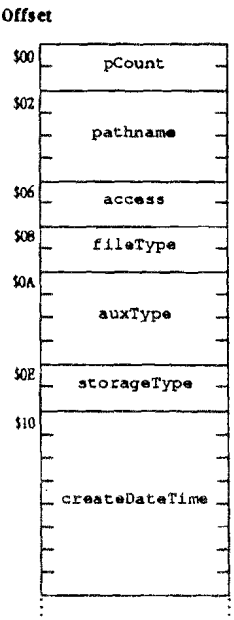
Description

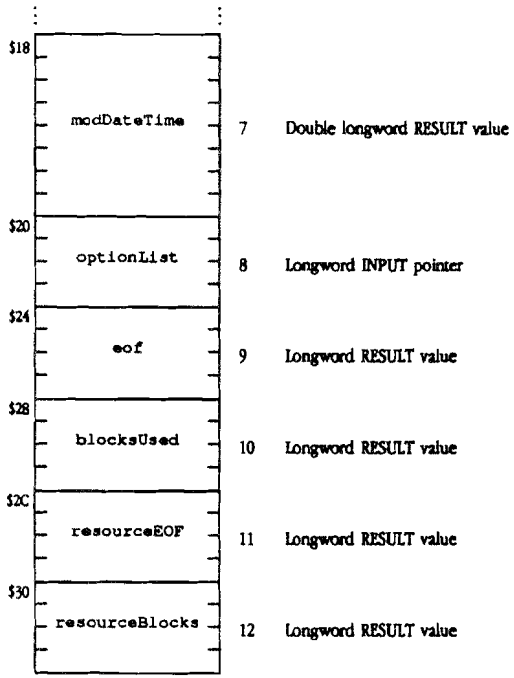
This call returns certain file attributes of an existing open or closed block file.

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 SetFileInfo call.

Parameters





`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 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: 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.

$01	standard file
$05	extended file
$0D	volume directory or subdirectory file

`createDateTime` Double longword result value: Value of the file's creation date and time attributes. The format of the date and time is shown in Table 4-1 in Chapter 4.

`modDateTime` Double longword result value: Value of the file's modification date and time attributes. The format of the date and time is shown in Table 4-1 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, this parameter gives 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.

For a subdirectory or a volume directory file, this parameter is undefined.

`blocksUsed` Longword result value: For a standard file, this parameter gives the total 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.

For a subdirectory or a volume directory file, this parameter is undefined.

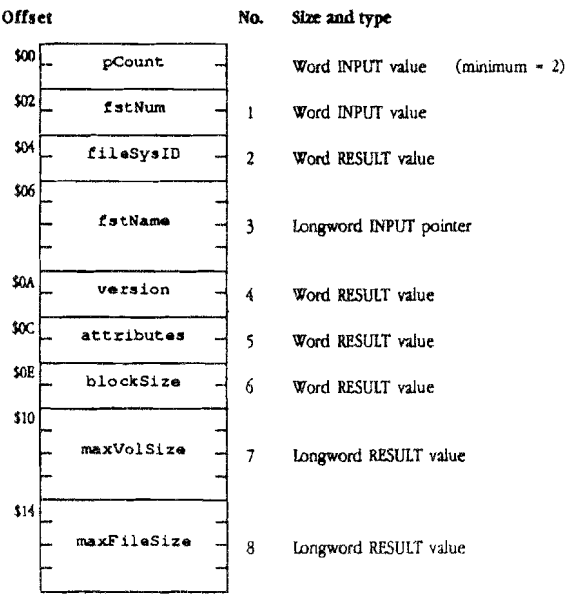
`resourceEOF` Longword result value: If the specified file is an extended file, this parameter gives the number of bytes that can be read from the file's resource fork. Otherwise, the parameter is undefined.

`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.

$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 guide.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 8.

`fstNum` Word input value: An 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, one makes repeated calls to GetFSTInfo with `fstNum` values of 1, 2, 3, and so on until GS/OS returns error $53: parameter out of range.

`maxFileSize` Longword result value: The maximum size (in bytes) of files handled by the FST.

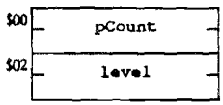
Errors

$53 parameter out of range

$201B GetLevel

Description This function returns the current value of the system file level. See also the SetLevel call.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`level` Word result value: The value of the system file level.

