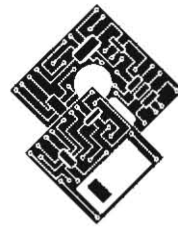
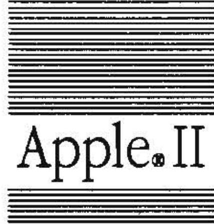


APPLE
PROGRAMMER'S
AND DEVELOPER'S
ASSOCIATION

GS/OS Reference, Volume 1 Beta Draft

APDA# K2S023



GS/OS™ Reference

Includes System Loader

Volume 1: Applications and GS/OS

APDA Draft

August 31, 1988

© Copyright Apple Computer, Inc. 1988

 Apple Computer, Inc.

This manual is copyrighted by Apple or by Apple's suppliers, with all rights reserved. Under the copyright laws, this manual may not be copied, in whole or in part, without the written consent of Apple Computer, Inc. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased may be sold, given, or lent to another person. Under the law, copying includes translating into another language.

© Apple Computer, Inc., 1988
20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Apple, the Apple logo, AppleTalk, Apple IIGS, DuoDisk, ProDOS, Macintosh, and IIGS are registered trademarks of Apple Computer, Inc.

APDA, Finder, ProFile, and UniDisk are trademarks of Apple Computer, Inc.

Simultaneously published in the United States and Canada.

2/21/88

Contents

Figures and Tables xiv

Preface / 1

About this book / 2

How to use this book / 2

- What it contains / 3
- Other materials you'll need / 5
- Visual cues / 5
- Terminology / 5
- Language notation / 6

Roadmap to the Apple IIGS technical manuals / 6

- Introductory Apple IIGS manuals / 7
- Apple IIGS machine-reference manuals / 9
- Apple IIGS Toolbox manuals / 10
- Apple IIGS operating-system manuals / 10
- All-Apple manuals / 11
- The APW manuals / 11
- The MPW IIGS manuals / 12
- The debugger manual / 12

Introduction What is GS/OS? / 13

The components of GS/OS / 14

GS/OS Features / 16

- File-system independence / 16
- Enhanced device support / 16
- Speed enhancements / 17
- Eliminated ProDOS restrictions / 17
- ProDOS 16 compatibility / 17

Where to find call descriptions / 17
GS/OS system requirements / 19
Background to the development of GS/OS / 20

Part I The Application Level / 23

1 GS/OS Abstract File System / 25
A high-level interface / 26
Classes of GS/OS files / 28
Directory files / 28
Standard files / 29
Extended files / 30
Filenames / 30
Pathnames / 31
Full pathnames / 31
Prefixes and partial pathnames / 32
Prefix designators / 32
Predefined prefix designators / 33
File information / 34
File access / 35
File types and auxiliary types / 35
EOF and mark / 37
Creation and modification date and time / 39
Character devices as files / 39
Groups of GS/OS calls / 40
File access calls / 41
Volume and pathname calls / 42
System information calls / 43
Device calls / 43

2 GS/OS and Its Environment / 45

Apple IIGS memory / 46
    Entry points and fixed locations / 47
Managing application memory / 48
    Obtaining application memory / 49
    Accessing data in a movable memory block / 49
Allocating stack and direct page / 51
    Automatic allocation of stack and direct page / 52
    Definition during program development / 52
    Allocation at load time / 52
    GS/OS default stack and direct page / 53
System startup considerations / 54
Quitting and launching applications / 54
    Specifying whether an application can be restarted from memory / 54
    Specifying the next application to launch / 55
        Specifying a GS/OS application to launch / 55
        Specifying a ProDOS 8 application to launch / 55
    Specifying whether control should return to your application / 56
    Quitting without specifying the next application to launch / 56
    Launching another application and not returning / 56
    Launching another application and returning / 57
Machine state at application launch / 57
    Machine state at GS/OS application launch / 57
    Machine state at ProDOS 8 application launch / 59
    Pathname prefixes at GS/OS application launch / 59
    Pathname prefixes at ProDOS 8 application launch / 61

3 Making GS/OS Calls / 63

GS/OS call methods / 64
- Calling in a high-level language / 64
- Calling in assembly language / 64
  - Making a GS/OS call using macros / 65
  - Making an inline GS/OS call / 66
  - Making a stack call / 66
- Including the appropriate files / 67

GS/OS parameter blocks / 67
- Types of parameters / 67
- Parameter block format / 68
- GS/OS string format / 68
  - GS/OS input string structures / 69
  - GS/OS result buffer / 69

Setting up a parameter block in memory / 70

Conditions upon return from a GS/OS call / 71

Checking for errors / 72

4 Accessing GS/OS Files / 73

The simplest access method / 74

Creating a file / 74

Opening a file / 75

Working on open files / 76
- Reading from and writing to files / 76
- Setting and reading the EOF and Mark / 77
- Enabling or disabling newline mode / 77
- Examining directory entries / 77
- Flushing open files / 77
- Closing files / 77

Setting and getting file levels / 78

Working on closed files / 78
- Clearing backup status / 79
- Deleting files / 79

Setting or getting file characteristics / 79
Changing the creation and modification date and time / 80
Copying files / 81
Copying single files / 81
Copying multiple files / 81

5 Working with Volumes and Pathnames / 83
Working with volumes / 83
Getting volume information / 84
Building a list of mounted volumes / 84
Getting the name of the boot volume / 84
Formatting a volume / 85
Working with pathnames / 85
Setting and getting prefixes / 86
Changing the path to a file / 86
Expanding a pathname / 86
Building your own pathnames / 86
Introducing devices / 87
Device names / 87
Block devices / 87
Character devices / 88
Direct access to devices / 88
Device drivers / 88

6 Working with System Information / 91
Setting and getting system preferences / 92
Checking FST information / 92
Finding out the version of the operating system / 92
Getting the name of the current application / 93

7 GS/OS Call Reference / 95

The parameter block diagram and description / 96

$201D BeginSession / 97

$2031 BindInt / 98

$2004 ChangePath / 99

$200B ClearBackup / 101

$2014 Close / 102

$2001 Create / 103

$202E DControl / 108

$2002 Destroy / 110

$202C DInfo / 112

$202F DRead / 116

$202D DStatus / 118

$2030 DWrite / 120

$201E EndSession / 122

$2025 EraseDisk / 123

$200E ExpandPath / 125

$2015 Flush / 127

$2024 Format / 129

$2028 GetBootVol / 131

$2020 GetDevNumber / 132

$201C GetDirEntry / 134

$2019 GetEOF / 139

$2006 GetFileInfo / 140

$202B GetFSTInfo / 144

$201B GetLevel / 147

$2017 GetMark / 148

$2027 GetName / 149

$200A GetPrefix / 150

$200F GetSysPrefs / 151

$202A GetVersion / 152
$2011 NewLine / 153
$200D Null / 155
$2010 Open / 156
$2003 OSShutdown 161
$2029 Quit / 163
$2012 Read / 165
$201F SessionStatus / 168
$2018 SetEOF / 169
$2005 SetFileInfo / 171
$201A SetLevel / 175
$2016 SetMark / 176
$2009 SetPrefix / 178
$200C SetSysPrefs / 180
$2032 UnbindInt / 182
$2008 Volume / 183
$2013 Write / 185

Part II The File System Level / 187

8 File System Translators / 189

The FST Concept / 190
Calls handled by FSTs / 192
Programming for multiple file systems / 193
Don't assume file characteristics / 193
Use GetDirEntry / 194
Keep rebuilding your device list / 194
Handle errors properly / 194
FSTs and file-access optimization / 195
Present and future FSTs / 195
Disk initialization and FSTs / 196

9 The ProDOS FST / 199

The ProDOS file system / 200

GS/OS and the ProDOS FST / 200

Calls to the ProDOS FST / 201

- GetDirEntry ($201C) / 201
- GetFileInfo ($2006) / 202
- SetFileInfo ($2005) / 202

10 The High Sierra FST / 203

CD-ROM and the High Sierra/ISO 9660 formats / 204

Limitations of the High Sierra FST / 205

Apple extensions to ISO 9660 / 207

High Sierra FST calls / 208

- GetFileInfo ($2006) / 209
- Volume ($2008) / 210
- Open ($2010) / 210
- Read ($2012) / 211
- GetDirEntry ($201C) / 212

$2033 FSTSpecific / 214

- What a map table is / 215
- MapEnable (FSTSpecific subcall) / 216
- GetMapSize (FSTSpecific subcall) / 217
- GetMapTable (FSTSpecific subcall) / 217
- SetMapTable (FSTSpecific subcall) / 218

11 The Character FST / 221

Character devices as files / 222

Character FST calls / 222

- Open ($2010) / 223
- Read ($2012) / 223
- Write ($2013) / 224
- Close ($2014) / 224
- Flush ($2015) / 225

Appendixes / 227

Appendix A GS/OS ProDOS 16 Calls / 229

$0031 ALLOC_INTERRUPT / 230
$0004 CHANGE_PATH / 231
$000B CLEAR_BACKUP_BIT / 233
$0014 CLOSE / 234
$0001 CREATE / 235
$0032 DEALLOC_INTERRUPT / 239
$0002 DESTROY / 240
$002C D_INFO / 242
$0025 ERASE_DISK / 243
$000E EXPAND_PATH / 245
$0015 FLUSH / 247
$0024 FORMAT / 248
$0028 GET_BOOT_VOL / 250
$0020 GET_DEV_NUM / 251
$001C GET_DIR_ENTRY / 252
$0019 GET_EOF / 256
$0006 GET_FILE_INFO / 257
$0021 GET_LAST_DEV / 260
$001B GET_LEVEL / 262
$0017 GET_MARK / 263
$0027 GET_NAME / 264
$000A GET_PREFIX / 265
$002A GET_VERSION / 266
$0011 NEWLINE / 267
$0010 OPEN / 269
$0029 QUIT / 271
$0012 READ / 273

$0022 READ_BLOCK / 275

$0018 SET_EOF / 276

$0005 / SET_FILE_INFO / 277

$001A SET_LEVEL / 280

$0016 SET_MARK / 281

$0009 SET_PREFIX / 282

$0008 VOLUME / 284

$0013 WRITE / 286

$0023 WRITE_BLOCK / 288

Appendix B ProDOS 16 Calls and FSTs / 289

The ProDOS FST / 290

The High Sierra FST / 290

GET_FILE_INFO ($06) / 291

VOLUME ($08) / 292

GET_DIR_ENTRY ($1C) / 292

The Character FST / 293

OPEN ($10) / 293

READ ($12) / 294

WRITE ($13) / 294

CLOSE ($14) / 294

FLUSH ($15) / 295

ProDOS 16 device calls / 295

Appendix C The GS/OS Exerciser / 297

Starting the Exerciser / 298

Call options / 299

Making GS/OS calls / 299

Other commands / 301

Appendix D GS/OS System Disks and Startup / 305

Application system disks / 306

System startup from ProDOS volumes / 307

System startup from non-ProDOS volumes / 308

Startup (boot file routine) / 309

ReadInFile (boot file routine) / 310

GetBootName (boot file routine) / 311

GetFSTName (boot file routine) / 311

Sample boot file startup routine / 312

Appendix E Apple Extensions to ISO 9660 / 317

What the Apple extensions do / 318

The protocol identifier / 318

The Directory Record SystemUse Field / 320

SystemUseID / 322

Filename transformations / 324

ProDOS / 324

Macintosh HFS / 325

ISO 9660 associated files / 326

Appendix F GS/OS Error Codes and Constants / 327

Glossary / 331

Figures and Tables

Preface / 1

Figure P-1. Roadmap to Apple IIGS technical manuals / 8

Table P-1 Apple IIGS technical manuals / 9

Introduction What is GS/OS? / 13

Figure I-1 Interface levels in GS/OS / 14

Figure I-2 Where to find call descriptions in this book. / 19

Part I The Application Level / 23

Chapter 1 GS/OS Abstract File System / 25

Figure 1-1 Application level in GS/OS / 26

Figure 1-2 Example of a hierarchical file structure / 27

Figure 1-3 Directory file format / 29

Figure 1-4 Prefixes and partial pathnames / 32

Figure 1-5 Automatic movement of EOF and mark / 38

Table 1-1 Examples of prefix use / 34

Table 1-2 GS/OS file types and auxiliary types / 36

Table 1-3 GS/OS call groups / 41

Chapter 2 GS/OS and Its Environment / 45

Figure 2-1 Apple IIGS memory map / 46

Figure 2-2 Pointers and handles / 51

Table 2-1 GS/OS vector space / 48

Table 2-2 Machine state at GS/OS application launch / 57

Table 2-3 Machine state at GS/OS application launch / 59

Table 2-4 Prefix values when GS/OS application launched at boot time / 60

Table 2-5 Prefix values—GS/OS application launched after GS/OS application quits / 60

Table 2-6 Prefix values—GS/OS application launched after ProDOS 8 application quits / 60

Table 2-7 Prefix and pathname values at ProDOS 8 application launch / 61

Chapter 3 Making GS/OS Calls / 63

Figure 3-1 GS/OS and Pascal strings / 69

Figure 3-2 GS/OS input string structure / 69

Figure 3-3 GS/OS result buffer / 70

Table 3-1 Registers on exit from GS/OS / 71

Table 3-2 Status and control bits on exit from GS/OS / 72

Chapter 4 Accessing GS/OS Files / 73

Table 4-1 Date and time format / 80

Part II The File System Level / 187

Chapter 8 File System Translators / 189

Figure 8-1 The file system level in GS/OS / 191

Table 8-1 GS/OS calls handled by FSTs / 192

Chapter 10 The High Sierra FST / 203

Table 10-1 High Sierra FST calls / 208

Appendixes / 227

Appendix B ProDOS 16 Calls and FSTs / 289

Table B-1 High Sierra FST ProDOS 16 calls / 291

Appendix C The GS/OS Exerciser / 297

Figure C-1 Exerciser main screen / 298

Figure C-2 Parameter-setup screen / 300

Figure C-3 Device-list screen / 302

Figure C-4 Modify-memory screen / 303

Table C-1 ASCII table / 304

Appendix D GS/OS System Disks and Startup / 305

Table D-1 Directories and files on a GS/OS system disk / 306

Appendix E Apple Extensions to ISO 9660 / 317

Table E-1 Defined values for SystemUseID / 322

Table E-2 Contents of SystemUse field for each value of SystemUseID / 322

Table E-3 ProDOS-to-ISO 9660 filename transformations / 325

Appendix F GS/OS Error Codes and Constants / 327

Table F-1 GS/OS errors / 328

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

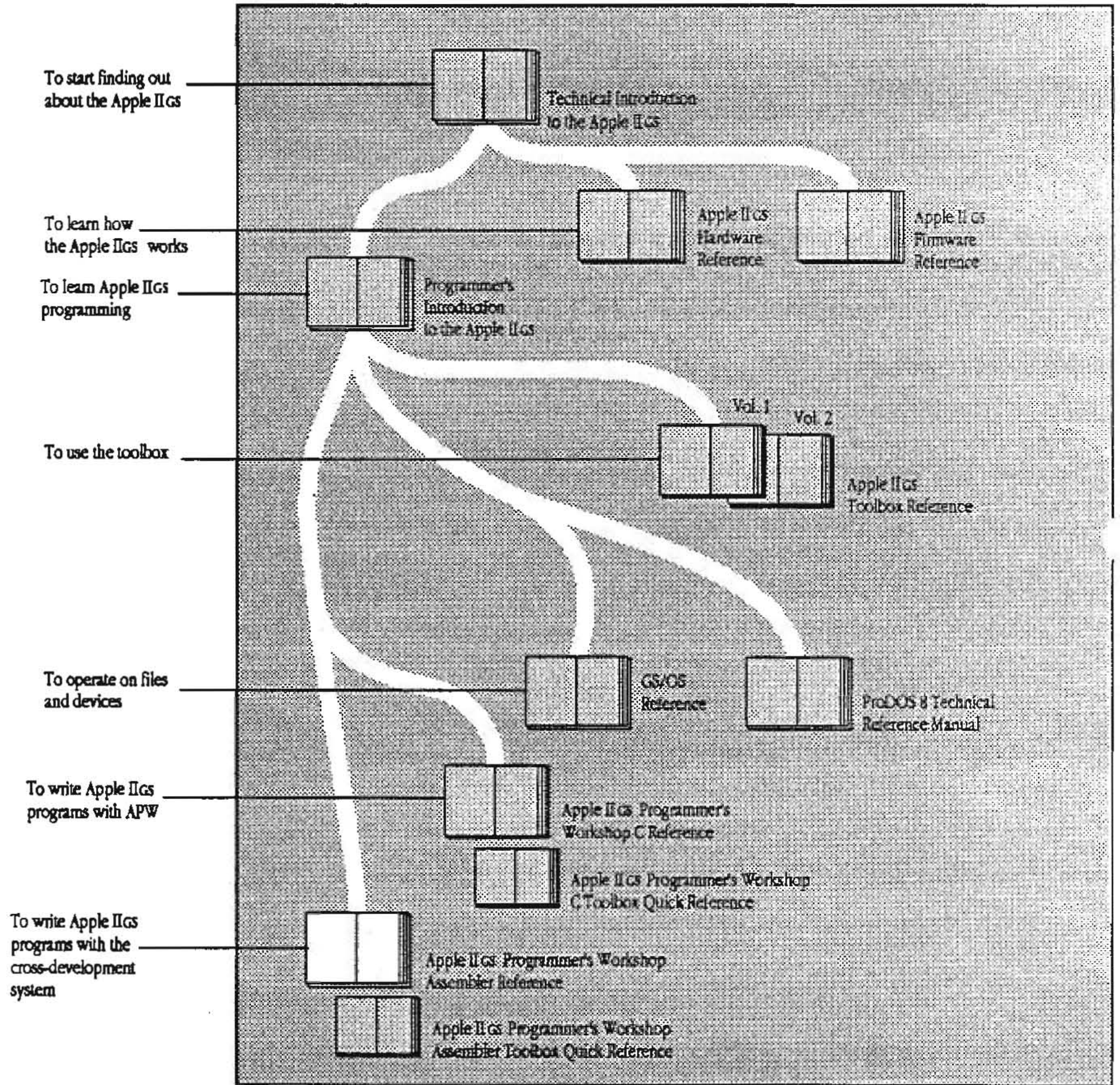


Table P-1 Apple IIGS technical manuals

Title	Subject
Technical Introduction to the Apple IIGS	What the Apple IIGS is
Apple IIGS Hardware Reference	Machine internals—hardware
Apple IIGS Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Apple IIGS	Concepts and a sample program
Apple IIGS Toolbox Reference, Volume 1	How tools work, some specifications
Apple IIGS Toolbox Reference, Volume 2	More toolbox specifications
ProDOS 8 Technical Reference Manual	Standard Apple II operating system
Apple IIGS ProDOS 16 Reference	Apple IIGS operating system and loader
Apple IIGS Programmer's Workshop Reference	Using APW
APW Assembler Reference	Using the APW Assembler
APW C Reference	Using the APW C Compiler
MPW IIGS Tools Reference	Using the cross-development system
MPW IIGS Assembler Reference	Using the MPW IIGS Assembler
MPW IIGS C Reference	Using the MPW IIGS C Compiler
MPW IIGS Pascal Reference	Using the MPW IIGS Pascal Compiler
Apple IIGS Debugger Reference	Debugger for all Apple IIGS programs

Apple IIGS machine-reference manuals

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

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

- **The 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.

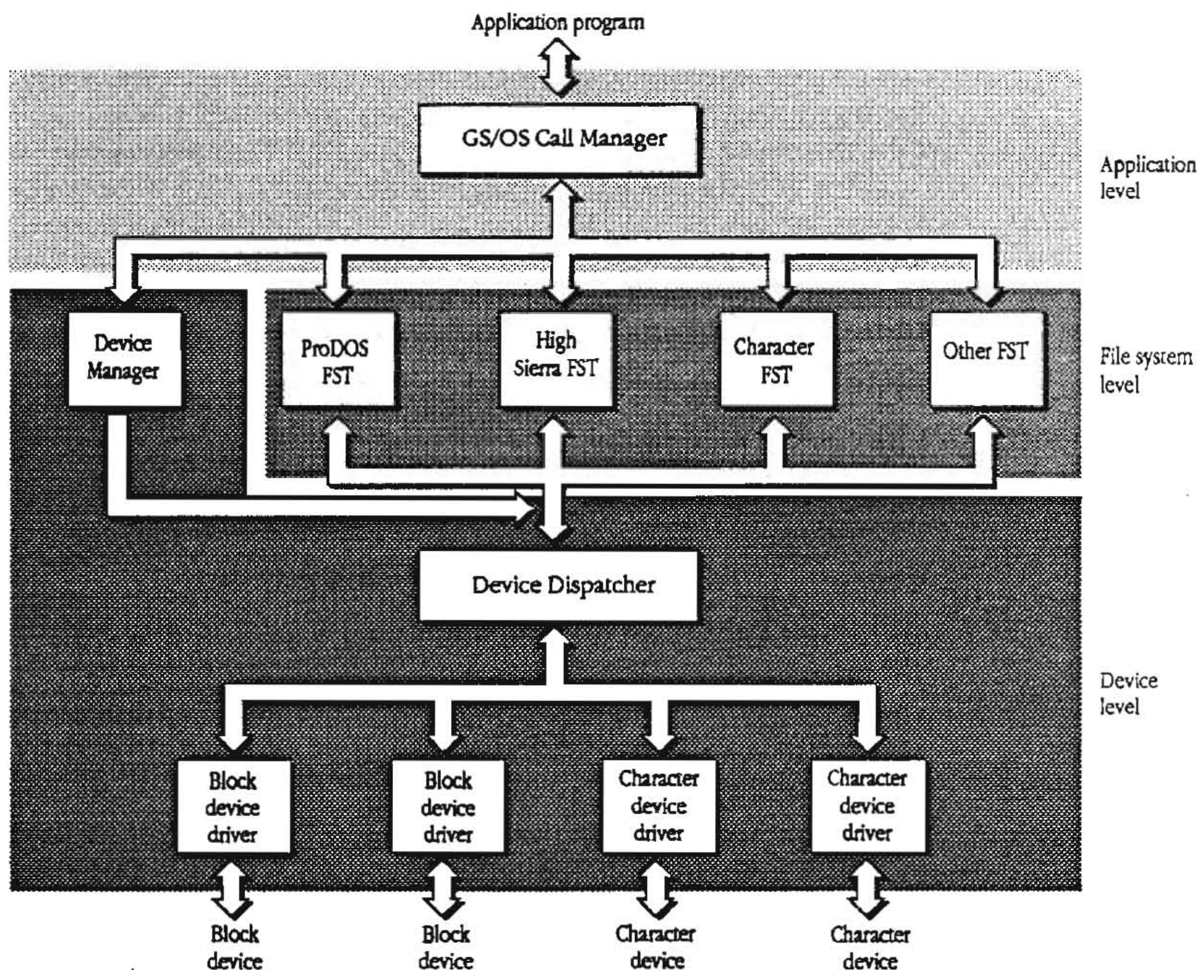
Even more important, GS/OS is file-system independent: by making GS/OS calls, your application can read and write files transparently among many different and normally incompatible file systems. GS/OS accomplishes this by defining a generic GS/OS file interface, the abstract file system. Your application 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 I-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 1 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:

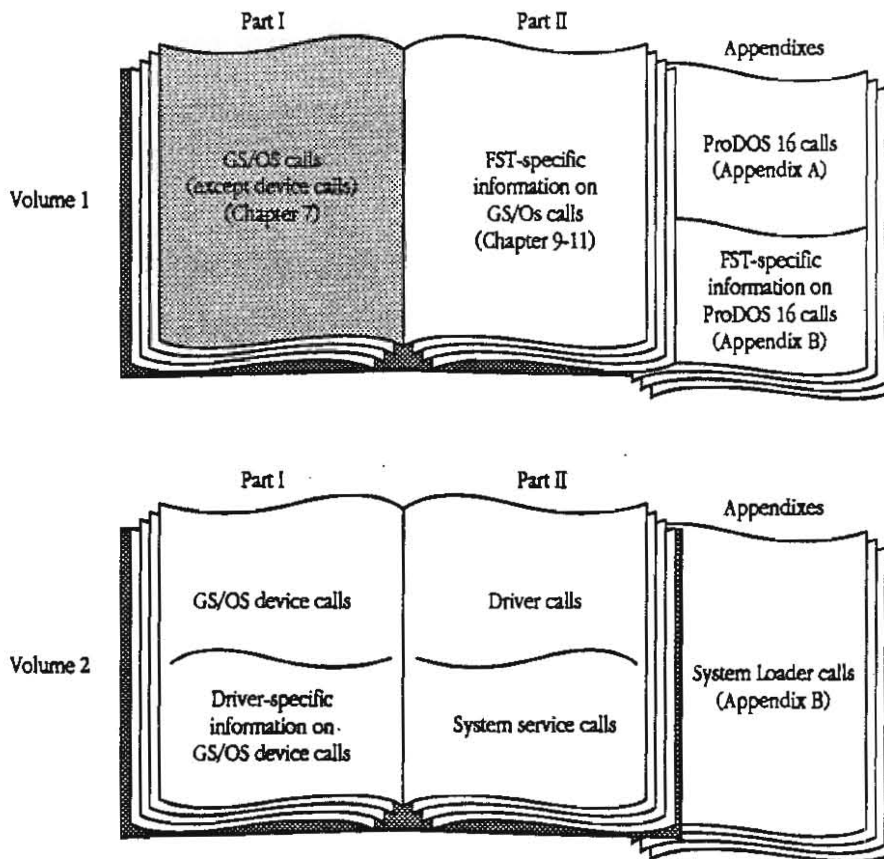
- **GS/OS device calls:** These are a subset of the application-level; standard GS/OS calls described in Volume 1, but they are special because they bypass the file level altogether and access devices directly.
- **Driver-specific information on GS/OS device calls:** Because different devices have different 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.

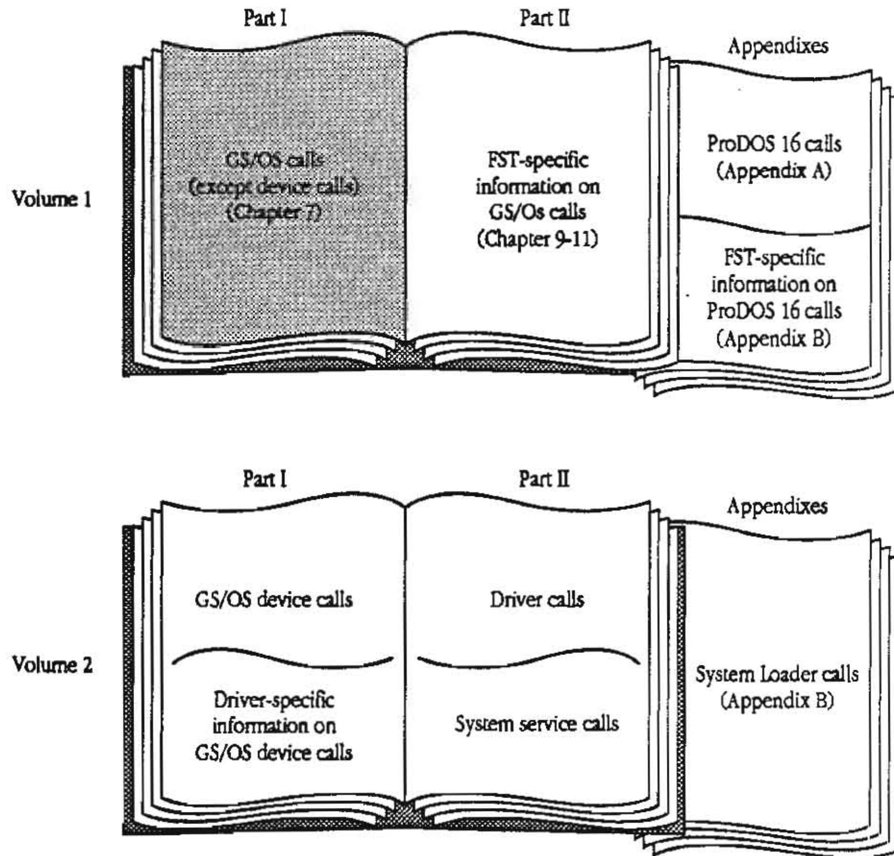
Apple has created several operating systems for the Apple II family of computers. GS/OS is the latest in that line; it is related to several earlier systems, but has far greater capabilities than any of them. Here are thumbnail sketches of the other systems:

- **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 modem 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



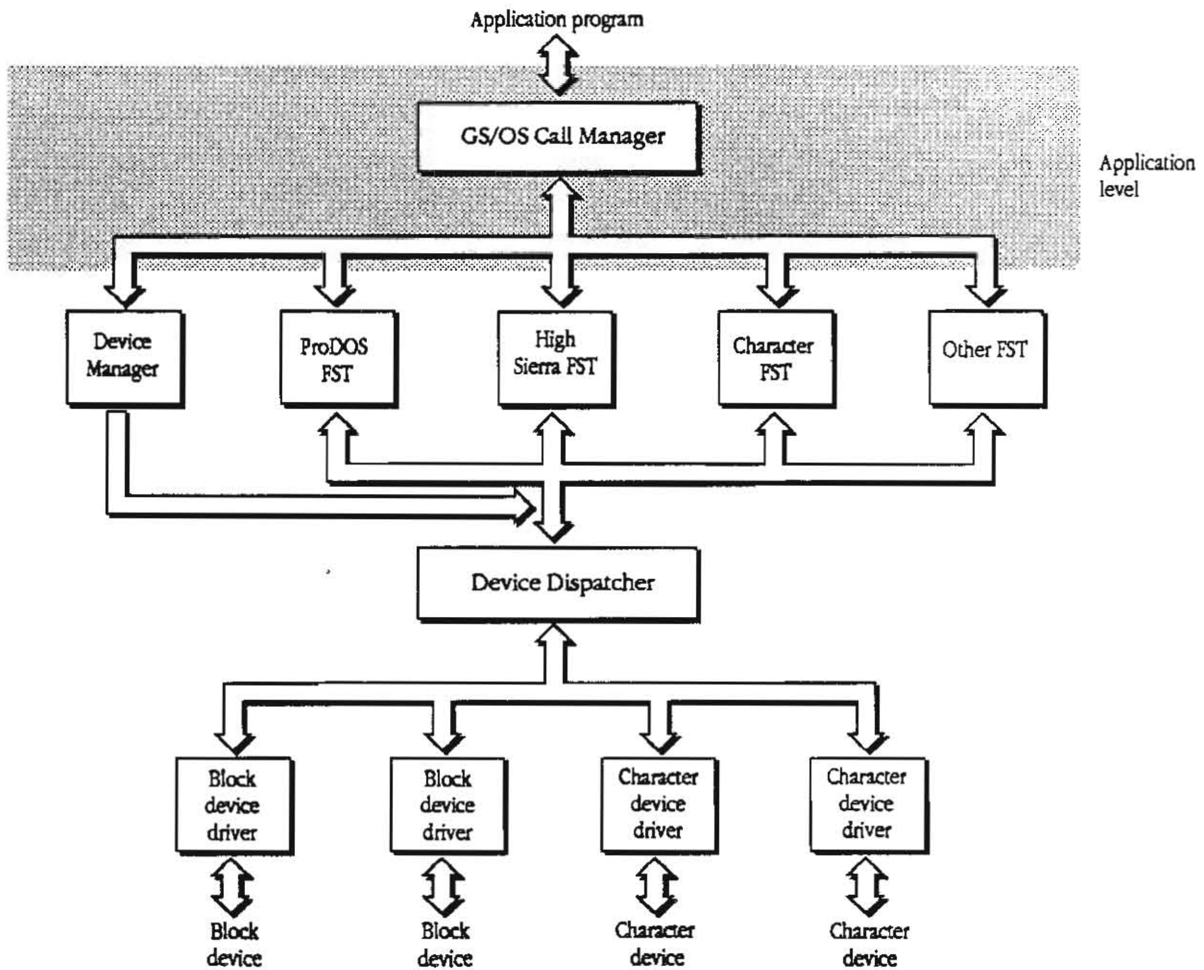
Chapter 1 **The GS/OS Abstract File System**

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.

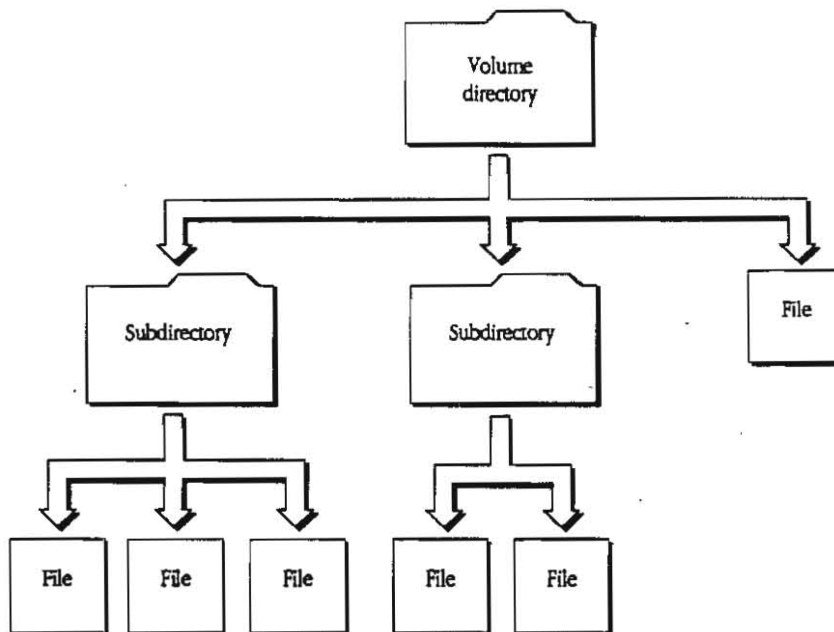
Figure 1-1 Application level in GS/OS



GS/OS can keep your application from dealing with FSTs and devices at all, and thus allow you to take a higher-level approach, by supporting files in a hierarchical file system.

In a **hierarchical file system**, some files, called directory files, can contain the names of either files or other directories. Those directories can in turn contain the names of files or other directories. Figure 1-2 shows the relationships among files in a hierarchical file system.

Figure 1-2 Example of a hierarchical file structure



In GS/OS, the root-level directory is called a **volume directory.** A volume is a logical entity that allows you to access the files contained on a physical storage medium Only block devices can be identified by volume name, and then only if the named volume is mounted. For example, an entire disk is identified by its **volume name**, which is the filename of its volume directory.

GS/OS also makes certain assumptions about what each file in this hierarchical file system looks like. The assumptions are as follows:

- that each file can be classified as a directory, standard, or extended file
- that each file has a name in a certain format
- that the logical location of each file can uniquely identified by a pathname, which is a collection of the filenames that lead to it

- that each file has access privileges
- that each file has a filetype and an auxiliary file type
- that each file has a creation and modification date and time

The following sections define these assumptions.

Classes of GS/OS files

Every GS/OS file is a collection of bytes on a device.

The three classes of files are as follows:

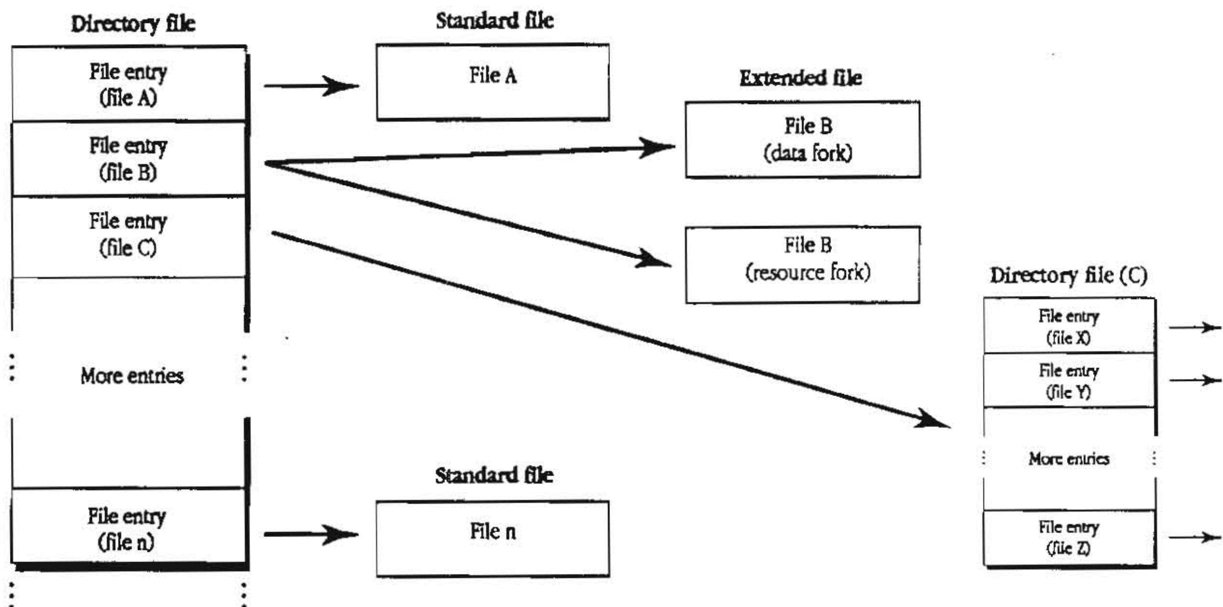
- **directory files**, which store information about other files
- **standard files**, which are a collection of a single sequence of data
- **extended files**, which are a collection of two sequences of data

Note These classes of files are for block devices. GS/OS also allows you to treat character devices as if they contained files. See Chapter 11 "The Character FST" for more information.

Directory files

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

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

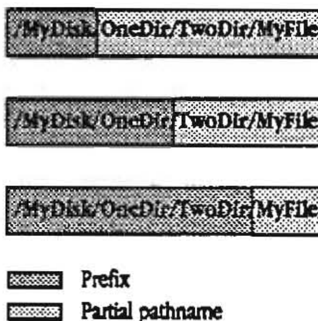
`a/b/c/` must start with a volume or device name

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/").

If you supply a partial pathname that doesn't contain a prefix designator to GS/OS, GS/OS automatically creates the full pathname by adding the prefix designator 0/ in front of the partial pathname. GS/OS determines the separator for a partial pathname in the same way that it determines the separator for a full pathname.

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

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

- **Short prefixes**, referred to by the prefix designators "*" and "0" through "7," cannot be longer than 63 characters. Short prefixes are identical to the prefixes supported by ProDOS 16.
- **Long prefixes**, referred to by the prefix designators "8" through "31," can contain up to about 8,000 characters.

This means that GS/OS allows you to set 32 prefixes. You set and read prefixes using the standard GS/OS calls SetPrefix and GetPrefix. GetPrefix returns a string in which all separators are colons (ASCII $3A) and alphabetic characters have the same case in which they were entered by way of a SetPrefix call.

Predefined prefix designators

For programming convenience, some prefix designators have predefined values. One has a fixed value, and the others have default values assigned by GS/OS at application launch (see Tables 2-4 through 2-6 in Chapter 2). The most important predefined prefix designators are as follows:

*/ the boot prefix—it is the name of the volume from which the presently running GS/OS was booted.

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

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

2/ the system run-time library prefix—it is the pathname of the subdirectory (on the boot volume) that contains the run-time library files used by applications. Run-time libraries are described in Volume 2.

Your application can assign the rest of the prefixes. In fact, once your application is running, it can also change the value of any prefix (including 0/, 1/, or 2/) except prefix */.

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

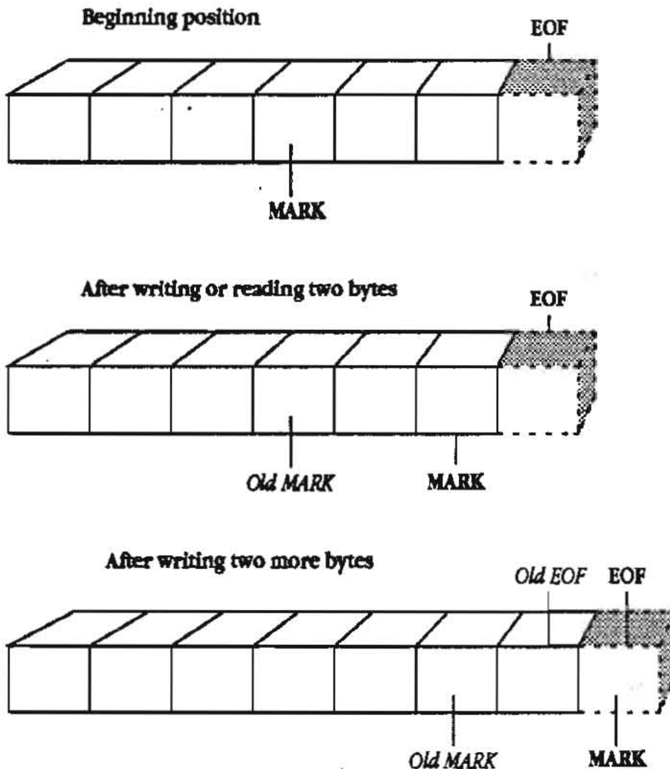
To aid reading from and writing to files, each open standard file and each fork of an open extended file has a byte count indicating the size of the file in bytes (EOF), and another defining the current position in the file (the mark). GS/OS moves both EOF and mark automatically when data is added to 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.

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.

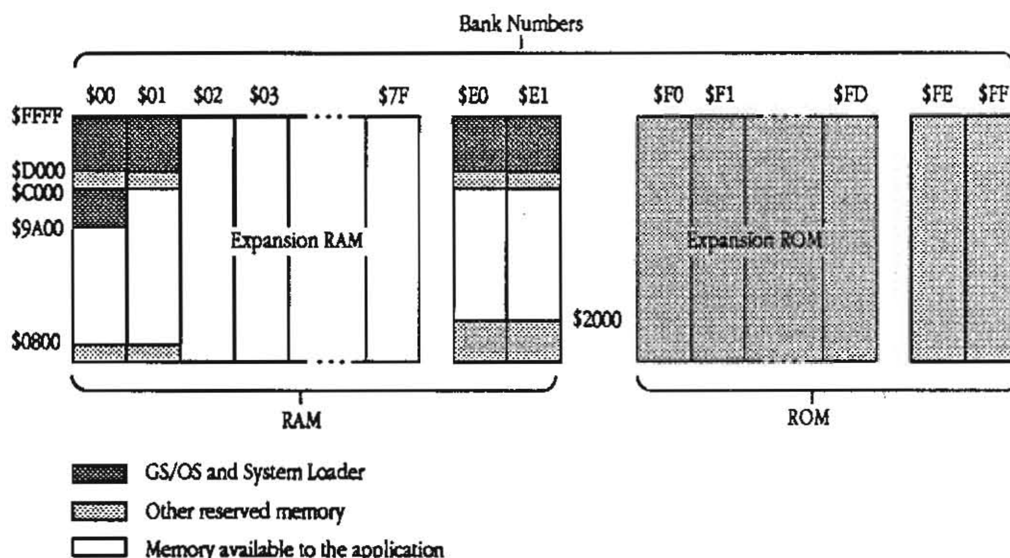
Chapter 2 **GS/OS and Its Environment**

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.

Memory allocatable to applications through the Memory Manager is in bank $00, at locations $0800–$9A00, bank $01 at $0800-$C000, and banks $02–$7F at locations $0000-$FFFF (all 64 KB) in each bank.

For example, a 1 MB Apple IIGS Memory Expansion Card makes available 16 additional banks of memory numbered $02 to $11.

Under most circumstances, you should simply request memory from the Memory Manager and not use fixed locations. The only fixed locations you need to use are listed in the next sections.