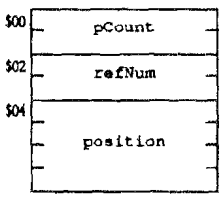
Errors

$01 bad system call number
$04 parameter count out of range
$07 ProDOS is busy
$59 invalid file level

$2017 GetMark

Description This function returns the current file mark for the specified file. See also the SetMark call.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 2.

`refNum` Word input value: The identifying number assigned to the file by the Open call.

`position` Longword result value: The current value of the file mark in bytes relative to the beginning of the file.

Errors

$43 invalid reference number

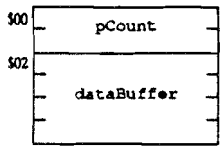
$2027 GetName

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 1/ with the filename returned by this call. Do this before making any change in prefix 1/.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`dataBuffer` Longword input pointer: Points to a result buffer where the filename is to be returned.

Errors

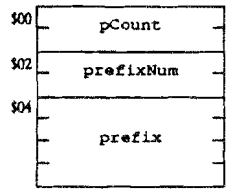
$4F buffer too small

$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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 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 structure where the prefix value is returned.

Errors

$4F buffer too small
$53 invalid parameter

$200F GetSysPrefs

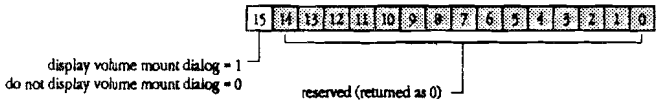
Description This call returns the value of the current global system preferences. The value of system preferences affects the behavior of some system calls. See also the SetSysPrefs call.

Parameters

Offset		No.	Size and type
$00	pCount		Word INPUT value (minimum = 1)
$02	preferences	1	Word RESULT value

`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`preferences` Word result value: Value of system preferences, as follows:



Errors (none)

$202A GetVersion

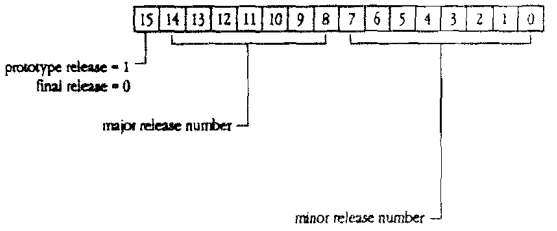
Description This call returns the version number of the GS/OS operating system. This value can be used by application programs to condition version-dependent operations.

Parameters

Offset		No.	Size and type
$00	pCount		Word INPUT value (minimum = 1)
$02	version	1	Word RESULT value

`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`version` Word result value: Version number of the operating system, in the following format:



Errors (none except general system errors)

$2011 NewLine

Description

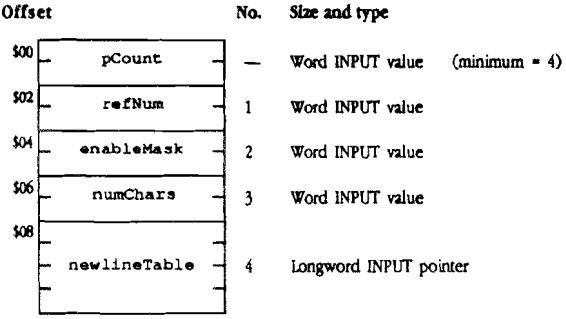
This function enables or disables the newline read mode for an open file and, when enabling newline read mode, specifies the newline enable mask and newline character or characters.

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 there is another character 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 continues at step 1.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 4; maximum is 4.

`refNum`	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 parameter is $0000, disable 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: The number of newline characters contained in the newline character table. If the `enableMask` 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.

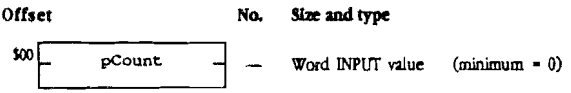
Errors

$43	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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 0; maximum is 0.