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

Entry points and fixed locations

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

The supported GS/OS entry points are $E100A8 and E100B0. These locations are the entry points for all GS/OS calls. The Tool Locator entry point is $E10000, which is the entry point for all Apple IIGS tool calls, including the System Loader (described in Chapter 2).

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

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

Table 2-1 GS/OS vector space

Address	Description
$E10000	Entry vector for all Apple IIGS tool calls.
$E100A8 - $E100AB	Entry vector for inline GS/OS system calls
$E100AC - $E100AF	Reserved for internal use
$E100B0 - $E100B3	Entry vector for stack-based GS/OS system calls
$E100B4 - $E100B9	Reserved for internal use
$E100BA - $E100BB	Two null bytes (guaranteed to be zeros)
$E100BC	OS_KIND byte—indicates currently running operating system, as follows:
	$00 - ProDOS 8
	$01 - GS/OS
	$FF - none; operating system is being loaded or switched
$E100BD	OS_BOOT byte—indicates the operating system that was initially booted, as follows:
	$00 - ProDOS 8
	$01 - GS/OS
$E100BE - $E100BF	$0000 - GS/OS is not busy
	$8000-GS/OS is busy processing a system call

Managing application memory

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

The Memory Manager keeps track of how much memory is free and what parts are allocated to whom. Memory is allocated in blocks of arbitrary length; each block possesses several attributes that describe how the Memory Manager can modify it (such as moving it or deleting it), and how it must be placed in memory (for example, at a fixed address). See the chapter on the Memory Manager in the *Apple IIGS Toolbox Reference* for more information.

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

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

Obtaining application memory

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

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

Accessing data in a movable memory block

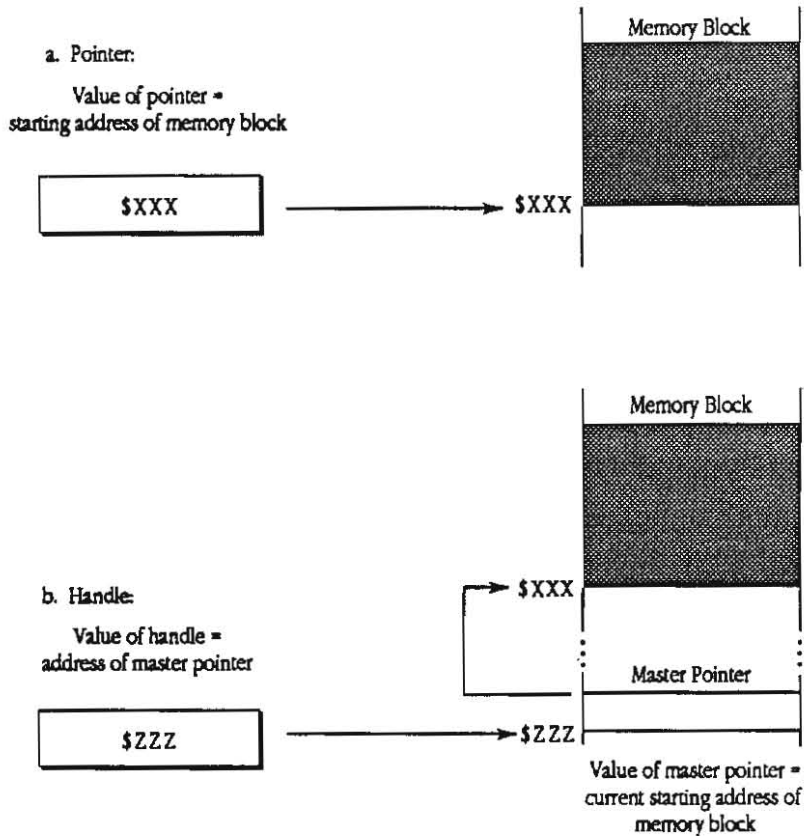
To access data in a movable block, an application cannot use a simple pointer because the Memory Manager may move the block and change the data's address. Instead, each time the Memory Manager allocates a memory block, it returns to the requesting application a handle referencing that block.

A handle is a pointer to a pointer; it is the address of a fixed (nonmovable) location, called the master pointer, that contains the address of the block. If the Memory Manager changes the location of the block, it updates the address in the master pointer; the value of the handle itself is not changed. Thus the application can continue to access the block using the handle, no matter how often the block is moved in memory. Figure 2-2 illustrates the difference between a pointer and a handle.

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

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

Figure 2-2 Pointers and handles



Allocating stack and direct page

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 the next application to launch

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

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

Specifying a GS/OS application to launch

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

Specifying a ProDOS 8 application to launch

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

The GS/OS program dispatcher then takes the following steps:

1. Shuts down GS/OS and the System Loader.
2. Allocates all special memory for the application.
3. Loads and starts up ProDOS 8.

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

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

Specifying whether control should return to your application

The **quit return stack** is a stack of user IDs used to restart applications that have previously quit. If an application specifies a TRUE quit return flag 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.

Machine state at ProDOS 8 application launch

When a ProDOS 8 program is launched, the machine state is as shown in Table 2-3.

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

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

Pathname prefixes at GS/OS application launch

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

Tables 2-4 through 2-6 show the initial values of the prefix numbers that a GS/OS application receives, under the three different launching conditions possible on the Apple IIGS.

Note: In each of the following cases, prefix 1 and prefix 9 are both set to the full pathname of the directory containing the current application. If the string is greater than 64 characters long, prefix 1 is set to a null string and prefix 9 contains the full string.

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.

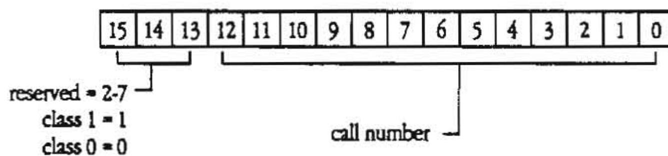
To make a GS/OS assembly language call, your application must provide

- 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.

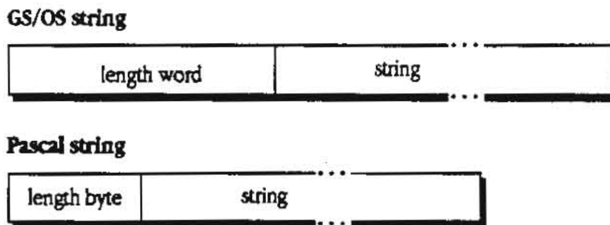
Pointers in the parameter block must be written with the low-order byte of the low-order word at the lowest address.

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



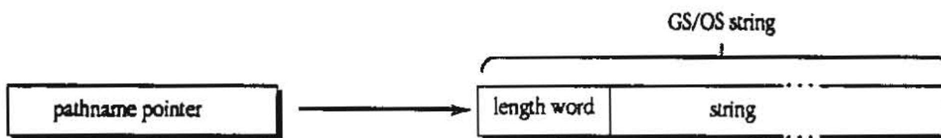
String parameters consist of a pointer parameter in the call's parameter block that points to a data structure containing the string. For standard GS/OS calls, that data structure varies depending on whether the string parameter is an input to or output from the call.

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.

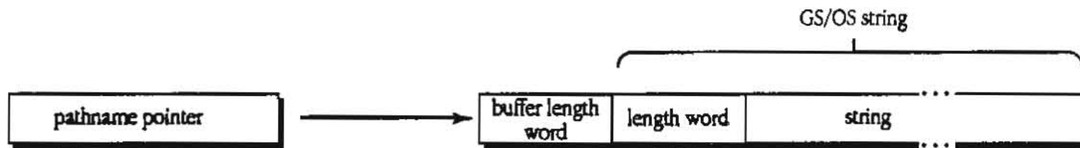
Figure 3-2 GS/OS input string structure



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

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

`pCount` Word input value: 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

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

`pCount` Word input value: 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

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

`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

Offset	Field	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 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.

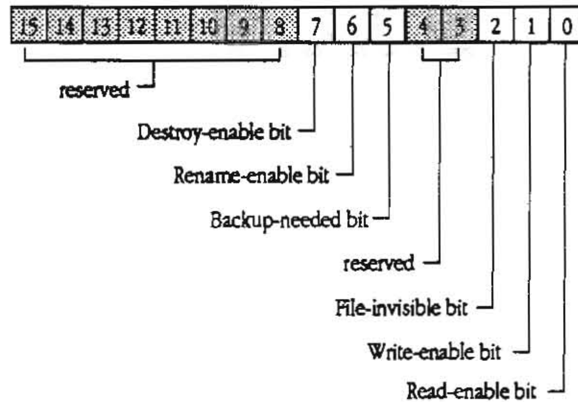
Parameters

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

`pCount` Word input value: 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

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

`pCount` Word input value: 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

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

`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

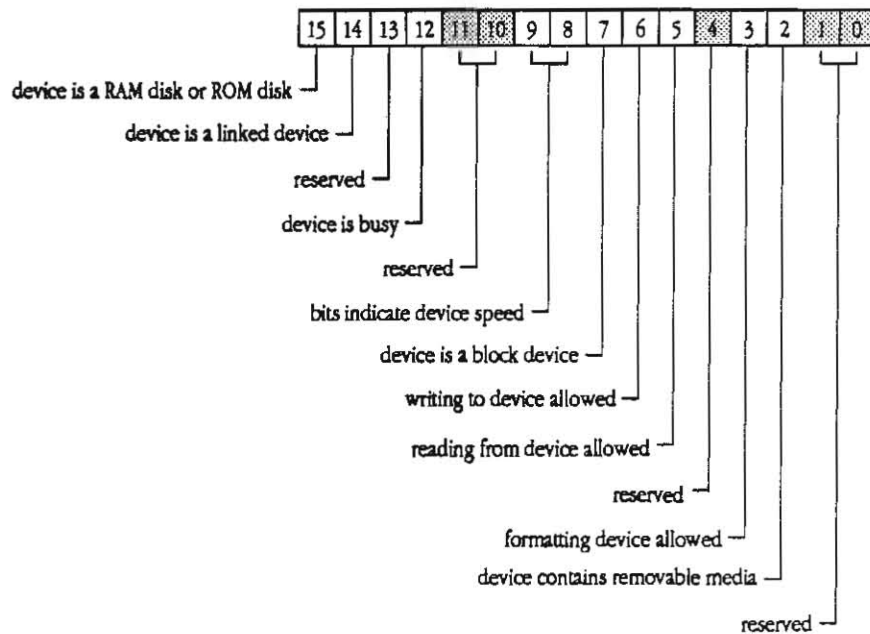
Offset	Field	No.	Size and type
$00	pCount	—	Word INPUT value (minimum =2)
$02	devNum	1	Word INPUT value
$04	devName	2	Longword INPUT pointer
$08	characteristics	3	Word RESULT value
$0A	totalBlocks	4	Longword RESULT value
$0E	slotNum	5	Word RESULT value
$10	unitNum	6	Word RESULT value
$12	version	7	Word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

`pCount` Word input value: 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:

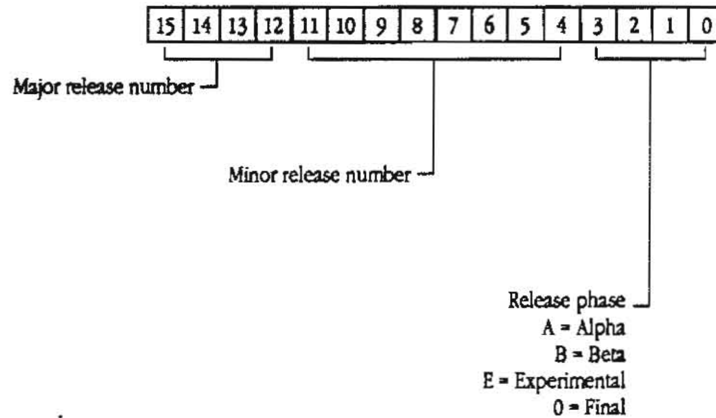


`totalBlocks` Longword result value: If the device is a block device, this parameter gives the maximum number of blocks on volumes handled by the device. For character devices, this parameter returns zero.

`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
		$0013	Hard disk (generic)
$0001	Profile 5 MB	$0014	Floppy disk (generic)
$0002	Profile 10 MB	$0015	Tape drive (generic)
$0003	Apple 3.5 Drive (includes UniDisk 3.5 Drive)	$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

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

`pCount` Word input value: 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

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

`pCount` Word input value: 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

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

`pCount` Word input value: 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

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

`pCount` Word input value: 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

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

`pCount` Word input value: 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

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

`pCount` Word input value: 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

$2028 GetBootVol

Description

Returns the volume name of the volume from which the file GS/OS was last loaded and executed. The volume name returned by this call is equivalent to the prefix specified by */.

Parameters

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

`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 memory area where a GS/OS output string structure giving the boot volume name is to be returned.

Errors

$4F buffer too small

$2020 GetDevNumber

Description

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

GS/OS assigns device numbers at boot time. The numbers are a series of consecutive integers beginning with 1. There is no algorithm for determining the device number for a particular device.

Because a device may hold different volumes and because volumes may be moved from one device to another, the device number returned for a particular volume name may be different at different times.

Parameters

Offset	Field	No.	Size and type
$00	pCount		Word INPUT value (minimum = 2)
$02	devName	1	Longword INPUT pointer
$06	devNum	2	Word RESULT value

`pCount` Word input value: 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

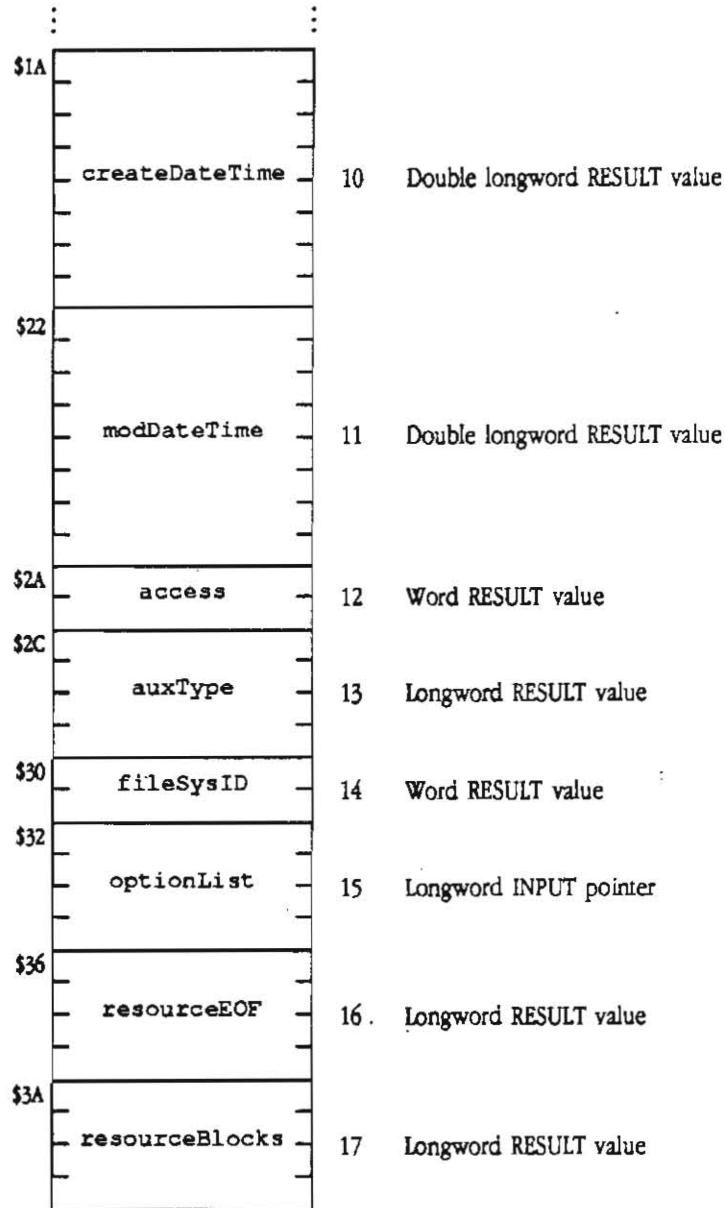
$201C GetDirEntry

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

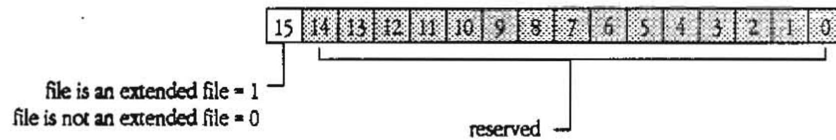
Offset	Field	No.	Size and type
$00	pCount		Word INPUT value (minimum = 5)
$02	refNum	1	Word INPUT value
$04	flags	2	Word RESULT value
$06	base	3	Word INPUT value
$08	displacement	4	Word INPUT value
$0A	name	5	Longword INPUT pointer
$0E	entryNum	6	Word RESULT value
$10	fileType	7	Word RESULT value
$12	eof	8	Longword RESULT value
$16	blockCount	9	Longword RESULT value
$1A	⋮		



`pCount` Word input value: The number of parameters in this parameter block. Minimum is 5; maximum is 17.

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

`flags` Word result value: Flags that indicate various attributes of the file, as follows:



`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.

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

$2019 GetEOF

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

Parameters

Offset		No.	Size and type
$00	pCount		Word INPUT value (minimum = 2)
$02	refNum	1	Word INPUT value
$04	eof	2	Longword RESULT value

`pCount` Word input value: 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

$2006 GetFileInfo

Description

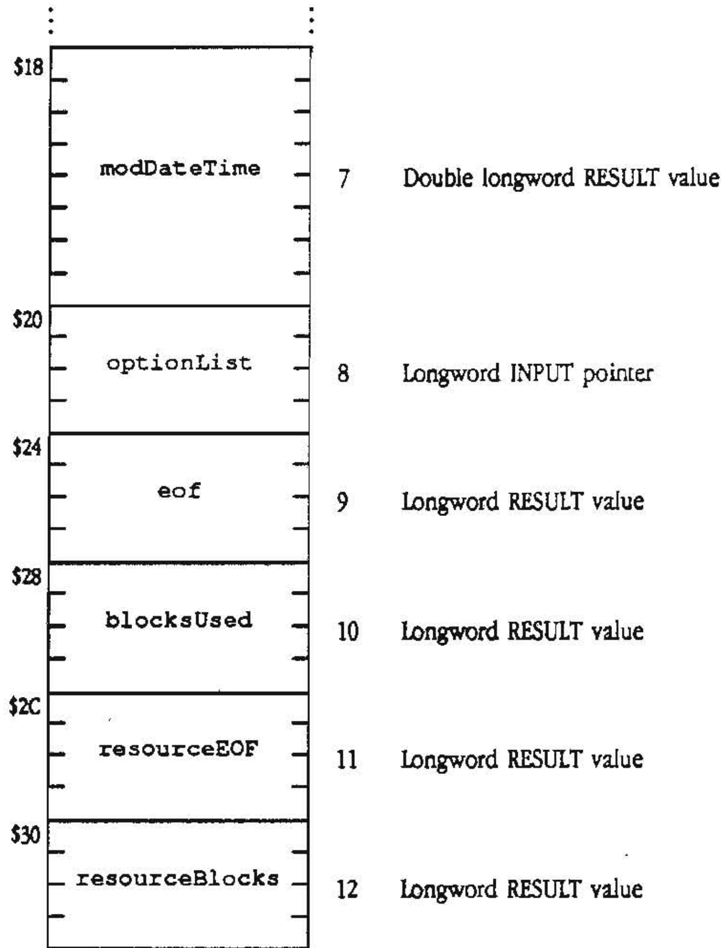
This call returns certain file attributes of an existing open or closed block file.

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

Offset		No.	Size and type
$00	pCount		Word INPUT value (minimum = 2)
$02	pathname	1	Longword INPUT pointer
$06	access	2	Word RESULT value
$08	fileType	3	Word RESULT value
$0A	auxType	4	Longword RESULT value
$0E	storageType	5	Word RESULT value
$10	createDateTime	6	Double longword RESULT value
⋮			



`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.

Errors

$10	device not found
$27	I/O error
$40	invalid pathname syntax
$44	path not found
$45	volume not found
$46	file not found
$4A	version error
$4B	unsupported storage type
$52	unsupported volume type
$53	invalid parameter
$58	not a block device

$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

Offset	Parameter	No.	Size and type
$00	pCount		Word INPUT value (minimum = 2)
$02	fstNum	1	Word INPUT value
$04	fileSysID	2	Word RESULT value
$06	fstName	3	Longword INPUT pointer
$0A	version	4	Word RESULT value
$0C	attributes	5	Word RESULT value
$0E	blockSize	6	Word RESULT value
$10	maxVolSize	7	Longword RESULT value
$14	maxFileSize	8	Longword RESULT value

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

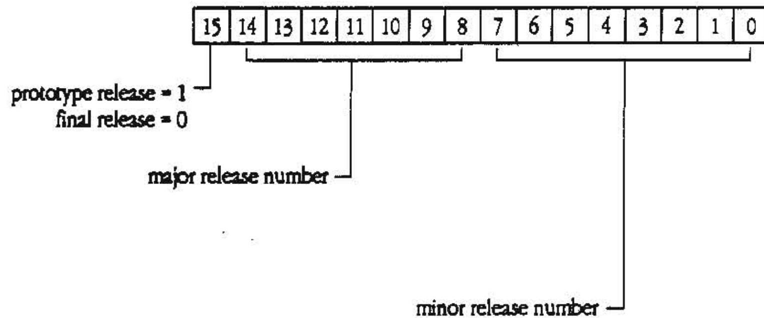
`fstNum` Word input value: An FST number. GS/OS assigns FST numbers in sequence (1, 2, 3, and so on) as it loads the FSTs. There is no fixed correspondence between FSTs and FST numbers. To get information about every FST in the system, one makes repeated calls to GetFSTInfo with `fstNum` values of 1, 2, 3, and so on until GS/OS returns error $53: parameter out of range.

`fileSysID` Word result value: Identifies the file system as follows:

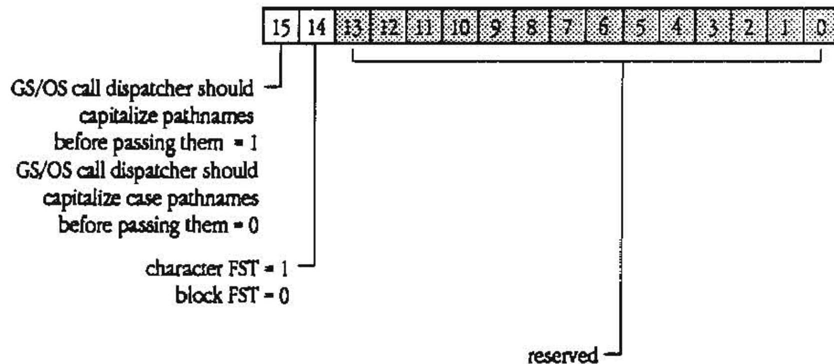
$0000	reserved	$0007	LISA
$0001	ProDOS/SOS	$0008	Apple CP/M
$0002	DOS 3.3	$0009	reserved
$0003	DOS 3.2 or 3.1	$000A	MS/DOS
$0004	Apple II Pascal	$000B	High Sierra
$0005	Macintosh (MFS)	$000C	ISO 9660
$0006	Macintosh (HFS)	$000D–$FFFF	reserved

`fstName` Longword input pointer: Points to a result buffer where GS/OS is to return the name of the FST.

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



`attributes` Word result value: General attributes of the FST, as follows:



`blockSize` Word result value: The block size (in bytes) of blocks handled by the FST.

`maxVolSize` Longword result value: The maximum size (in blocks) of volumes handled by the FST.

`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

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

`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

Offset		No.	Size and type
$00	pCount		Word INPUT value (minimum = 2)
$02	refNum	1	Word INPUT value
$04	position	2	Longword RESULT value

`pCount` Word input value: 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

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

`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

Offset	Field	No.	Size and type
$00	pCount		Word INPUT value (minimum = 2)
$02	prefixNum	1	Word INPUT value
$04	prefix	2	Longword INPUT pointer

`pCount` Word input value: 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

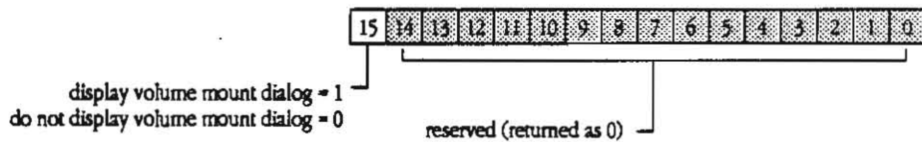
Description This call returns the value of the current global system preferences. The value of system preferences affects the behavior of some system calls. See also the SetSysPrefs call.

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

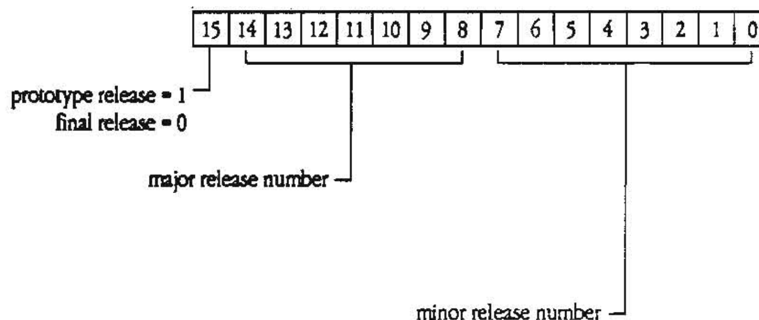
Description This call returns the version number of the GS/OS operating system. This value can be used by application programs to condition version-dependent operations.

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

Offset		No.	Size and type	
$00	pCount	—	Word INPUT value	(minimum = 4)
$02	refNum	1	Word INPUT value	
$04	enableMask	2	Word INPUT value	
$06	numChars	3	Word INPUT value	
$08	newlineTable	4	Longword INPUT pointer	

`pCount` Word input value: 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

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)

$2010 Open

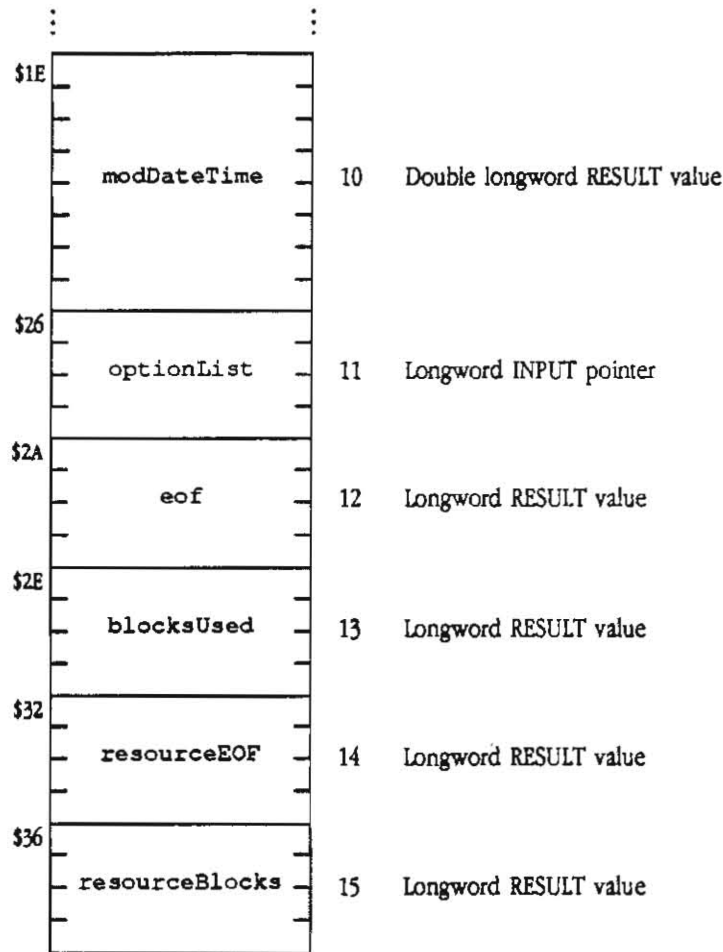
Description

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	Field	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	⋮		

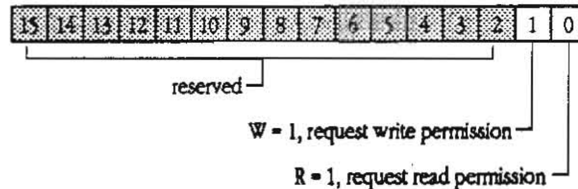


`pCount` Word input value: The number of parameters in this parameter block. Minimum is 2; maximum is 15.

`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.

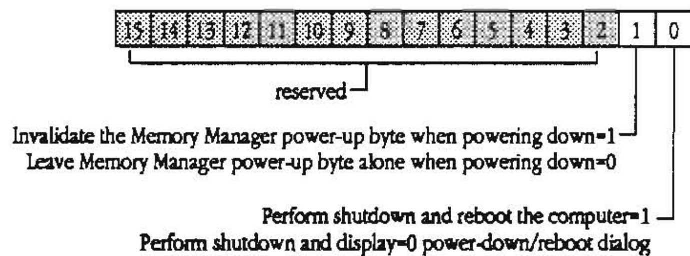
If Bit 1 is cleared to 0, GS/OS leaves the Memory Manager power-up byte alone; this leaves any RAM disks intact while the machine is rebooted. If Bit 1 is set to 1, however, GS/OS invalidates the power-up byte, which effectively destroys any RAM disk, before rebooting the computer.

Parameters

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

`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:



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."

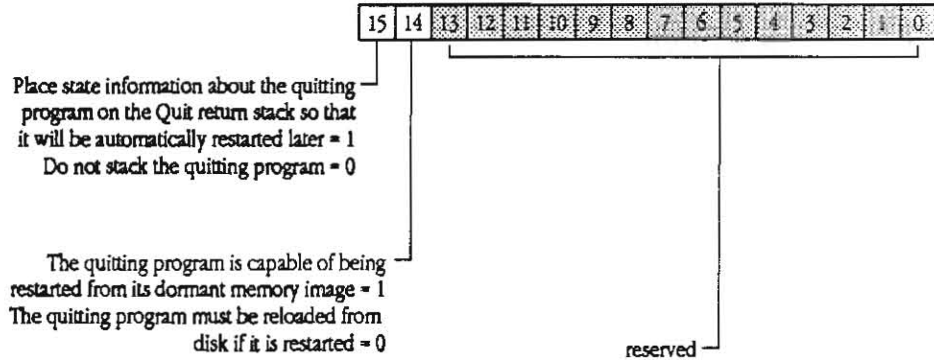
Parameters

Offset	Field	No.	Size and type
$00	pCount	—	Word INPUT value (minimum = 0)
$02	pathname	1	Longword INPUT pointer
$06	flags	2	Word INPUT value

`pCount` Word input value: 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

Offset	Field	No.	Size and type
$00	pCount	—	Word INPUT value (minimum = 4)
$02	refNum	1	Word INPUT value
$04	dataBuffer	2	Longword INPUT pointer
$08	requestCount	3	Longword INPUT value
$0C	transferCount	4	Longword RESULT value
$10	cachePriority	5	Word INPUT value

`pCount` Word input value: 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

Errors

$27	I/O error
$2E	disk switched
$43	invalid reference number
$4C	eof encountered
$4E	access not allowed

$201F SessionStatus

Description

This call returns a value that tells whether or not a write-deferral session is in progress. See also BeginSession and EndSession in this chapter.

Parameters

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

`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

Offset	Field	No.	Size and type
$00	pCount	—	Word INPUT value (minimum = 3)
$02	refNum	1	Word INPUT value
$04	base	2	Word INPUT value
$06	displacement	3	Longword INPUT value

`pCount` Word input value: 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

$2005 SetFileInfo

Description

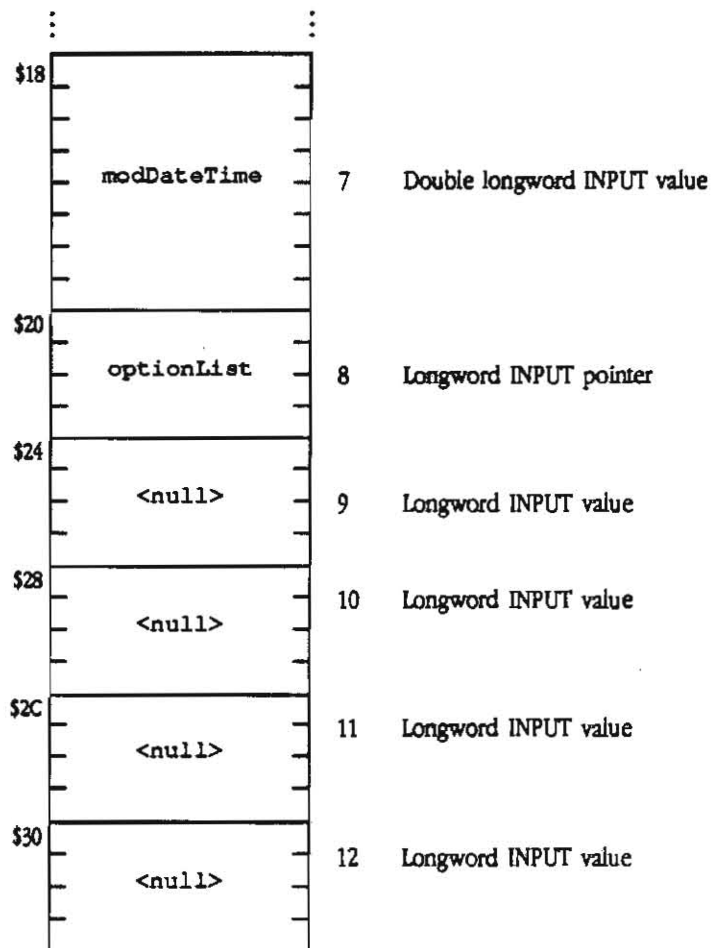
This call sets certain file attributes of an existing open or closed block file. This call immediately modifies the file information in the file's directory entry whether the file is open or closed. It does not affect the file information seen by previously open access paths to the same file.

Important A GetFileInfo call following a SetFileInfo call on an open file may not return the values set by the SetFileInfo call. To guarantee recording of the attributes specified in a SetFileInfo call, you must first close the file.

See also the GetFileInfo call.

Parameters

Offset	Field	No.	Size and type
$00	`pCount`	—	Word INPUT value (minimum = 2)
$02	`pathname`	1	Longword INPUT pointer
$06	`access`	2	Word INPUT value
$08	`fileType`	3	Word INPUT value
$0A	`auxType`	4	Longword RESULT value
$0E	`<null>`	5	Word INPUT value
$10	`createDateTime`	6	Double longword INPUT value
$18	⋮		



`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

$201A SetLevel

Description

This function sets the current value of the system file level.

Whenever a file is opened, GS/OS assigns it a file level equal to the current system file level. A Close call with a reference number of $0000 closes all files with file level values at or above the current system file level. Similarly, a Flush call with reference number of $0000 flushes all files with file level values at or above the current system file level. See also the GetLevel call.

Parameters

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

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

`level` Word input value: The new value of the system file level. Must be in the range $0000-$00FF.

Errors

$59 invalid file level

$2016 SetMark

Description

This call sets the file mark (the position from which the next byte will be read or to which the next byte will be written) to a specified value. The value can never exceed EOF, the current size of the file. See also the GetMark call.

Parameters

Offset	Field	No.	Size and type
$00	pCount	—	Word INPUT value (minimum = 3)
$02	refNum	1	Word INPUT value
$04	base	2	Word INPUT value
$06	displacement	3	Longword INPUT value

`pCount` Word input value: 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

Offset	Field	No.	Size and type
$00	pCount	—	Word INPUT value (minimum = 2)
$02	prefixNum	1	Word INPUT value
$04	prefix	2	Longword INPUT pointer

`pCount` Word input value: 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

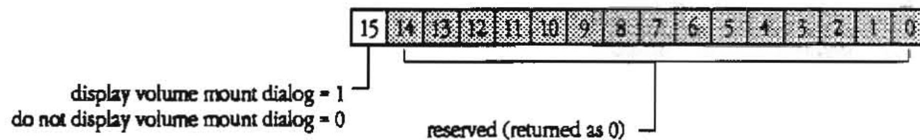
Description This call sets the value of the global system preferences. The value of system preferences affects the behavior of some system calls. See also the GetSysPrefs call.

Parameters

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

`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.

- For those calls that specify a pathname parameter, the Mount facility displays the dialog only if system preference bit 15 is 1. Otherwise, Mount returns the CANCEL return code which normally causes the system to return a volume-not-found error. Thus, an application can be written to either handle volume-not-found errors itself (system-preference bit 15 = 0) or to allow the system to automatically display mount dialogs (bit 15 = 1), except for the situation where the System Loader is attempting to load a dynamic segment.
- For those calls that result in the System Loader attempting to load a dynamic segment, the System Loader always sets the system preference bit (bit 15) to 1, and then resets it to its original value when the segment has been loaded. Thus, the Mount dialog box is always displayed when a dynamic segment is requested.

Errors (none)

$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

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

`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

Offset	Field	No.	Size and type
$00	pCount	—	Word INPUT value (minimum = 2)
$02	devName	1	Longword INPUT pointer
$06	volName	2	Longword INPUT pointer
$0A	totalBlocks	3	Longword RESULT value
$0E	freeBlocks	4	Longword RESULT value
$12	fileSysID	5	Word RESULT value
$14	blockSize	6	Word RESULT value

`pCount` Word input value: 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

Offset	Parameter	No.	Size and type
$00	pCount	—	Word INPUT value (minimum = 4)
$02	refNum	1	Word INPUT value
$04	dataBuffer	2	Longword INPUT pointer
$08	requestCount	3	Longword INPUT value
$0C	transferCount	4	Longword RESULT value
$10	cachePriority	5	Word INPUT value

`pCount` Word input value: 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.

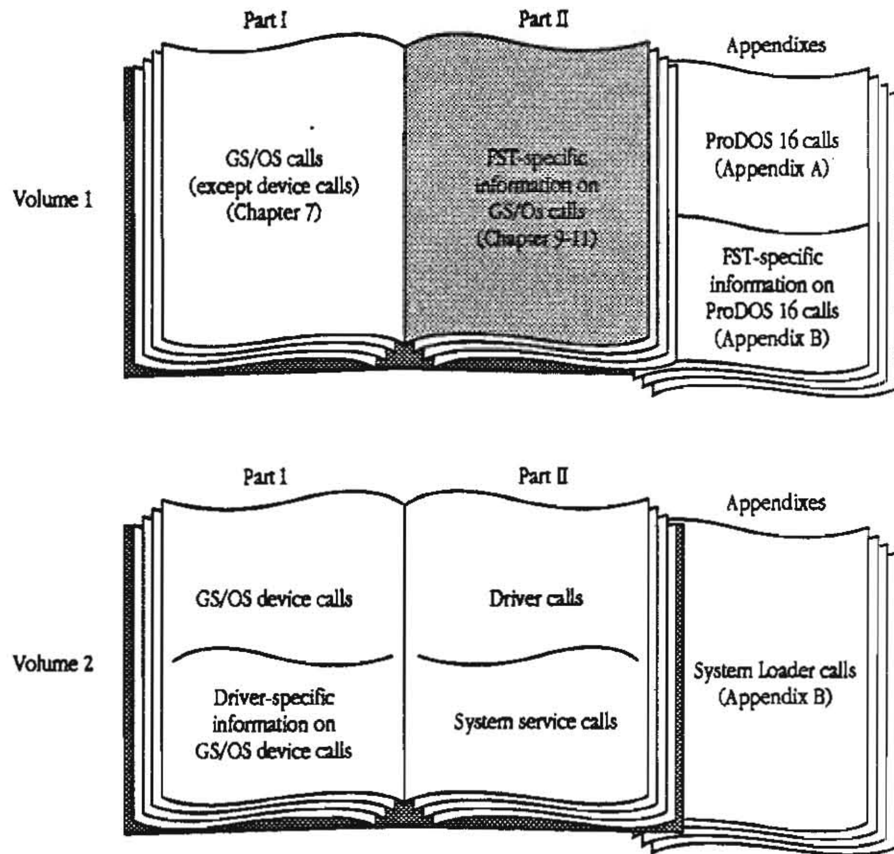
`cachePriority` Word input value: Specifies whether or not disk blocks handled by the call are candidates for caching, as follows:

$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

Part II The File System Level



Chapter 8 **File System Translators**

This chapter describes how GS/OS is able to communicate with many different types of files and devices, in a manner that is transparent to the application. The operating system does this by supporting

- a generic GS/OS file interface (the abstract file system, described in Chapter 1) with which applications communicate
- individual file system translators (FSTs) that act as intermediaries between the GS/OS file interface and specific file systems and devices

This chapter discusses FSTs in general; the following chapters in Part II describe the individual FSTs suppplied with GS/OS.

Note: The file system translators in GS/OS handle both standard GS/OS (class 1) calls and ProDOS 16-compatible (class 0) calls. Only the standard GS/OS calls are described in this chapter and the rest of Part II; for information on how FSTs handle ProDOS 16-style calls, see Appendix B of this volume.

The FST Concept

Every file system, such as ProDOS or Macintosh HFS, stores directories, subdirectories, files, and possibly other data structures on disk volumes in a format unique to that file system. Furthermore, each file system provides a slightly different set of system calls for accessing its files. The uniqueness of these data structures and system calls makes it very difficult for an application program that uses one file system to also access a volume created under another file system. Thus, application programs are nearly always written to run with one particular file system.

A **file system translator** (FST) is a GS/OS software module that accepts GS/OS calls made by applications and translates those calls into a form acceptable to the particular file system the FST supports. Likewise, the FST takes data read from the device and converts it to a form consistent with the generic GS/OS file interface (the abstract file system, described in Chapter 1). This makes it possible to write an application in which the same set of file I/O calls can access files on volumes created by any file system for which there is an FST. Application programs can thus transparently access files from any file system, using standard GS/OS system calls.

Note: FSTs provide only the file access capabilities of GS/OS (see Chapter 4), which are similar to those of ProDOS 16. Because all FSTs use the same standard set of calls, they cannot implement all access capabilities and all calls for all file systems. Moreover, some FSTs cannot even support all of the capabilities provided by GS/OS. The High Sierra FST, for example, does not permit calls that write to disk.