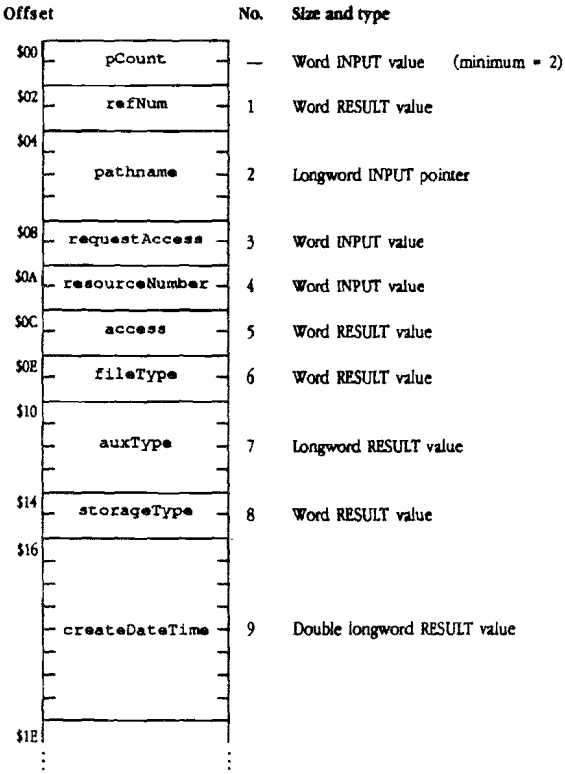
Errors (none)

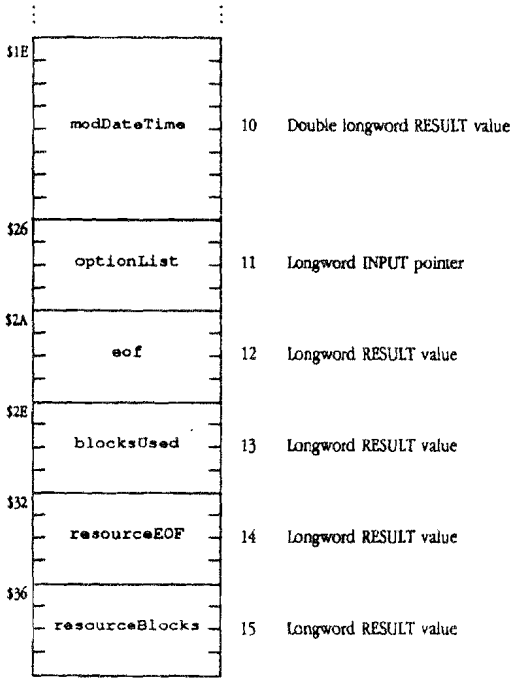
$2010 Open

Description This call causes GS/OS to establish an access path to a file. Once an access path is established, the user may perform file 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

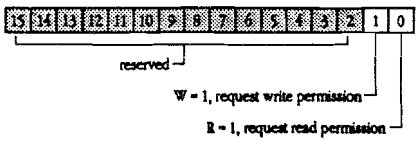


`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 15.

`refNum` Word result value: A 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 the desired access permissions, as follows:



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: This parameter is meaningful only when the `pathname` parameter specifies an extended file. In this 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: 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. Values are shown in Table 1-2 in Chapter 1.

`auxType` Longword result value: Value of the file's auxiliary type attribute. Values are shown in Table 1-2 in Chapter 1.

`storageType` Word result value: Value of the file's storage type attribute, as follows:

$01	standard file
$05	extended file
$0D	volume directory or subdirectory file

`createDateTime` Double longword result value: Value of the file's creation date and time attributes. The format of the date and time is shown in Table 4-1 in Chapter 4.

`modDateTime` Double longword result value: Value of the file's modification date and time attributes. The format of the date and time is shown in Table 4-1 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, this parameter gives 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.

For a subdirectory or volume directory file, this parameter is undefined.

`blocksUsed` Longword result value: For a standard file, this parameter gives the number of bytes used by the file. For an extended file, this parameter gives 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, this parameter gives the number of bytes that can be read from the file's resource fork, even when one is opening the data 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, even if one is opening the data fork. Otherwise, the parameter is undefined.

Errors

$27	I/O error
$28	no device connected
$2E	disk switched
$40	invalid pathname syntax
$44	path not found
$45	volume not found
$46	file not found
$4A	version error
$4B	unsupported storage type
$4E	access not allowed
$4F	buffer too small
$50	file is open
$52	unsupported volume type
$58	not a block device

$2003 OSShutdown

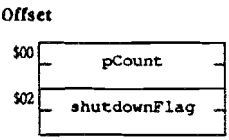
Description

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 cold reboot. 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 reboots 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 reboot. If the user chooses to reboot, GS/OS then looks at Bit 1 of the `shutdownFlag` parameter.

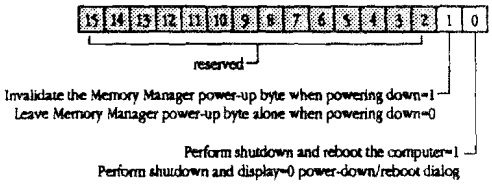
If Bit 1 is cleared to 0, GS/OS leaves the Memory Manager power-up byte alone; this leaves any RAM disks intact while the machine is rebooted. If Bit 1 is set to 1, however, GS/OS invalidates the power-up byte, which effectively destroys any RAM disk, before rebooting the computer.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`shutdownFlag` Word input value: Two Boolean flags that give 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 reboot the computer=1
Perform shutdown and display=0 power-down/reboot dialog

Errors (none)

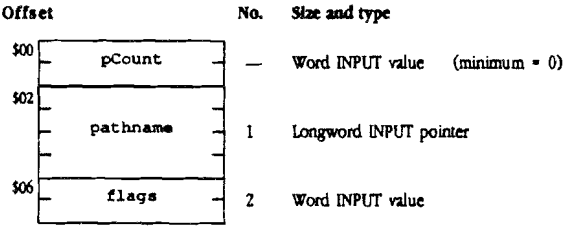
$2029 Quit

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."

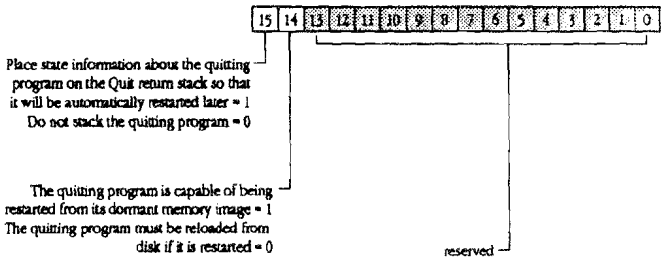
Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 0; maximum is 2.

`pathname` Longword input pointer: Points to a GS/OS string representing the pathname of the program to run next. If this 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:



Comments Only one error condition causes the Quit call to return to the calling application: error $07 (GS/OS busy). All other errors are managed within the GS/OS program dispatcher.

Errors

$07 GS/OS busy

$2012 Read

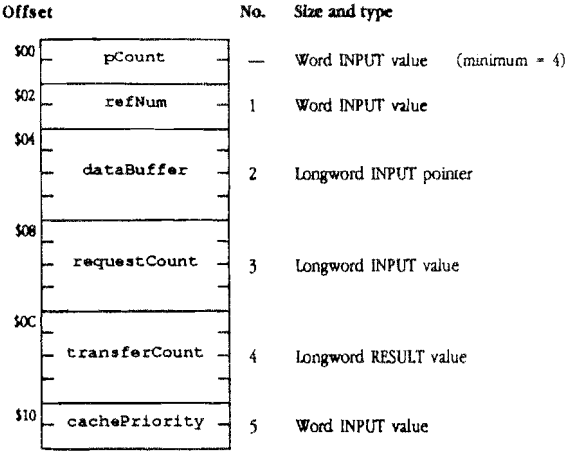
Description

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 of three 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 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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 4; maximum is 5.

`refNum` 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.

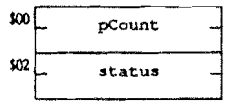
`cachePriority` Word input value: Specifies whether or not disk blocks handled by the read call are candidates for caching, as follows:

$0000 do not cache blocks involved in this read
$0001 cache blocks involved in this read if possible

$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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`status` Word result value: A value that tells whether or not a write-deferral session is in progress.

$0000 no session in progress
$0001 session in progress

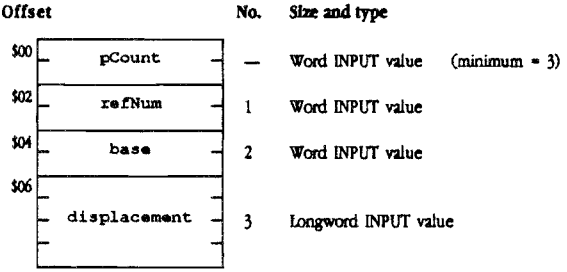
Errors (none)

$2018 SetEOF

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 GetEOF call.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 3; maximum is 3.

`refNum` Word input value: The identifying number assigned to the file by the Open call.

`base` Word input value: A value that tells how to interpret the `displacement` parameter.