Figure 8-1 The file system level in GS/OS

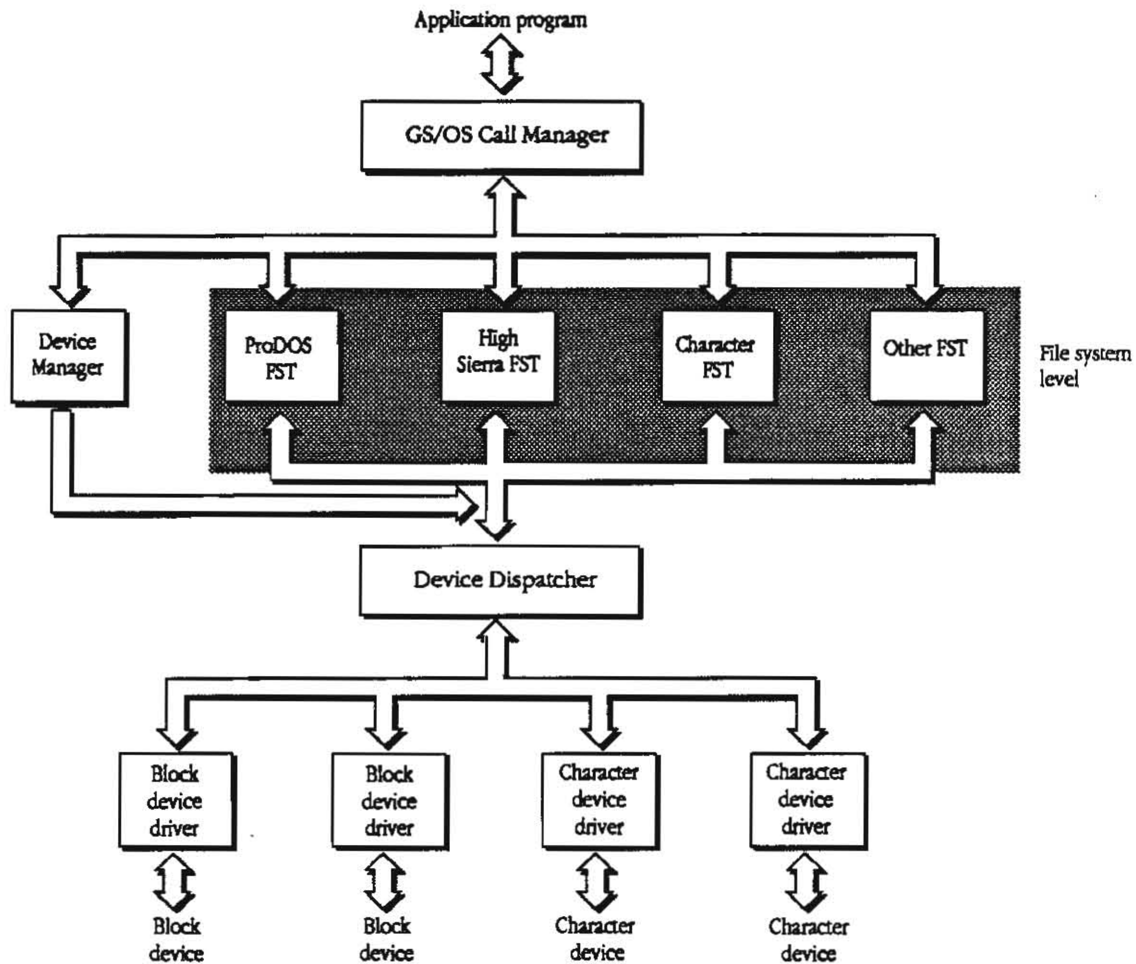


Figure 8-1 shows the conceptual position of FSTs in the GS/OS hierarchy. They make up the file system level, which mediates between the GS/OS call dispatcher at the application level and individual device drivers at the device level. When an FST receives a call, the call has been preprocessed by the GS/OS call dispatcher. The FST either processes the call and returns successfully or encounters an error condition and returns unsuccessfully with an error code. FSTs call the Device Dispatcher, which performs the actual I/O with calls to the device drivers. In addition, FSTs depend on various services provided by the call manager, such as pathname prefix management and error handling.

To GS/OS, all FSTs are equal. Any FST can be removed from the system by the user, and any FST can be added. The user adds or removes FSTs from GS/OS by moving FST files into or out of the subdirectory SYSTEM/FSTS on the boot disk. See Appendix D.

Calls handled by FSTs

GS/OS calls can be classified by the part of the operating system that handles them. File calls are handled by FSTs, device calls are handled by the the Device Manager, and other calls are handled by the GS/OS call manager itself. Table 8-1 lists all the GS/OS calls handled by FSTs.

Table 8-1 GS/OS calls handled by FSTs

Call no.	Call name	Call no.	Call name
$2001	Create	$2015	Flush
$2002	Destroy	$2016	SetMark
$2004	ChangePath	$2017	GetMark
$2005	SetFileInfo	$2018	SetEOF
$2006	GetFileInfo	$2019	GetEOF
$2008	Volume	$201C	GetDirEntry
$200B	ClearBackupBit	$2020	GetDevNum
$2010	Open	$2024	Format
$2012	Read	$2025	EraseDisk
$2013	Write	$2033	FSTSpecific
$2014	Close		

As an application writer, you can expect that every FST will in some way support each of the calls listed in Table 8-1. Depending on the file system accessed, the call may be meaningful, it may do nothing and return no error, or it may do nothing and return an error. See the description of each FST for details.

All of the calls listed in table 8-1 are described in Chapter 7 of this volume, except for FSTSpecific. **FSTSpecific** is a call whose function is completely definable by each FST. For example, the High Sierra FST (see Chapter 10) uses the call to control file type emulation. FSTSpecific is documented individually for each FST that uses it, in the chapter that describes the FST.

Programming for multiple file systems

When you first write an application for GS/OS, it may seem strange not to know what file system your own application's files will be stored on. In reality, it makes your job simpler, but you may have to be careful in the beginning to avoid making some common incorrect assumptions.

Don't assume file characteristics

File-system independence is a cornerstone of the GS/OS design. To be most useful and efficient, and to avoid file-system-specific problems, your application should also be as file-system independent as possible.

In general, you will be working with file information in the format returned by the GS/OS call GetFileInfo, rather than in the format of any real file system. For example, don't assume file-typing conventions other than the file type/auxiliary type provided in the GS/OS abstract file system; it is the job of each FST to translate that information into the file-type format for each file system.

Remember that different file systems use different block sizes. Don't simply assume that a block is 512 (or 256, or 520, or 1024) bytes; if you need to know the exact size of a block on a volume, use the GS/OS Volume call to the device holding that volume.

In manipulating filenames and pathnames, don't assume any fixed limit on name length, and don't assume other restrictions such as a limited ASCII character set. Always allow for the GS/OS pathname syntax: both colons and slashes are valid separators, and colons can *only* be separators. Detailed filename and pathname rules are presented in Chapter 1 of this volume.

In general, go through the GS/OS file system level (by making standard GS/OS calls) as much as possible, rather than performing file-system-specific or device-specific operations which may require the presence of a particular FST, device driver, or device. Use GS/OS's file-system independence and device independence to your own advantage.

Use GetDirEntry

If your program needs to catalog a volume, don't read directory files directly—that is, don't use the Read call to find out what is in a directory. GetDirEntry gives you the information in a standard format for all file systems, whereas with Read you need to know the exact format of a directory file for the specific file system you are accessing. And, because the files of interest may be in any of a number of file systems, it is far simpler to use GetDirEntry and let GS/OS take care of the details for you.

Keep rebuilding your device list

Some applications construct a list of online devices only when they start up. This works fine if the list never changes, but under GS/OS new devices can be added dynamically during execution. Therefore, instead of constructing your own device list, scan the device list each time you need to use it. For example, use repeated DInfo or Volume calls with consecutive device numbers, until an error is returned (such as invalid device number) signals that there are no more on-line devices.

Handle errors properly

Your application's normal error-handling routines may be adequate for processing errors under GS/OS, as long as you remember to always check for errors. A typical file-system-specific error might occur, for example, from attempting to save a file from a file system that normally allows saving, such as ProDOS, to a High Sierra disc. As long as your program is prepared to receive and act on any file error that GS/OS can generate, there should be no problem. Remember also that, because different file systems have different size limits on parameters, error $53 (parameter out of range) might be a very common occurrence.

On the other hand, you may needlessly restrict your application's capabilities if you assume an error will occur when it may not. For example, if your program is written assuming a read-only file system, it may unnecessarily prevent a user from saving a file to a different file system that is not read-only. In general, it is probably better to let GS/OS decide what file permissions and file calls are appropriate and then act on the returned errors if necessary.

Furthermore, what you do when an error occurs can be significant. For example, if a user attempts to save a very large file to a volume whose file system does not support the size of that file, your application should put up a Standard File dialog box to let the user save the data to another file system, rather than simply abort the save and lose the data.

Remember also that GS/OS allows access to character devices with file calls. Therefore, calls such as Read or SetMark may be applied to devices (like a printer) for which they have no meaning. Thus your error-handling should allow for not only different file systems, but completely different devices as well. In fact, it is common for character devices to return status information with error codes; if your file-access routines do not check for typical character-device errors, you may lose critical information.

FSTs and file-access optimization

The file system translators written for GS/OS are designed to make file reads and writes as fast and efficient as possible. You may be able to read a file under GS/OS faster than you can under the file's native operating system. Furthermore, the disk caching available under GS/OS (see Chapter 11 of Volume 2) makes reading faster still.

As much as possible, consecutive file blocks are written to consecutive sectors on disk for fast access. More importantly, though, FSTs are optimized for large, multi-block transfers; for the application writer, this means that it is best to read and write data in chunks as large as possible. If you are interested in speed, try also to avoid Newline read mode (which forces every character to be examined in turn) and the Flush call (which is slowed by the careful checking and updating it must perform).

For the fastest possible multiblock copying, use the GS/OS call BeginSession to temporarily defer block writes while copying, and then EndSession to flush the cache when you are done copying. BeginSession and EndSession are most useful when doing multiple-file copies, because directory blocks are not written to disk as every file is copied. See the descriptions of BeginSession, EndSession and SessionStatus in Chapter 7.

Present and future FSTs

GS/OS applications can read files from any file system for which there is an existing, installed GS/OS file system translator. Currently, Apple defines the following file systems, each specified by its own file system ID. This, then, is the total list of potential FSTs:

File system ID	Description	File system ID	Description
$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

Also, as new file systems are defined, Apple assigns them unique file system IDs. In theory, then, all of the above file systems (and any future systems) can be accessed through GS/OS once FSTs are written for them. In practice, Apple will create new FSTs as dictated by demand and time constraints. The currently existing FSTs are described individually in subsequent chapters. Future releases of GS/OS will include file system translators for other file systems.

Disk initialization and FSTs

Disk initialization is a complex issue under an operating system that supports multiple file formats and many different types of devices.

For example, a system could be configured with several FSTs. A user might wish to write any one of the file formats on a 3.5-inch disk or a 5.25-inch disk. Or, if a single 3.5-inch drive supports multiple low-level formatting styles, a formatting routine might select different encoding schemes for different file systems.

The Initialization Manager is a GS/OS routine that puts a dialog box on the screen, allowing the user to select among valid formatting choices (given the current system configuration of FSTs and device drivers). Once the user has made a selection, the appropriate FST then performs the format call and writes the new file system.

Your application can use either the of the GS/OS calls Format or EraseDisk to initialize disks. The format call physically formats the disk and writes out the file system; the EraseDisk call simply writes out a new directory without formatting the disk. Either call causes the initialization dialog box to appear; after the user makes the desired choices, the appropriate FST proceeds with the formatting. For both calls, the return parameter `fileSysID` indicates which file system (if any) the user chose. Format and EraseDisk are described in more detail in Chapter 7 of this volume.

Use of the Initialization Manager adds a user dialog to the initialization process. Because the Initialization Manager dialog box allows the user to cancel, it is probably not necessary for your application also to make the user confirm that a format or erasure is desired.

Chapter 9 **The ProDOS FST**

The ProDOS file system translator (ProDOS FST) provides a transparent application interface to the ProDOS file system. The ProDOS FST can access any block device whose GS/OS device driver can perform 512-byte block reads and writes.

Note: This chapter describes only standard GS/OS (class 1) calls; for descriptions of how the ProDOS FST handles equivalent ProDOS 16 (class 0) calls, see Appendix B.

The ProDOS file system

The ProDOS file system is the native file system for most of the Apple II family of computers. All applications that run under either ProDOS 8 or ProDOS 16 create and read ProDOS files (if they create files at all).

The ProDOS file system is characterized by a hierarchical structure, 512-byte logical blocks, 16 MB maximum file size and 32 MB maximum volume size. ProDOS files are either standard (sequential) files or directory files; there are no random-access, record-based file types recognized as such by ProDOS.

ProDOS filenames can be up to 15 characters long, consisting of the numerals 0–9, the uppercase letters A–Z, and the period (.), in any combination (except that the first character must be a letter). A ProDOS volume name is like a filename but is preceded by a slash (/) or a colon. A ProDOS pathname consists of a sequence of slash-separated filenames, starting with a volume name.

The ProDOS file system is described in the *ProDOS 8 Technical Reference Manual* and the *Apple IIgs ProDOS 16 Reference.*

GS/OS and the ProDOS FST

The GS/OS abstract file system described in Chapter 1 is closely related to the ProDOS file system. Therefore, the ProDOS file system duplicates many features of the abstract file system exactly, and many GS/OS calls to the ProDOS FST behave exactly as described in Chapter 1. Here are the principal differences:

- ProDOS 8 and ProDOS 16 do not create or recognize extended files, equivalent to the resource forks of Macintosh files. However, the ProDOS FST under GS/OS can store and retrieve extended files in ProDOS format, by defining a new storage type ($0005). When a file is stored in this format, a GS/OS application can retrieve its resource fork and its data fork; applications under ProDOS 8 and ProDOS 16, however, cannot access the file at all; attempts to open the file result in error $4B (unsupported storage type).
- Under GS/OS, a ProDOS pathname can have either slashes (/) or colons (:) as filename separators. The GS/OS Call Manager converts both types of separators to an internal format before passing on the pathname to the ProDOS FST.

- Because ProDOS files and volumes have maximum sizes smaller than those supported by GS/OS, parameters related to size (such as `EOF`, `position`, `blockCount`, `requestCount`, and `transferCount`) may not be accepted by the ProDOS FST if they are too large. In such cases the ProDOS FST returns error $53 (parameter out of range).
- Because several file-entry fields in ProDOS directories on disk are smaller than their equivalent parameters in the GS/OS calls that access file entries, the high-order parts of some of those parameters are always zero when a file entry is read, and must also be zero when a file entry is stored. See the individual call descriptions under "Calls to the ProDOS FST."

Calls to the ProDOS FST

This section describes how the ProDOS FST handles certain GS/OS calls differently from the general procedures described in Chapter 7. Calls not listed in this section are handled exactly as described in Chapter 7.

GetDirEntry ($201C)

GetDirEntry returns file information contained in a volume directory or subdirectory entry. Under the ProDOS FST, the following fields have limitations different from the general values permitted by GS/OS:

`fileType`	Only the low-order byte contains information.
`EOF`	Only the three low-order bytes contain information.
`blockCount`	Only the two low-order bytes contain information.
`auxType`	Only the two low-order bytes contain information.
`optionList`	Not used.
`resourceEOF`	Only the three low-order bytes contain information.
`resourceBlockCount`	Only the two low-order bytes contain information.

GetFileInfo ($2006)

GetFileInfo returns certain file attributes for an existing block file. Under the ProDOS FST, the following fields have limitations different from the general values permitted by GS/OS:

`fileType`	Only the low-order byte contains information.
`auxType`	Only the two low-order bytes contain information.
`storageType`	Only the low nibble of the low byte contains information.
`optionList`	Not used.
`EOF`	Only the three low-order bytes contain information.
`blocksUsed`	Only the two low-order bytes contain information.

SetFileInfo ($2005)

SetFileInfo assigns certain file attributes to an existing block file. Under the ProDOS FST, the following fields have limitations different from the general values permitted by GS/OS:

`fileType`	Only the low-order byte can be nonzero; otherwise, error $53 (parameter out of range) is returned.
`auxType`	Only the two low-order bytes can be nonzero; otherwise, error $53 (parameter out of range) is returned.
`optionList`	Not used.

Chapter 10 **The High Sierra FST**

This chapter describes the GS/OS High Sierra file system translator (High Sierra FST). The High Sierra FST provides transparent application access to compact read-only optical discs (CD-ROM) and other media upon which High Sierra or ISO 9660-formatted files may reside.

The High Sierra and ISO 9660 file formats are not documented here. See the publications listed under "CD-ROM and the High Sierra/ISO 9660 Formats" for more information. For information on the Apple extensions to ISO 9660, see Appendix E.

Note: This chapter describes only standard GS/OS calls; for descriptions of how the High Sierra FST handles equivalent GS/OS ProDOS 16-compatible calls, see Appendix B.

CD-ROM and the High Sierra/ISO 9660 formats

Compact discs provide a new and promising method of information storage and retrieval. Compact discs can hold vast amounts of information on a medium that is durable and inexpensive to manufacture. The information can be played back using existing, well-established technology based on CD music players.

A single CD-ROM disc holds about 550 megabytes of information. This large capacity is CD-ROM's main advantage, but it comes at a price. Compared to magnetic disk drives, CD-ROM players have much slower access times; it can take up to one second to find a byte of information on a CD-ROM disc, compared to less than a tenth of a second on a large-capacity hard disk.

CD-ROM's biggest disadvantage, however, is that—at present—its optical storage technology is read-only. Users can read from a CD, but they cannot write to it (hence the name CD-*ROM*).

The **High Sierra Group format** (named for the location of an ad-hoc committee's original meeting place) and the **ISO 9660 format** (the International Standards Organization's version of High Sierra) are two nearly identical CD-ROM file formats that support the large files a compact disc can hold. They also simultaneously attempt to minimize the penalties caused by slow access. Here are some of the highlights of the formats that are relevant to GS/OS:

- Logical sectors contain 2048 bytes (2 KB) of data. A logical sector can contain 1, 2, or 4 logical blocks.
- Files can contain data in any form or for any purpose; High Sierra/ISO 9660 specifies nothing about file contents.
- File identifiers can consist of three parts: a filename, a filename extension, and a version number. Directories have the filename part only. Nondirectory files under High Sierra need one or more of the three parts (except that a file cannot be identified by a version number alone). Under ISO 9660, a nondirectory file must include all three parts.

  The filename is 0 or more characters (uppercase A–Z, digits 0–9, or underscore); it must be followed by a period. The filename extension is 0 or more characters, and it must be followed by a semicolon. The version number is one to six digits. The sum of the filename and extension must be between 2 and 31 characters, including the period. Under ISO 9660, then, a minimum legal file name is something like "A.;1" or ".A;1".

Note: See the section "Apple Extensions to ISO 9660," later in this chapter, for information on how to devise High Sierra/ISO 9660 filenames that are transformable to other file systems with different conventions.

- High Sierra/ISO 9660 is hierarchical; files may be placed in subdirectories. To speed access to files deep within subdirectories, there is a Path Table that can be loaded into RAM for fast searching; it is an index to all subdirectories on disc. In addition, directory entries are kept small (and therefore fast to search) by putting auxiliary directory information—such as creation dates and access permissions—into extended attribute records (XARs), stored separately.
- Both ISO 9660 and High Sierra support associated files (equivalent to resource forks of GS/OS extended files); however, the High Sierra FST supports associated files for ISO 9660-formatted files only.
- High Sierra/ISO 9660 supports hidden files.

The High Sierra/ISO 9660 format from which Apple's High Sierra FST was designed is defined in these two documents:

- *Working Paper for Information Processing—Volume and File Structure of Compact Read-Only Optical Discs for Information Interchange*, published by the CDROM Ad Hoc Advisory Committee. May 28, 1986. This is the original High Sierra Group proposal.
- *ISO 9660: Information Porcessing—Volume and File Structure of CD-ROM for Information Interchange*, first edition, 1988. This is the ISO 9660 standard, a slightly modified version of the High Sierra Group format.

Non-CD-ROM implementation: Although High Sierra and ISO 9660 were developed specifically for compact disc storage, nothing in either format requires the files to be on CD-ROM. It is possible to have High Sierra/ISO 9660 files on essentially any storage medium that can be formatted to accept them.

Limitations of the High Sierra FST

In translating file calls back and forth between CD-ROM drivers and GS/OS, the High Sierra FST cannot support all aspects of the High Sierra/ISO 9660 file system, nor can it meaningfully implement all GS/OS application calls. The High Sierra FST provided by Apple has the following features:

- It supports associated files (GS/OS extended files) for ISO 9660-formatted discs only.
- It permits only a single volume descriptor—the Standard File Structure Volume Descriptor— per physical volume.

- It does not support multi-volume sets (named and logically linked groups of volumes occupying more than one disc).
- It does not support multi-extent files (files occupying more than one disc).
- It does not support random-access, record-based files; that is, it can read such files as streams of bytes, but it cannot access individual records directly
- It maps the existence bit of the file flags into the invisibility bit of the GS/OS access word.
- It ignores the file permissions field in the extended attribute record.
- It is a read-only implementation.

This last limitation imposes strong restrictions on GS/OS system calls that write data to the disc: those calls always return a write-protect error, after identifying that the file or device requested is present and is in High Sierra or ISO 9660 format.

In accessing files on a CD-ROM disc, remember that, under High Sierra or ISO 9660, block size is not fixed across volumes. If necessary, you can use the GS/OS Volume call to get the block size for a particular volume. Block counts returned by other calls are always in terms of blocks of the size returned by the Volume call.

An **associated file** in ISO 9660 is analogous to the resource fork of a GS/OS extended file. If an ISO 9660 file named MyFile has an associated file, the associated file has these characteristics:

- It is also named MyFile (its file identifier is identical).
- Its associated bit (in the file flags byte of the directory record) is set.
- Its directory entry resides immediately *before* the other MyFile's directory entry.

Thus, GS/OS refers to the first MyFile (whose associated bit is set) as the resource fork of the extended file MyFile, and the immediately following MyFile (whose associated bit is clear) as the data fork of MyFile. Only data files can have associated files; directories cannot.

File types: High Sierra/ISO 9660 does not provide an explicit file typing convention. This can be a problem because many applications select a particular file type as a filter when calling the Standard File Tool Set to display files to the user. In such a case, files from a High Sierra/ISO 9660 disc would never be selectable.

To remedy this problem, the High Sierra FST, through the call FSTSpecific, defines and implements a convention by which High Sierra/ISO 9660 filenames can be used to convey file type information. See the discussion under "FSTSpecific", later in this chapter. In addition, Apple has defined a protocol that extends ISO 9660 to store file-type and other information needed by either ProDOS or Macintosh HFS files. See the next section "Apple Extensions to ISO 9660".

Apple extensions to ISO 9660

To facilitate the transformation of ProDOS files or Macintosh HFS files to ISO 9660 files on CD-ROM without loss of needed ProDOS or Macintosh file information Apple has defined a protocol that extends the ISO 9660 specification. Discs created using the Apple extensions are valid ISO 9660 discs, and retain the filename as well as the filetype/auxiliary type (ProDOS) or filetype/creator/bundle bit/icon resource (Macintosh) information needed to reconstruct the original files from which they were made.