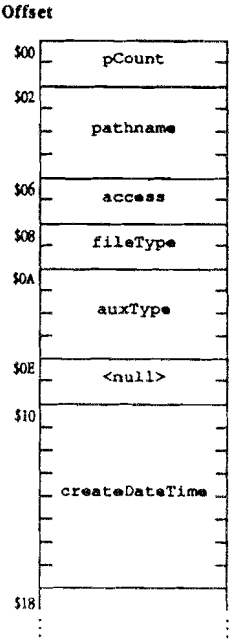
$0000 set EOF equal to displacement
$0001 set EOF equal to old EOF minus displacement
$0002 set EOF equal to file mark plus displacement
$0003 set EOF equal to file mark minus displacement

`displacement` Longword input value: Used to compute the new value of the eof as described for the `base` parameter.

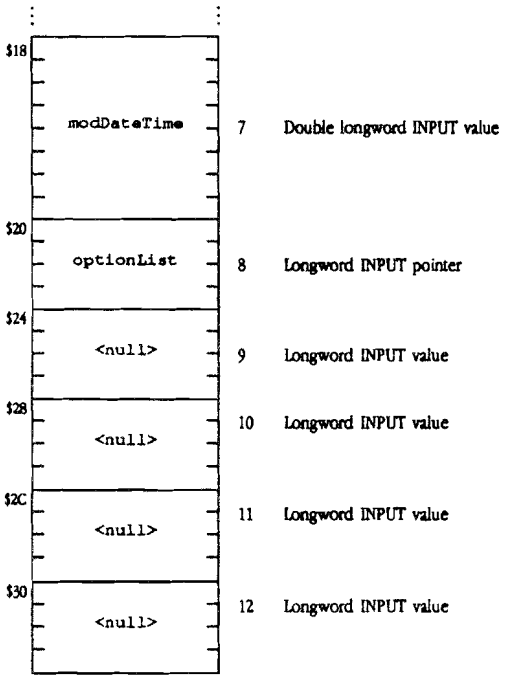
Errors

$27	I/O error
$2B	write-protected disk
$43	invalid reference number
$4D	position out of range
$4E	file not write-enabled
$5A	block number out of range

Parameters



No.	Size and type
—	Word INPUT value (minimum = 2)
1	Longword INPUT pointer
2	Word INPUT value
3	Word INPUT value
4	Longword RESULT value
5	Word INPUT value
6	Double longword INPUT value



pCount	Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 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: 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 result value: Value of the file's auxiliary type attribute.
<null>	Word input value: This parameter is unused and must be set to zero.

`createDateTime` Double longword input value: Value of the file's creation date and time attributes. 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 4-1 in Chapter 4.

`modDateTime` Double longword input value: Value of the file's modification date and time attributes. 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 4-1 in Chapter 4.

`optionList` Longword input pointer: Points to a GS/OS result buffer to which FST-specific information can be returned.

`<null>` Longword input value: This parameter is unused and must be set to zero.

`<null>` Longword input value: This parameter is unused and must be set to zero.

`<null>` Longword input value: This parameter is unused and must be set to zero.

`<null>` Longword input value: This parameter is unused and must be set to zero.

Errors

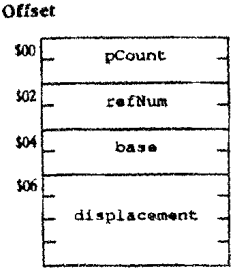
$10	device not found
$27	I/O error
$2B	write-protected disk
$40	invalid pathname syntax
$44	path not found
$45	volume not found
$46	file not found
$4A	version error
$4B	unsupported storage type
$4E	access: file not destroy-enabled
$52	unsupported volume type
$53	invalid parameter
$58	not a block device

$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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 3; maximum is 3.

`refNum` Word input value: The identifying number assigned to the file by the Open call.

`base` Word input value: A value that tells how to interpret the `displacement` parameter, 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: A value used to compute the new value for the file mark, as described for the `base` parameter.

Errors

$27	I/O error
$43	invalid reference number
$4D	position out of range
$5A	block number out of range

$2009 SetPrefix

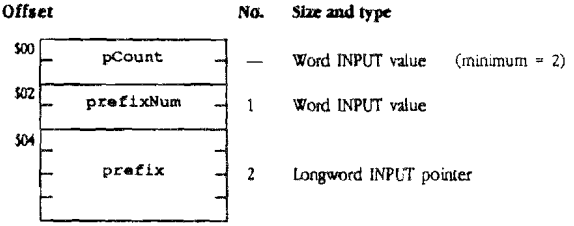
Description

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 described below. See also the GetPrefix call.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 2.

`prefixNum` Word input value: A 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.

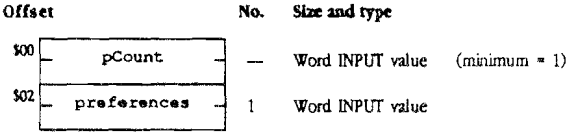
Errors

$40	invalid pathname syntax
$53	invalid parameter

$200C SetSysPrefs

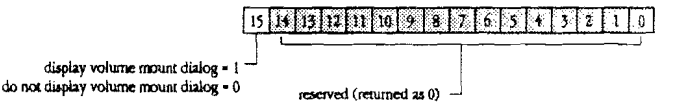
Description This call sets the value of the global system preferences. The value of system preferences affects the behavior of some system calls. See also the GetSysPrefs call.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`preferences` Word input value: Value of system preferences, as follows:



Comments Under certain circumstances, parts of the system call the system's Mount facility to display a dialog 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 always displays the dialog.

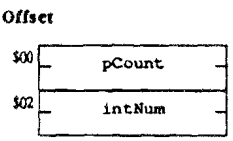
$2032 UnbindInt

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 Volume 2. See also the BindInt call.

Parameters



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 1; maximum is 1.

`intNum` Word input value: Interrupt identification number of the binding between interrupt source and interrupt handler that is to be undone.

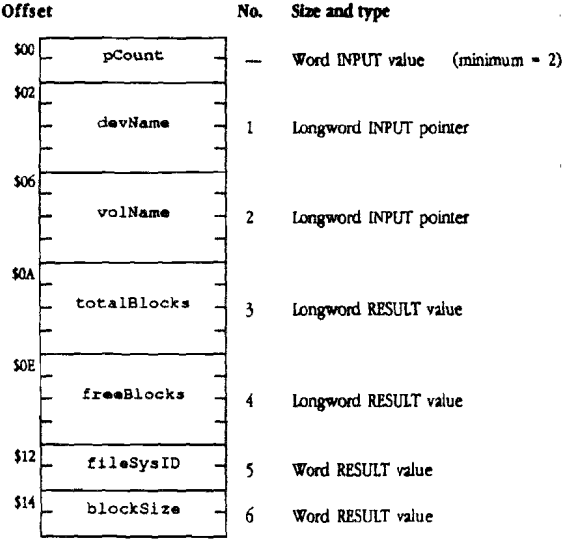
Errors

$53 parameter out of range

$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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 6.

`devName` Longword input pointer: Points to a GS/OS input string structure containing the name of a block device.

`volName` Longword input pointer: Points to a GS/OS output string structure 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	$0007	LISA
$0001	ProDOS/SOS	$0008	Apple CP/M
$0002	DOS 3.3	$0009	reserved
$0003	DOS 3.2 or 3.1	$000A	MS/DOS
$0004	Apple II Pascal	$000B	High Sierra
$0005	Macintosh (MFS)	$000C	ISO 9660
$0006	Macintosh (HFS)	$000D–$FFFF	reserved

`blockSize` Word result value: The size, in bytes, of a block.

Errors

$10	device not found
$11	invalid device request
$27	I/O error
$28	no device connected
$2E	disk switched
$45	volume not found
$4A	version error
$52	unsupported volume type
$53	invalid parameter
$57	duplicate volume
$58	not a block device

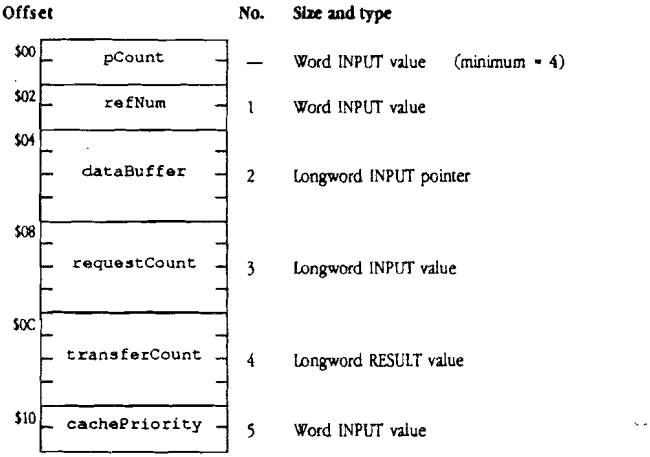
$2013 Write

Description

This call attempts to transfer the number of bytes specified by requestCount 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



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 4; maximum is 5.

`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.

`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 I/O error
$2B write-protected disk
$2E disk switched
$43 invalid reference number
$48 volume full
$4E access not allowed
$5A block number out of range