Because ISO 9660 does not provide for file typing and icons, the extra information is stored in a special data structure in the file's directory record. Filenames are preserved through transformations of ProDOS or Macintosh filenames to valid ISO 9660 names, and back again.

This section does not discuss the protocol in detail. Please see Appendix E, "Apple Extensions to ISO 9660,"

if you need to create or work with ProDOS or Macintosh files stored as ISO 9660 files. Here are the main highlights of the protocol:

- **The Protocol identifier:** The protocol identifier consists of 32 bytes in the `systemIdentifier` field of the Standard Volume Descriptor of an ISO 9660 volume. It is the characters "`APPLE COMPUTER, INC., TYPE: `" followed by 4 bytes of protocol flags. The current version of the type description gives the version number of the protocol and indicates whether the disc's files should be transformed to ProDOS file names when read.

  The presence of the protocol identifier indicates that the Apple extensions have been applied to the disc's files.

- **The systemUse field:** The `systemUse` field in the file's directory record is an optional field. The Apple extensions use that field to store the extra file information. If the `systemUse` field is present, and if it begins with the proper signature word, the subsequent information in the field can be interpreted as ProDOS or Macintosh HFS information.

- **ProDOS filename transformations:** If you (through an authoring tool) are creating ISO 9660 files from ProDOS files, you can transform ProDOS filenames to valid ISO 9660 filenames in such a way that users (through a receiving system) can access the files using their original ProDOS filenames. Do this:

1. Replace all periods in the ProDOS filename with underscores. If the file is a directory file, that completes the transformation.
2. If the file is not a directory file, append the characters ".;1" to the filename. It is now a valid ISO 9660 filename.

   In use, the receiving system performs the above transformation on user-supplied filenames before searching for them on disc and reverses the transformation before presenting filenames to the user.

   If the transformation is to be done, it must be applied to all files on a disc.

- **HFS filename transformations**: Unlike with ProDOS, it is not possible to make a simple, reversible transformation from all valid Macintosh HFS filenames to valid ISO 9660 filenames. To make the transformation as consistent as possible, however, Apple recommends these guidelines:

1. Convert all lowercase characters to uppercase.
2. Replace all illegal characters, including periods, with underscrores.
3. If the filename needs to be shortened, truncate the rightmost characters.
4. If the file is not a directory file, append the characters ".;1" to the filename.

   Such a transformation is not perfectly reversible, but its results are at least consistent across all files and discs.

High Sierra FST calls

Table 10-1 lists all the GS/OS system calls supported by the High Sierra FST. Those in column 1 perform meaningful tasks; those in column 2 always return an error (with the exception of Flush; see the call description).

Table 10-1 High Sierra FST calls

Meaningful		Not meaningful	
$2006	GetFileInfo	$2001	Create
$2008	Volume	$2002	Destroy

$2010	Open	$2004	ChangePath
$2012	Read	$2005	SetFileInfo
$2014	Close	$2013	Write
$2016	SetMark	$2015	Flush
$2017	GetMark	$2018	SetEOF
$2019	GetEOF	$200B	ClearBackupBit
$201C	GetDirEntry	$2024	Format
$2020	GetDevNum	$2025	EraseDisk
$2033	FSTSpecific		

With the exception of Flush, all the calls on the right side of Table 10-1 do nothing and return error $2B (write-protected). Flush also does nothing, but it returns with the carry flag cleared (no error).

The following sections describe how the High Sierra FST's handling of some of the calls listed on the left side of Table 10-1 differs from standard GS/OS practice, as documented in Chapter 7. Calls listed on the left side of Table 10-1 that are *not* described below are handled exactly as documented in Chapter 7. Refer also to Chapter 7 for complete explanations of the calls and parameters listed here.

GetFileInfo ($2006)

GetFileInfo returns certain attributes of an existing block file. The file may be open or closed.

Parameter differences

`access` — The only possible values for this parameter under High Sierra/ISO 9660 are $01 (read-permission only) and $05 (read-permission only, file invisible).

`fileType` — This output word value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file-type mapping table. See Appendix E "Apple Extensions to ISO 9660"; see also the call FSTSpecific, described later in this chapter.

`modDateTime` — This output double longword value always has the same value as `createDateTime`.

`auxType` — This output long word value is always $0000 unless the Apple extensions to ISO 9660 have been applied. See Appendix E.

`optionList` This is a longword input pointer to a data area to which results can be returned. If an Extended Attribute Record (XAR) exists for the file, the High Sierra FST returns the contents of the XAR in the data area pointed to. If an XAR does not fit in the allotted space, the High Sierra FST returns as much of the data as possible and generates error $4F (buffer too small).

Errors

In addition to the standard GS/OS GetFileInfo errors, the High Sierra FST can return these errors from a GetFileInfo call:

$4F buffer too small

Volume ($2008)

Given the name of a block device, Volume returns the name of the volume mounted in the device and other information about the volume.

Parameter differences

`freeBlocks` This longword output value is aways $0000.

`fileSysID` This word result value describes the file system of the volume being accessed. For High Sierra, `fileSysID` = $000B; for ISO 9660, `fileSysID` = $000C. If any other type of volume is accessed, the High Sierra FST returns error $52 (unsupported volume type).

Open ($2010)

This call causes the FST to establish an access path to a file. Once an access path is established, the user may perform file reads and other related operations on the file.

A file can be opened more than once as long as it is not opened for write access, and each open assigns a different reference number. Because High Sierra/ISO 9660 files are read-only, it is always possible to have multiple open copies of a document.

Parameter differences

`requestAccess` If this word input parameter is included, and if its value is anything other than $0000 (use default permissions stored with file) or $0001 (read-access requested), the High Sierra FST returns error $4E (access not allowed).

`fileType` This word output value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file-type mapping table. See Appendix E "Apple Extensions to ISO 9660"; see also the FSTSpecific call, described later in this chapter.

`auxType` This longword output value is always $0000 unless the Apple extensions to ISO 9660 have been applied. See Appendix E.

`modDateTime` This double longword output parameter always has the same value as `createDateTime`.

`optionList` This is a longword input pointer to a data area to which results can be returned.. If an Extended Attribute Record (XAR) exists for the file, the High Sierra FST returns the contents of the XAR in the data area pointed to. If an XAR does not fit in the allotted space, the High Sierra FST returns as much of the data as possible and generates error $4F (buffer too small).

`fileType` This output word value equals $000F if the file is a directory; otherwise, it is $0000 ("unknown")—*unless* the filename extension matches an entry in the file-type mapping table. See Appendix E "Apple Extensions to ISO 9660"; see also the call FSTSpecific, described later in this chapter.

`auxType` This output long word value is always $0000 unless the Apple extensions to ISO 9660 have been applied. See Appendix E.

Errors

In addition to the standard GS/OS Open errors, the High Sierra FST can return this error from an Open call:

$4F buffer too small

Read ($2012)

This call attempts to transfer the requested number of bytes, starting at the current mark, from a specified file into a specified buffer. The file mark is updated to reflect the number of bytes read.

The High Sierra FST allows applications to read directory files as well as data files (but only with standard GS/OS calls; ProDOS 16 Read calls to directories return error $4E—access not allowed). Even so, as a reminder that directory structures differs for different file systems, the High Sierra FST always returns a "caution" error ($66) after a successful read of a directory.

Also, the High Sierra FST does not allow Read calls and GetDirEntry calls with the same file reference number: if an open file has previously been accessed by GetDirEntry, and a Read call is made with the same reference number, the High Sierra FST returns error $4E (access not allowed). To avoid that error, open the directory twice.

Errors

In addition to the standard GS/OS Read errors, the High Sierra FST can return this error from a Read call:

$66 FST Caution

GetDirEntry ($201C)

This call returns information about a directory entry in the volume directory or a subdirectory. Before executing this call, the application must open the directory or subdirectory. The call allows the application to step forward or backward through file entries or to specify absolute entries by entry number.

The High Sierra FST does not allow Read calls and GetDirEntry calls with the same file reference number: if an open file has previously been accessed by Read, and a GetDirEntry call is made with the same reference number, the High Sierra FST returns error $4E (access not allowed). To avoid that error, open the directory twice.

Parameter differences

`fileType` This output word value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file-type mapping table. See Appendix E "Apple Extensions to ISO 9660"; see also the FSTSpecific call, described later in this chapter.

`modDateTime` This double longword output parameter always has the same value as `createDateTime`.

`auxType` This longword output value is always $0000 unless the Apple extensions to ISO 9660 have been applied. See Appendix E.

`fileSysID` This word result value describes the file system of the directory being accessed. For High Sierra, `fileSysID` = $000B; for ISO 9660, `fileSysID` = $000C. If any other type of directory is accessed, the High Sierra FST returns error $52 (unsupported volume type).

`optionList` This is a longword input pointer to a data area to which results can be returned. If an Extended Attribute Record (XAR) exists for the file, the High Sierra FST returns the contents of the XAR in the data area pointed to. If an XAR does not fit in the allotted space, the High Sierra FST returns as much of the data as possible and generates error $4F (buffer too small).

Errors:

In addition to the standard GS/OS GetDirEntry errors, the High Sierra FST can return this error from a GetDirEntry call:

$4F buffer too small

$2033 FSTSpecific

Note: FSTSpecific is a call that is not described with the rest of the GS/OS calls in Chapter 7. Its function can be defined individually for any file system translator.

Description The High Sierra FST uses the call FSTSpecific to control file-type mapping. That is, it simulates file types in High Sierra/ISO 9660 files (which do not have file types) by mapping filename extensions to specific GS/OS file types. FSTSpecific maintains a table in memory that controls which extensions correspond to which file types.

The default table contains only two entries; it equates filenames with extensions of `.txt` and `.bat` to GS/OS file type $04 (text file).

FSTSpecific uses a command number as one of its parameters and therefore functions as four different calls. The four calls are:

MapEnable	Enables/disables file-type mapping
GetMapSize	Returns size, in bytes, of current file-type map
GetMapTable	Returns the current file-type map
SetMapTable	Replaces the current file-type map

Note: This mapping is independent of and unrelated to the file-typing implemented by the Apple extensions to ISO 9660 described in Appendix E.

Parameters This is the FSTSpecific parameter block:

Offset	Field	No.	Sixe and type	
$00	`pCount`	—	Word INPUT value	minimum = 3)
$02	`fileSysID`	1	Word INPUT value	
$04	`commandNum`	2	Word INPUT value	
⋮		3	(subcall-specific parameter)	

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

`fileSysID`	Word input value: The file system ID of the FST to which the call is directed. For High Sierra, `fileSysID` = $000B; for ISO 9660, `fileSysID` = $000C.
`commandNum`	Word input value: A number that specifies which particular subcall of FSTSpecific to execute, as follows:

subcall	`commandNum`
MapEnable	$0000
GetMapSize	$0001
GetMapTable	$0002
SetmapTable	$0003

See the individual subcall descriptions later in this chapter.

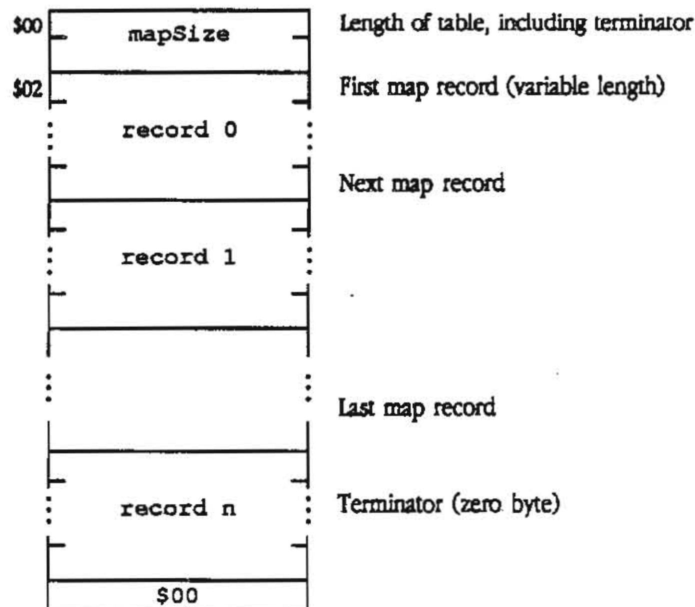
(subcall-specific) Word or longword input or result value: Depends on the specific subcall; see the individual subcall descriptions later in this chapter.

Errors

$04	parameter count out of range
$53	invalid parameter
$54	out of memory

What a map table is

The map table is the data structure that records which filename extensions are to be assigned to which filetypes. The format of a map table is as follows:



A map record consists of a text string (with MSBs off) followed by a zero byte followed by a file type byte. The text string can be any length and can include any legal characters for a High Sierra filename (text must be uppercase, for example). In APW assembly language, a map table can be defined as follows:

```
mapTable    dc    i2'end-mapTable+1'      ;Length of table.
      dc    c'.TXT',h'00 04'  ;Record 0.
      dc    c'.TYPE',h'00 7f' ;Record 1.
end   dc    h'00' ;Terminator.
```

MapEnable (FSTSpecific subcall)

The MapEnable subcall toggles file mapping on or off.

Parameters This is the FSTSpecific parameter block for the MapEnable subcall:

\$00	pCount
\$02	fileSysID
\$04	commandNum
\$06	enable

The following parameters have particular values for this subcall.

`commandNum` For MapEnable, `commandNum` = \$0000.

`enable` Word input value that equals either \$0000 or \$0001. If `enable` = \$0000, file-type mapping is disabled. If `enable` = \$0001, file-type mapping is enabled.

GetMapSize (FSTSpecific subcall)

The GetmapSize subcall returns the size of the current file map.

Parameters This is the FSTSpecific parameter block for the GetMapSize subcall:

\$00	pCount
\$02	fileSysID
\$04	commandNum
\$06	mapSize

The following parameters have particular values for this subcall.

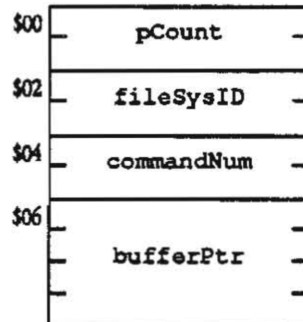
`commandNum` For GetMapSize, `commandNum` = \$0001.

`mapSize` Word result value that is equal to the size (in bytes) of the current map table.

GetMapTable (FSTSpecific subcall)

The subcall GetMapTable returns a pointer to the current map table.

Parameters This is the FSTSpecific parameter block for the GetMapTable subcall:



The following parameters have particular values for this subcall.

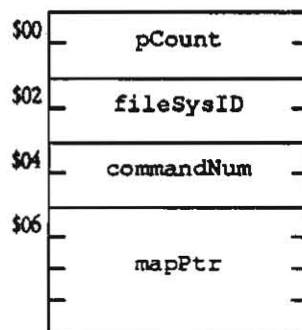
commandNum For GetMapTable, commandNum = $0002.

bufferPtr A longword input pointer to a memory area large enough to hold the map table that will be returned by the call.

SetMapTable (FSTSpecific subcall)

The subcall SetMapTable sets the current map table to the one pointed to by the input pointer.

Parameters This is the FSTSpecific parameter block for the SetMapTable subcall:



The following parameters have particular values for this subcall.

commandNum For SetMapTable, commandNum = $0003.

`mapPtr` Longword input pointer to the new map table. As long as there is space in memory for the new table, it will replace the old one. If there is not enough space, error $54 (out of memory) is returned and the original table remains in effect. No validity checking is done on the table.

Chapter 11 **The Character FST**

The Character file system translator (Character FST) provides a file-system-like interface to character devices such as the console, printers, and modems. The Character FST works with both generated and loaded drivers.

Note: This chapter describes only standard GS/OS (class 1) calls; for descriptions of how the Character FST handles equivalent ProDOS 16 (class 0) calls, see Appendix B.

Character devices as files

The Character FST eanbles applications to read from and write to character devices as if they were files. That is, your application can open , read, write, and close a printer, modem, console, or other character device in a manner exactly analogous to performing those actions on a file on a block device.

Not all GS/OS calls can be made to character devices, of course, and those that do may not always function exactly the same as for block devices. This chapter discusses those calls that do apply to character devices and notes any character-device-specific features they have.

Note: Although GS/OS lets you treat character devices as files in some ways, you cannot create, destroy, or rename character files with GS/OS calls. The system and the user control the existence and the names of character devices

The Character FST allows multiple Open calls, with both read- and write-access, to a character file. Block-device FSTs, on the other hand, can allow multiple Opens for read-access only.

Character FST calls

The Character FST supports this subset of GS/OS calls:

Open
Newline
Read
Write
Close
Flush

All other GS/OS calls return error $58 (Not a block device).

The following descriptions explain how the Character FST responds to some of these calls differently from the standard procedures documented in Chapter 7. Any of the supported calls not described here function exactly as documented in Chapter 7.

Open ($2010)

Open establishes an access path to a character file. With the `requestAccess` parameter, an application can request limited access rights to the character file.

Parameter differences

`pCount`	Maximum value = 3. Unlike with block devices, you cannot use the Open call to a character device to get information normally returned by GetFileInfo.
`pathname`	This input pointer must point to a character device name.
`requestAccess`	The following values are allowed: $00 open with available permissions $01 open for read-access only $02 open for write-access only $03 open for both read- and write-access

Errors

In addition to the standard GS/OS Open errors, the Character FST can return these errors from an Open call:

$04	pCount error
$24	driver prior open
$26	driver no resources
$28	driver no device
$2F	driver off line
$54	out of memory

Read ($2012)

The Read call attempts to transfer the requested number of bytes from the specified character file into the application's data buffer.

Parameter differences

`pCount`	Minimum is 4; maximum is 4.
`cachePriority`	Not used. Data transfers with character devices are not cached.

Errors

In addition to the standard GS/OS Read errors, the Character FST can return these errors from a Read call:

$04	pCount error
$23	driver not open
$2F	driver off line
$53	parameter out of range
$54	out of memory

Write ($2013)

The Write call attempts to transfer the requested number of bytes from the application's data buffer to the specified character file.

Parameter differences

`pCount` Minimum is 4; maximum is 4.

`cachePriority` Not used. Data transfers with character devices are not cached.

Errors

In addition to the standard GS/OS Write errors, the Character FST can return these errors from a Write call:

$04	pCount error
$23	driver not open
$2F	driver off line

Close ($2014)

The Close call terminates access to the specified (by `refNum`) character file. Close also involves flushing the file (see the Flush call), to ensure completion of all data transfer before a character file is closed.

Errors

In addition to the standard GS/OS Close errors, the Character FST can return these errors from a Close call:

$04	pCount error
$23	driver not open
$2F	driver off line

Flush ($2015)

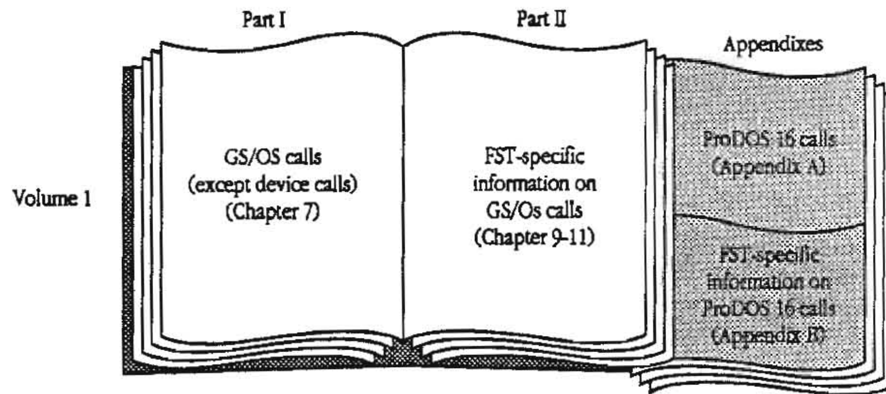
The Flush routine completes any pending data transfer to the character file specifed by `refNum`. If the character device is synchronous, all data transfer is by definition completed when the Write call returns, so the Flush routine simply returns with no error. If the device is asynchronous (such as interrupt-driven or direct memory access), the Flush routine waits until all data has been transferred and then returns. If the file is multiply opened, all (output) access paths to the character file (not just the one with the specified `refNum`) are flushed.

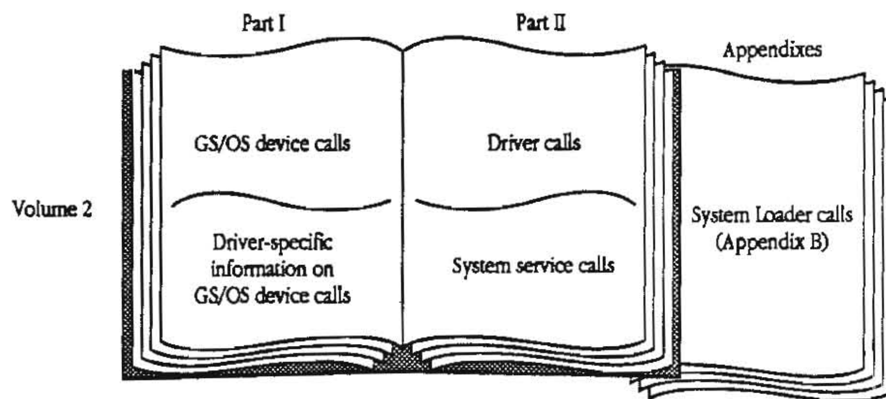
Errors

In addition to the standard GS/OS Flush errors, the Character FST can return these errors from a Flush call:

$04	pCount Error
$23	Driver not open
$2F	Driver Off Line

Appendixes





Appendix A **GS/OS ProDOS 16 Calls**

This appendix provides a detailed description of all the GS/OS ProDOS 16 calls, arranged in alphabetical order by call name. These calls are provided only for compatibility with ProDOS 16. For the standard GS/OS calls, see Chapter 7, "GS/OS Call Reference," in Part I of this manual.

The descriptions in this appendix follow the same conventions as those for the standard GS/OS calls.

$0031 ALLOC_INTERRUPT

Description

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

For a complete description of GS/OS's interrupt handling subsystem, see Volume 2. See also the DEALLOC_INTERRUPT call in this appendix.

Parameters

Offset		Size and type
$00	intNum	Word RESULT value
$02	intCode	Longword INPUT pointer

`intNum`

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

`intCode`

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

Errors

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

$0004 CHANGE_PATH

Description

This call changes a file's pathname to another pathname on the same volume, or renames a volume.

CHANGE_PATH cannot be used to change a device name. You must use the configuration program to change device names.

Parameters

Offset	Parameter	Size and type
$00	pathname	Longword INPUT pointer
$04	newPathname	Longword INPUT pointer

`pathname` Longword input pointer: Points to a Pascal string that represents the name of the file whose pathname is to be changed.

`newPathname` Longword input pointer: Points to a Pascal string that represents the new pathname of the file whose name is to be changed.

Comments

A file may not be renamed while it is open.

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

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

Errors

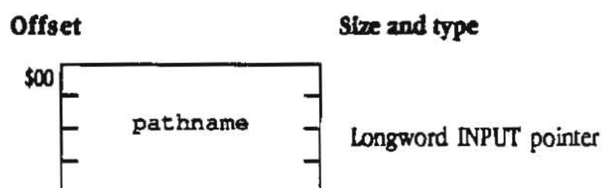
$10	device not found
$27	I/O error
$2B	write-protected disk
$40	invalid pathname syntax
$44	path not found
$45	volume not found
$46	file not found
$47	duplicate pathname
$4A	version error
$4B	unsupported storage type
$4E	access: file not destroy-enabled
$50	file open
$52	unsupported volume type
$53	invalid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

$000B CLEAR_BACKUP_BIT

Description

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

Parameters



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

Errors

$27	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

$0014 CLOSE

Description

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

If the specified value of the `fileRefNum` parameter is $0000, all files at or above the current system file level are closed.

Parameters

Offset		Size and type
$00	fileRefNum	Word INPUT value

`fileRefNum` Word input value: The identifying number assigned to the file by the OPEN call. A value of $0000 indicates that all files at or above the current system file level are to be closed.

Errors

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

$0001 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.

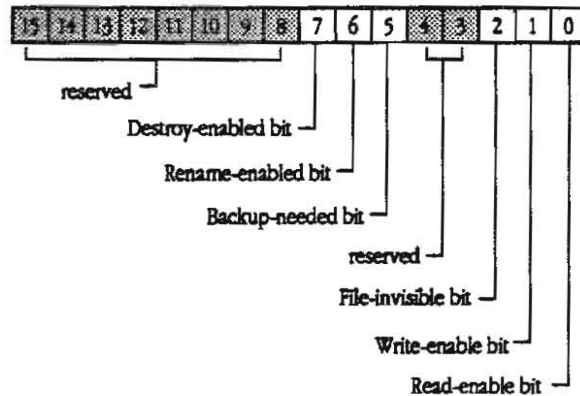
This call sets up file system state information for the new file and initializes the file to the empty state.

Parameters

Offset		Size and type
$00	pathname	Longword INPUT pointer
$04	fAccess	Word INPUT value
$06	fileType	Word INPUT value
$08	auxType	Longword INPUT value
$0C	storageType	Word INPUT value
$0E	createDate	Word INPUT value
$10	createTime	Word INPUT value

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

`fAccess` Word input value: Specifies how the file may be accessed after it is created and whether or not the file has changed since the last backup.



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

Software that supports file hiding (invisibility) should use the I bit to indicate whether or not to display a file or subdirectory.

`fileType` Word input value: Used conventionally by system and application programs to categorize the file's contents. The value of this field has no effect on GS/OS's handling of the file, except that only certain file types may be executed directly by GS/OS. Many file types have already been standardized by Apple, as listed in Table 1-2 in Chapter 1.

`auxType` Longword input value: Used by system and application programs to store additional information about the file. The value of this field has no effect on GS/OS's handling of the file. By convention, the interpretation of values in this field depends on the value in the `fileType` field. Many auxiliary types have been standardized by Apple, as listed in Table 1-2 in Chapter 1.

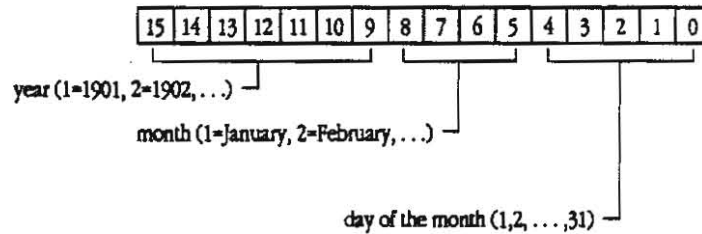
`storageType` Word input value: The value of this parameter determines whether the file being created is a standard file, extended file, or subdirectory file, as follows:

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

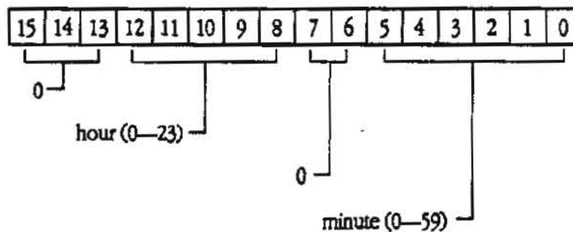
All other values are invalid.

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

`createDate` Word input value: This parameter specifies a date that GS/OS saves as the file's creation date value. If this word is $0000, GS/OS gets the date from the system clock.



`createTime` Word input value: This parameter specifies the time that GS/OS saves as the file's creation time value. If this word is $0000, GS/OS gets the time from the system clock.



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

The storage type of a file cannot be changed after it is created. For example, there is no direct way to add a resource fork to a standard file or to remove one of the forks from an extended file.

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

Errors

$10	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

$0032 DEALLOC_INTERRUPT

Description This function removes a specified interrupt handler from the interrupt vector table. See also the ALLOC_INTERRUPT call in this appendix.

Parameters

Offset		Size and type
$00	intNum	Word INPUT value

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

Errors

$53 parameter out of range

$0002 DESTROY

Description

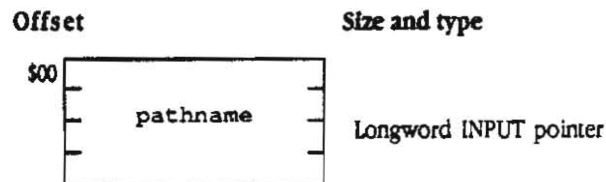
This call deletes a specified standard file, extended file (both the data fork and resource fork), or subdirectory and updates the state of the file system to reflect the deletion. After a file is destroyed, no other operations on the file are possible.

This call cannot be used to delete a volume directory; the FORMAT call reinitializes volume directories. Similarly, this call cannot be used to delete character-device file.

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

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

Parameters



`pathname` — Longword input pointer: Points to a Pascal string that represents the pathname of the file to be deleted.

Comments

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

Errors

$10	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

$002C D_INFO

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

Parameters

Offset	Field	Size and type
$00	devNum	Word INPUT value
$02	devName	Longword INPUT pointer

devNum — Word input value: A device number. GS/OS assigns device numbers in sequence (1, 2, 3, and so on) as it loads or creates the device drivers. There is no fixed correspondence between devices and device numbers. To get information about every device in the system, one makes repeated calls to D_INFO with devNum values of 1, 2, 3, and so on until GS/OS returns error $53: parameter out of range.

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

Errors

$11 invalid device number
$53 parameter out of range

$0025 ERASE_DISK

Description

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

Parameters

Offset	Field	Size and type
$00	devName	Longword INPUT pointer
$04	volName	Longword INPUT pointer
$08	fileSysID	Word RESULT value

`devName` Longword input pointer: Points to a Pascal string that represents the device name of the device containing the volume to be erased.

`volName` Longword input pointer: Points to a Pascal string that represents the volume name to be assigned to the newly erased volume.

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

$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

Errors

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

$000E EXPAND_PATH

Description

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

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

Parameters

Offset		Size and type
$00	inputPath	Longword INPUT pointer
$04	outputPath	Longword INPUT pointer
$08	flags	Word INPUT value

`inputPath` Longword input pointer: Points to a Pascal input string that is to be expanded.

`outputPath` Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string that contains the expanded pathname.

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

Errors

$40 invalid pathname syntax
$4F buffer too small

$0015 FLUSH

Description

This call writes to the volume all file state information that is buffered in memory but has not yet been written to the volume. The purpose of this call is to assure that the representation of the file on the volume is consistent and up to date with the latest GS/OS calls affecting the file. Thus, if a power failure occurs immediately after the FLUSH call completes, it should be possible to read all data written to the file as well as all file attributes. If such a power failure occurs, files that have not been flushed may be in inconsistent states, as may the volume as a whole.

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

Parameters

Offset		Size and type
$00	fileRefNum	Word INPUT value

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

Errors

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

$0024 FORMAT

Description

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

Some devices do not support physical formatting, in which case the FORMAT call writes only the empty file system, and in effect is just like the ERASE_DISK call. See the ERASE_DISK call earlier in this chapter.

Parameters

Offset	Field	Size and type
$00	devName	Longword INPUT pointer
$04	volName	Longword INPUT pointer
$08	fileSysID	Word RESULT value

`devName` Longword input pointer: Points to a Pascal string that represents the device name of the device containing the volume to be formatted.

`volName` Longword input pointer: Points to a Pascal string that represents the volume name to be assigned to the newly formatted blank volume.

`fileSysID` Word result value: If the call is successful, this field identifies the file system with which the disk was formatted. If the call is unsuccessful, this field is undefined. The file system IDs are as follows:

$0000	reserved	$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

Errors

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

$0028 GET_BOOT_VOL

Description

This call returns the volume name of the volume from which the file GS/OS was last loaded and executed. The volume name returned by this call is equivalent to the prefix specified by */.

Parameters

Offset		Size and type
$00	dataBuffer	Longword INPUT pointer

`dataBuffer` Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string containing the boot volume name.

Errors

$4F buffer too small

$0020 GET_DEV_NUM

Description

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

GS/OS assigns device numbers at boot time. The numbers are a series of consecutive integers beginning with 1. There is no algorithm for determining the device number for a particular device.

Because a device may hold different volumes and because volumes may be moved from one device to another, the device number returned for a particular volume name may be different at different times.

Parameters

Offset	Field	Size and type
$00	devName	Longword INPUT pointer
$04	devNum	Word RESULT value

`devName` Longword input pointer: Points to a Pascal string that represents the device name or volume name (for a block device).

`devNum` Word result value: The device reference number of the specified device.

Errors

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

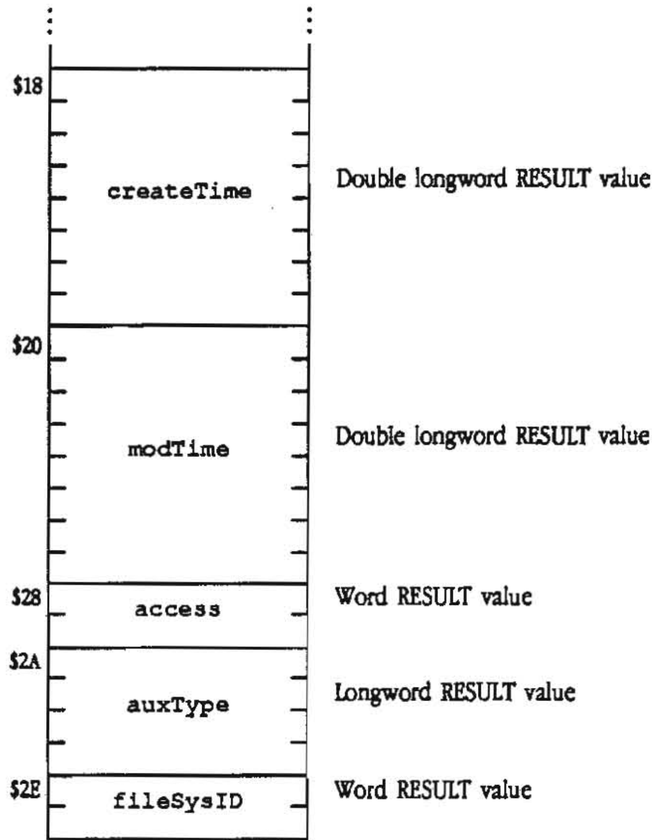
$001C GET_DIR_ENTRY

Description

This call returns information about a directory entry in the volume directory or a subdirectory. Before executing this call, the application must open the directory or subdirectory. The call allows the application to step forward or backward through file entries or to specify absolute entries by entry number.

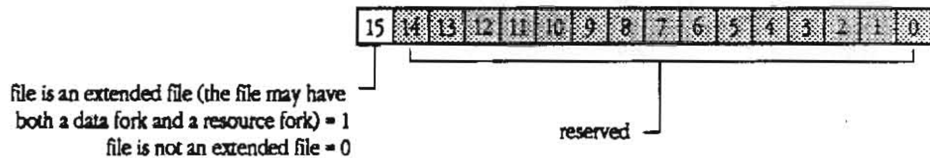
Parameters

Offset	Field	Size and type
$00	refNum	Word INPUT value
$02	flags	Word RESULT value
$04	base	Word INPUT value
$06	displacement	Word INPUT value
$08	nameBuffer	Longword INPUT pointer
$0C	entryNum	Word RESULT value
$0E	fileType	Longword RESULT value
$10	endOfFile	Longword RESULT value
$14	blockCount	Longword RESULT value
$18	⋮	



`refNum` Word input value: The identifying number assigned to the directory or subdirectory by the OPEN call.

`flags` Word result value: Flags that indicate various attributes of the file.



`base` Word input value: A value that tells how to interpret the displacement field, as follows:

$0000	displacement gives an absolute entry number
$0001	displacement is added to current displacement to get next entry number

	$0002 displacement is subtracted from current displacement to get next entry number
`displacement`	Word input value: In combination with the base parameter, the displacement specifies the directory entry whose information is to be returned. When the directory is first opened, GS/OS sets the current displacement value to $0000. The current displacement value is updated on every GET_DIR_ENTRY call. If the base and displacement fields are both zero, GS/OS returns a 2-byte value in the `entryNumber` parameter that specifies the total number of active entries in the subdirectory. In this case, GS/OS also resets the current displacement to the first entry in the subdirectory. To step through the directory entry by entry, you should set both the `base` and `displacement` parameters to $0001.
`nameBuffer`	Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string containing the name of the file or subdirectory represented in this directory entry.
`entryNum`	Word result value: The absolute entry number of the entry whose information is being returned. This field is provided so that a program can obtain the absolute entry number even if the `base` and `displacement` parameters specify a relative entry.
`fileType`	Longword result value: The value of the file type field of the directory entry.
`endOfFile`	Longword result value: Value of the EOF field of the directory entry.
`blockCount`	Longword result value: The value of the blocks used field of the directory entry.
`createTime`	Double longword result value: The value of the creation date/time field of the directory entry.
`modTime`	Double longword result value: The value of the modification date/time field of the directory entry.
`access`	Word result value: Value of the access attribute field of the directory entry.
`auxType`	Longword result value: Value of the auxiliary type field of the directory entry.
`fileSysID`	Word result value: File system identifier of the file system on the volume containing the file. Values of this field are described under the VOLUME call.

Errors

$10	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

$0019 GET_EOF

Description

This function returns the current logical size of a specified file. See also the SET_EOF call in this appendix.

Parameters

Offset	Field	Size and type
$00	eofRefNum	Word INPUT value
$02	eofPosition	Longword RESULT value

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

`eof` Longword result value: The current logical size of the file, in bytes.

Errors

$43 invalid reference number

$0006 GET_FILE_INFO

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

Important A GET_FILE_INFO call following a SET_FILE_INFO call on an open file may not return the values set by the SET_FILE_INFO call. To guarantee recording of the attributes specified in a SET_FILE_INFO call, you must first close the file.

See also the SET_FILE_INFO call in this appendix.

Parameters

Offset	Field	Size and type
$00	pathname	Longword INPUT pointer
$04	fAccess	Word RESULT value
$06	fileType	Word RESULT value
$08	auxType	Longword RESULT value
$0C	storageType	Word RESULT value
$0E	createDate	Word RESULT value
$10	createTime	Word RESULT value
$12	modDate	Word RESULT value
$14	modTime	Word RESULT value
$16	blocksUsed	Longword RESULT value

`pathname`	Longword input pointer: Points to a Pascal string representing the pathname of the file whose file information is to be retrieved.
`fAccess`	Word result value: Value of the file's access attribute, which is described under the CREATE call.
`fileType`	Word result value: Value of the file's file type attribute.
`auxType`	Longword result value: Value of the file's auxiliary type attribute.
`storageType`	Word result value: Value indicating the storage type of the file, as follows: $01 standard file $05 extended file $0D volume directory or subdirectory file
`createDate`	Word result value: Value for the file's creation date attribute which is described under the CREATE call.
`createTime`	Word result value: Value for the file's creation time attribute, which is described under the CREATE call.
`modDate`	Word result value: Value for the file's modification date attribute. The format is the same as the `createDate` parameter.
`modTime`	Word result value: Value for the file's modification time attribute. Format is the same as the `createTime` parameter.
`blocksUsed`	Longword result value: For a standard file, this field gives the total number of blocks used by the file. For an extended file, this field gives the number of blocks used by the file's data fork. For a subdirectory or volume directory file, this field is undefined.

Errors

$10	device not found
$27	I/O error
$40	invalid pathname syntax
$44	path not found
$45	volume not found
$46	file not found
$4A	version error
$4B	unsupported storage type
$52	unsupported volume type
$53	invalid parameter
$58	not a block device

$0021 GET_LAST_DEV

Description

This call returns the device number of the last accessed device. The last accessed device is defined as the last device to which any device command was directed by GS/OS as the result of a GS/OS call.

A program that uses this call must take into account that the last device value can change at any time due to a device-accessing GS/OS call made by an asynchronously executed process such as a desk accessory or interrupt handler.

To insure that the GET_LAST_DEV call returns the last device accessed by the given program, the program must:

1. Disable interrupts.
2. Make the GS/OS call that accesses the device (for example, OPEN, READ).
3. Make the GET_LAST_DEV call.
4. Restore the interrupt state that was current before step 1.

Unfortunately, this sequence locks out interrupts for more than the maximum recommended interrupt disable time. Therefore, system integrity cannot be guaranteed, especially in a networked environment, where rapid interrupt handling is crucial.

Important Because of this danger to system integrity, use this call with caution, if at all.

Parameters

Offset		Size and type
$00	devNum	Word RESULT value

devNum Word result value: Device number of the last accessed device.

Errors

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

$001B GET_LEVEL

Description This function returns the current value of the system file level. See also the SET_LEVEL call in this appendix.

Parameters

Offset		Size and type
$00	level	Word RESULT value

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

Errors

- $01 bad system call number
- $04 parameter count out of range
- $07 GS/OS is busy
- $59 invalid file level

$0017 GET_MARK

Description This function returns the current file mark for the specified file. See also the SET_MARK call in this appendix.

Parameters

Offset		Size and type
$00	markRefNum	Word INPUT value
$02	position	Longword RESULT value

`markRefNum` Word input value: The identifying number assigned to the file by the OPEN call.

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

Errors

$43 invalid reference number

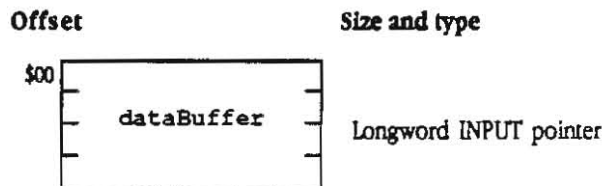
$0027 GET_NAME

Description

Returns the filename (not the complete pathname) of the currently running application program.

To get the complete pathname of the current application, concatenate prefix 1/ with the filename returned by this call. Do this before making any change in prefix 1/.

Parameters



`dataBuffer` Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string containing the filename.

Errors

$4F buffer too small

$000A GET_PREFIX

Description

This function returns the current value of any one of the numbered prefixes. The returned prefix string always starts and ends with a separator. If the requested prefix is null, it is returned as a string with the length field set to 0. This call should not be used to get the boot volume prefix (*/). See also the SET_PREFIX call in this appendix.

Parameters

Offset	Field	Size and type
$00	prefixNum	Word INPUT value
$02	prefix	Longword INPUT pointer

`prefixNum` Word input value: Binary value of the prefix number for the prefix to be returned.

`prefix` Longword input pointer: Points to a buffer in which GS/OS returns a Pascal string containing the prefix value.

Errors

$4F	buffer too small
$53	invalid parameter

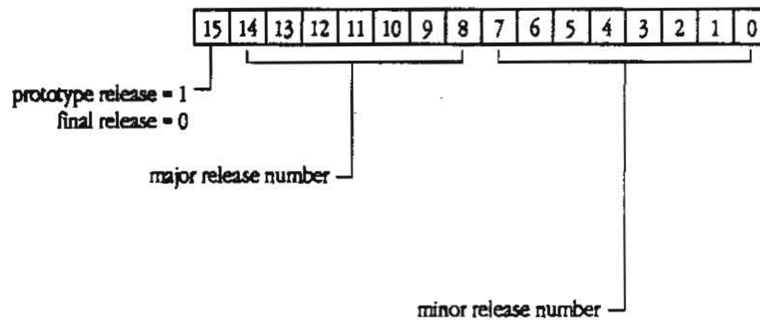
$002A GET_VERSION

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

Parameters

Offset		Size and type
$00	`version`	Word RESULT value

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



Errors

(none except general system errors)

$0011 NEWLINE

Description

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

When newline mode is disabled, a READ call terminates only after it reads the requested number of characters or encounters the end of file. When newline mode is enabled, the read also terminates if it encounters one of the specified newline characters.

When a READ call is made while newline mode is enabled and another character is in the file, GS/OS performs the following operations:

1. Transfers next character to user's buffer.
2. Performs a logical AND between the character and the low order byte of the newline mask specified in the last NEWLINE call for the open file.
3. Compares the resulting byte with the newline character or characters.
4. If there is a match, terminates the read; otherwise continues at step 1.

Parameters

Offset	Field	Size and type
$00	newLRefNum	Word INPUT value
$02	enableMask	Word INPUT value
$04	newlineChar	Word INPUT value

`newLRefNum` Word input value: The identifying number assigned to the file access path by the OPEN call.

`enableMask` Word input value: If the value of this field is $0000, newline mode is disabled. If the value is greater than $0000, the low-order byte becomes the newline mask. GS/OS performs a logical AND of each input character with the newline mask before comparing it to the newline characters.

`newlineChar` Word input value: The low-order byte of this field is the newline character. When disabling newline mode (`enableMask` = $0000), this parameter is ignored.

Errors

$43 invalid reference number

$0010 OPEN

Description

This call causes GS/OS to establish an access path to a file. Once an access path is established, the user may perform file READ and WRITE operations and other related operations on the file.

Parameters

Offset	Field	Size and type
$00	openRefNum	Word RESULT value
$02	openPathname	Longword INPUT pointer
$06	ioBuffer	Reserved

`openRefNum` Word result value: A reference number assigned by GS/OS to the access path. All other file operations (READ, WRITE, CLOSE, and so on) refer to the access path by this number.

`openPathname` Longword input pointer: Points to a Pascal string that represents the pathname of the file to be opened.

`ioBuffer` This field is reserved and must be set to $00000000.

Errors

$27	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	open file
$52	unsupported volume type
$58	not a block device

$0029 QUIT

Description

This call terminates the running application. It also closes all open files, sets the system file level to 0, initializes certain components of the Apple IIGS and the operating system, and then launches the next application.

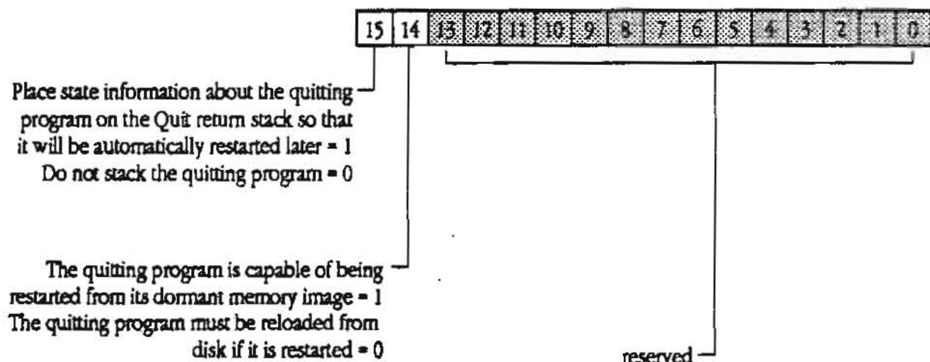
For more information about quitting applications, see Chapter 2, "GS/OS and Its Environment."

Parameters

Offset		Size and type
$00	quitPathname	Longword INPUT pointer
$04	flags	Word INPUT value

`quitPathname` Longword input pointer: Points to a Pascal string that represents the pathname of the program to run next. If the `quitPathname` parameter is null or the pathname itself has length 0, GS/OS chooses the next application, as described in Chapter 2.

`flags` Word input value: Two Boolean flags that give information about how to handle the program executing the QUIT call, as follows:



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

Errors

$07 GS/OS is busy

$0012 READ

Description

This function attempts to transfer the number of bytes given by the `requestCount` parameter, starting at the current mark, from the file specified by the `refNum` parameter into the buffer pointed to by the `dataBuffer` parameter. The function updates the file mark to reflect the new file position after the read.

Because of two situations that can cause the READ function to transfer fewer than the requested number of bytes, the function returns the actual number of bytes transferred in `transferCount`. If GS/OS reaches the end of file before transferring the number of bytes specified in `requestCount`, it stops reading and sets `transferCount` to the number of bytes actually read.

If newline mode is enabled and a newline character is encountered before the requested number of bytes have been read, GS/OS stops the transfer and sets `transferCount` to the number of bytes actually read, including the newline character.

Parameters

Offset		Size and type
$00	`fileRefNum`	Word INPUT value
$02	`dataBuffer`	Longword INPUT pointer
$06	`requestCount`	Longword INPUT value
$0A	`transferCount`	Longword RESULT value

`fileRefNum` Word input value: The identifying number assigned to the file by the OPEN call.

`dataBuffer`	Longword input pointer: Points to a memory area large enough to hold the requested data.
`requestCount`	Longword input value: The number of bytes to be read.
`transferCount`	Longword result value: The number of bytes actually read.

Errors

$27	I/O error
$2E	disk switched
$43	invalid reference number
$4C	eof encountered
$4E	access not allowed

$0022 READ_BLOCK

Description

This call reads one 512-byte block of information to a disk specified by device number.

Normally, you should use D_READ and D_WRITE for all direct device I/O. READ_BLOCK deals only with 512-byte blocks and devices with a maximum of 65,536 blocks, is valid only for the ProDOS FST, and exists only for compatibility with ProDOS 16.

Parameters

Offset	Field	Size and type
$00	blockDevNum	Word INPUT value
$02	blockDataBuffer	Longword INPUT pointer
$06	blockNum	Longword INPUT value

`blockDevNum` Word input value: The reference number assigned to the device.

`blockDataBuffer` Longword input pointer: Points to a data buffer large enough to hold the data to be read.

`blockNum` Longword input value: The number of the block to be read.

Errors

$11	invalid device request
$27	I/O error
$28	no device connected
$2B	write-protected disk
$53	invalid parameter

$0018 SET_EOF

Description

This call sets the logical size of an open file to a specified value which may be either larger or smaller than the current file size. The EOF value cannot be changed unless the file is write-enabled. If the specified EOF is less than the current EOF, the system may—but need not—free blocks that are no longer needed to represent the file. See also the GET_EOF call.

Parameters

Offset		Size and type
$00	eofRefNum	Word INPUT value
$02	eofPosition	Longword INPUT value

`eofRefNum` Word input value: The identifying number assigned to the file by the OPEN call.

`eofPosition` Longword input value: The new logical size of the file, in bytes.

Errors

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

$0005 SET_FILE_INFO

Description

This call sets certain file attributes of an existing open or closed block file. This call immediately modifies the file information in the file's directory entry whether the file is open or closed. It does not affect the file information seen by previously open access paths to the same file.

Important A GET_FILE_INFO call following a SET_FILE_INFO call on an open file may not return the values set by the SET_FILE_INFO call. To guarantee recording of the attributes specified in a SET_FILE_INFO call, you must first close the file.

See also the GET_FILE_INFO call.

Parameters

Offset		Size and type
$00	pathname	Longword INPUT pointer
$04	fAccess	Word INPUT value
$06	fileType	Word INPUT value
$08	auxType	Longword RESULT value
$0C	<null>	Word INPUT value
$0E	createDate	Word INPUT value
$10	createTime	Word INPUT value
$12	modDate	Word INPUT value
$14	modTime	Word INPUT value

`pathname`	Longword input pointer: Points to a Pascal string that represents the pathname of the file whose file information is to be set.
`fAccess`	Word input value: Value for the file's access attribute, which is described under the CREATE call.
`fileType`	Word input value: Value for the file's file type attribute.
`auxType`	Longword result value: Value of the file's auxiliary type attribute.
`<null>`	Word input value: This field is unused and must be set to zero.
`createDate`	Word input value: Value for the file's creation date attribute, which is described under the CREATE call. If the value of this field is zero, GS/OS does not change the creation date.
`createTime`	Word input value: Value for the file's creation time attribute, which is described under the CREATE call. If the value of this field is zero, GS/OS does not change the creation time.
`modDate`	Word input value: Value for the file's modification date attribute. Format is the same as for the `createDate` parameter. If the value of this field is zero, GS/OS supplies the date from the system clock.
`modTime`	Word input value: Value for the file's modification time attribute. Format is the same as for the `createTime` parameter. If the value of this field is zero, GS/OS supplies the time from the system clock.

Errors

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

$001A SET_LEVEL

Description

This function sets the current value of the system file level.

Whenever a file is opened, GS/OS assigns it a file level equal to the current system file level. A CLOSE call with a `refNum` parameter of $0000 closes all files with file level values at or above the current system file level. Similarly, a FLUSH call with a `refNum` parameter of $0000 flushes all files with file level values at or above the current system file level. See also the GET_LEVEL call in this appendix.

Parameters

Offset	Parameter	Size and type
$00	level	Word INPUT value

`level`

Word input value: The new value of the system file level. Must be in the range $0000–$00FF.

Errors

$59 invalid file level

$0016 SET_MARK

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 GET_MARK call in this appendix.

Parameters

Offset	Field	Size and type
$00	markRefNum	Word INPUT value
$02	position	Longword INPUT value

`markRefNum` Word input value: The identifying number assigned to the file by the OPEN call.

`position` Longword input value: The value assigned to the mark. It is the position (in bytes) relative to the beginning of the file at which the next read or write will begin.

Errors

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

$0009 SET_PREFIX

Description

This call sets one of the numbered pathname prefixes to a specified value. The input to this call can be any of the following pathnames:

- a full pathname
- a partial pathname beginning with a numeric prefix designator
- a partial pathname beginning with the special prefix designator "*/"
- a partial pathname without an initial prefix designator.

The SET_PREFIX call is unusual in the way it treats partial pathnames without initial prefix designators. Normally, GS/OS uses the prefix 0/ in the absence of an explicit designator. However, only in the SET_PREFIX call, it uses the prefix *n*/ where *n* is the value of the `prefixNum` field described below. See also the GET_PREFIX call in this appendix.

Parameters

Offset	Field	Size and type
$00	`prefixNum`	Word INPUT value
$02	`prefix`	Longword INPUT pointer

`prefixNum` Word input value: A prefix number that specifies the prefix to be set.

`prefix` Longword input pointer: Points to a Pascal string representing the pathname to which the prefix is to be set. If this field is not given, the prefix is set to the null string.

Comments

Specifying a pathname with length 0 or whose syntax is illegal sets the designated prefix to null.

GS/OS does not verify that the designated prefix corresponds to an existing subdirectory or file.

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

Errors

$40	invalid pathname syntax
$53	invalid parameter

$0008 VOLUME

Description Given the name of a block device, this call returns the name of the volume mounted in the device along with other information about the volume.

Parameters

Offset		Size and type
$00	deviceName	Longword INPUT pointer
$04	volName	Longword INPUT pointer
$08	totalBlocks	Longword RESULT value
$0C	freeBlocks	Longword RESULT value
$10	fileSysID	Word RESULT value

`deviceName` Longword input pointer: Points to a Pascal string that contains the name of a block device.

`volName` Longword input pointer: Points to a buffer in which GS/OS places a Pascal string containing the volume name of the volume mounted in the device.

`totalBlocks` Longword result value: Total number of blocks contained in the volume.

`freeBlocks` Longword result value: The number of free (unallocated) blocks in the volume.

`fileSysID` Word result value: Identifies the file system contained in the volume, as follows:

$0000	reserved	$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

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

$0013 WRITE

Description

This call attempts to transfer the number of bytes specified by the `requestCount` parameter from the application's buffer to the file specified by the `fileRefNum` parameter, starting at the current file mark.

The call returns the number of bytes actually transferred. It also updates the file mark to indicate the new file position and extends the EOF, if necessary, to accommodate the new data.

Parameters

Offset	Parameter	Size and type
$00	`fileRefNum`	Word INPUT value
$02	`dataBuffer`	Longword INPUT pointer
$06	`requestCount`	Longword INPUT value
$0A	`transferCount`	Longword RESULT value

`fileRefNum` Word input value: The identifying number assigned to the file by the OPEN call.

`dataBuffer` Longword input pointer: Points to the area of memory containing the data to be written to the file.

`requestCount` Longword input value: The number of bytes to write.

`transferCount` Longword result value: The number of bytes actually written.

Errors

$27	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

$0023 WRITE_BLOCK

Description

This call writes one 512-byte block of information to a disk specified by device number.

Normally, you should use D_READ and D_WRITE for all direct device I/O. WRITE_BLOCK deals only with 512-byte blocks and devices with a maximum of 65,536 blocks, is valid only for the ProDOS FST, and exists only for compatibility with ProDOS 16.

Parameters

Offset	Field	Size and type
$00	blockDevNum	Word INPUT value
$02	blockDataBuffer	Longword INPUT pointer
$06	blockNum	Longword INPUT value

`blockDevNum` Word input value: The reference number assigned to the device.

`blockDataBuffer` Longword input pointer: Points to a data buffer that holds the data to be written.

`blockNum` Longword input value: The block number of the destination disk block.

Errors

$11	invalid device request
$27	I/O error
$28	no device connected
$2B	write-protected disk
$53	invalid parameter

Appendix B **ProDOS 16 Calls and FSTs**

This appendix discusses how individual GS/OS file system translators handle ProDOS 16 (= GS/OS class 0) calls. It shows only the differences in each FST's call handling from what is presented in Appendix A, "GS/OS ProDOS 16 Calls." See that appendix for the standard way to make ProDOS 16 calls to GS/OS.

The ProDOS FST

The ProDOS FST translates ProDOS 16 calls to the format used by the ProDOS file system. Actually, because that is already the file system that ProDOS 16 calls are designed to access, no translation is necessary. All GS/OS ProDOS 16 calls that pass through the ProDOS FST function exactly as described in Appendix A.

See Chapter 9 of this volume for more information on the ProDOS FST. For further information on ProDOS 16, see the *Apple IIGS ProDOS 16 Reference.*

The High Sierra FST

The main difference between the High Sierra FST and other FSTs is that High Sierra does not support writing to a file. CD-ROM is a read-only medium.

Table B-1 lists the ProDOS 16 calls, both meaningful and not meaningful, that the High Sierra FST supports. A description of each call's differences from its standard meaning (described in Appendix A) follows.

See Chapter 10 of this volume for more information on the High Sierra file system translator.

Table B-1 High Sierra FST ProDOS 16 calls

Meaningful		Not meaningful	
$06	GET_FILE_INFO	$01	CREATE
$08	VOLUME	$02	DESTROY
$10	OPEN	$04	CHANGE_PATH
$12	READ	$05	SET_FILE_INFO
$14	CLOSE	$13	WRITE
$16	SET_MARK	$15	FLUSH
$17	GET_MARK	$18	SET_EOF
$19	GET_EOF	$0B	CLEAR_BACKUP_BIT
$1C	GET_DIR_ENTRY	$22	ERASE_DISK
$20	GET_DEV_NUM	$24	FORMAT

With the exception of the FLUSH call, all calls on the right side of Table B-1 do nothing and return error $2B (write-protected). The FLUSH call also does nothing, but it returns no error (carry flag = clear).

The following sections describe how the High Sierra FST's handling of some of the calls listed on the left side of Table B-1 is different from standard ProDOS 16 practice. Calls listed on the left side of Table B-1 that are not described below are handled exactly as documented in Appendix A.

GET_FILE_INFO ($06)

GET_FILE_INFO returns certain attributes of an existing block file. The file may be open or closed.

Parameter differences

`fileType` This word output value equals $000F if the file is a directory; otherwise, it is $0000 (unknown)—unless the filename extension matches an entry in the file-type mapping table. See the FSTSpecific call description in Chapter 10, "The High Sierra FST."

`modDate` This word output value always has the same value as `createDate`.

`modTime` This word output value always has the same value as `createTime`.

`blocksUsed` This longword output value is always the same as the `totalBlocks` parameter returned from a Volume call.

VOLUME ($08)

Given the name of a block device, this call returns the name of the volume mounted in that device and other information about the volume.

Parameter differences

`freeBlocks` This longword output value is aways $0000.

GET_DIR_ENTRY ($1C)

This call returns information about a directory entry in the volume directory or a subdirectory. Before executing this call, the application must open the directory or subdirectory. The call allows the application to step forward or backward through file entries or to specify absolute entries by entry number.

The High Sierra FST does not allow READ calls and GET_DIR_ENTRY calls to the same reference number: if an open file has previously been accessed by GET_DIR_ENTRY, and a READ call is made with the same reference number, the High Sierra FST returns error $4E (invalid access). To avoid the error, open the directory twice.

Parameter differences

`fileType` This word output value equals $000F if the file is a directory; otherwise, it is $0000 ("unknown")—unless the filename extension matches an entry in the file-type mapping table. See the FSTSpecific call description in Chapter 10, "The High Sierra FST."

`modDateTime` This double longword output value always has the same value as `createDateTime`.

`auxType` This ongword output value is always $0000.

`fileSysID` This word output value is always $000B for High Sierra or $000C for ISO 9660. If it has any other value, the High Sierra FST returns error $52 (unsupported volume type).

The Character FST

The Character file system translator (Character FST) provides a file-system-like interface to character devices such as the console, printers, and modems.

Because the Character FST handles ProDOS 16 calls, all ProDOS 16 applications automatically have the capability of accessing character devices as files when running under GS/OS. ProDOS 16 itself does not provide that capability to ProDOS 16 applications.

The Character FST supports this subset of ProDOS 16 calls:

OPEN
NEWLINE
READ
WRITE
CLOSE
FLUSH

Attempting to send any other GS/OS ProDOS 16 call to a character device results in error $58 (not a block device).

See Chapter 11 for a general description of the Character FST.

OPEN ($10)

OPEN establishes an access path to the character file.

Parameter differences

`pathname`	This longword input pointer must point to a character device name.

Errors

In addition to the standard ProDOS 16 OPEN errors, the Character FST can return these errors from an OPEN call:

$26 driver no resources
$2F driver off line
$54 out of memory

READ ($12)

The READ call attempts to transfer the requested number of bytes from the specified character file into the application's data buffer.

Errors

In addition to the standard ProDOS 16 READ errors, the Character FST can return these errors from a READ call:

$23 driver not open
$2F driver off line
$53 parameter out of range
$54 out of memory

WRITE ($13)

The WRITE call attempts to transfer the requested number of bytes from the application's data buffer to the specified character file.

Errors

In addition to the standard ProDOS 16 WRITE errors, the Character FST can return these errors from a WRITE call:

$23 driver not open
$2F driver off line
$54 out of memory

CLOSE ($14)

The CLOSE call terminates access to the specified (by `refNum`) character file. CLOSE also involves flushing the file (see the FLUSH call) to ensure completion of all data transfer before a character file is closed.

Errors

In addition to the standard ProDOS 16 CLOSE errors, the Character FST can return these errors from a CLOSE call:

$23 driver not open
$2F driver off line

FLUSH ($15)

The FLUSH routine completes any pending data transfer to the character file specifed by `refNum`. If the character device is synchronous, all data transfer is by definition completed when the WRITE call returns, so the FLUSH routine simply returns with no error. If the device is asynchronous (such as interrupt-driven or DMA), the FLUSH routine waits until all data has been transferred, and then returns. If the file is multiply opened, all (output) access paths to the character file (not just the one with the specified `refNum`) are flushed.

Errors

In addition to the standard ProDOS 16 FLUSH errors, the Character FST can return these errors from a FLUSH call:

$23 driver not open
$2F driver off line

ProDOS 16 device calls

The only ProDOS-16 device call is D_INFO, which is handled only by the Device Manager—no FST can accept this call. Therefore, the standard description of D_INFO in Appendix A is the complete specification.

See the Introduction and Chapter 1 of Volume 2 for more general information on the Device Manager and GS/OS device calls.

Appendix C **The GS/OS Exerciser**

The GS/OS Exerciser is an application that allows you to "exercise" GS/OS by practicing all its calls from the keyboard. You can learn exactly how each GS/OS call works and what its results are before writing it into your programs. The GS/OS Exerciser is an excellent tool for learning the details of the application interface to GS/OS.

Starting the Exerciser

Before using the GS/OS Exerciser, be sure to make a copy and put the original in a safe place.

Warning! The Exerciser is a powerful program that does not protect you in any way from destroying data in memory or on any disk you can access. You can easily modify parts of memory that are already in use, causing a system crash. You can unintentionally overwrite critical data on disk, even a disk's directory. *Be careful how you use this program!*

Once the program is running, you see the main screen (Figure C-1). Note that the Exerciser uses a text-based display.

Figure C-1 Exerciser main screen

```
             GS/OS System Call Exerciser vXX.XX      10 Aug 1988
        Copyright 1987,1988  Apple Computer Inc.  All Rights Reserved
------------------------------------------------------------------------
 $01-Create          $0F-GetSysPrefs    $1B-GetLevel         $28-GetBootVol
 $02-Destroy         $10-Open           $1C-GetDirEntry      $29-Quit
 $03-OSShutdown      $11-Newline        $1D-BeginSession     $2A-GetVersion
 $04-ChangePath      $12-Read           $1E-EndSession       $2B-GetFSTInfo
 $05-SetFileInfo     $13-Write          $1F-SessionStatus    $2C-DInfo
 $06-GetFileInfo     $14-Close          $20-GetDevNumber     $2D-DStatus
 $08-Volume          $15-Flush          $21-GET_LAST_DEV     $2E-DControl
 $09-SetPrefix       $16-SetMark        $22-READ_BLOCK       $2F-DRead
 $0A-GetPrefix       $17-GetMark        $23-WRITE_BLOCK      $30-DWrite
 $0B-ClrBackupbit    $18-SetEOF         $24-Format           $31-BindInt
 $0C-SetSysPrefs     $19-GetEOF         $25-EraseDisk        $32-UnbindInt
 $0D-Null            $1A-SetLevel       $27-GetName          $33-FSTSpecific
 $0E-ExpandPath

 J - Make inline calls to GS/OS          K - Make class 0 calls to GS/OS
 L - Catalog a directory                 M - Modify the contents of memory
 N - Catalog $00 levels of a directory   P - Set minimum p_count for all calls
 Q - Quit back to caller                 R - Visit the Monitor
------------------------------------------------------------------------
 Select command:  $01
```

Call options

The GS/OS Exerciser can make almost any call an application makes, and in several different ways. Here are some of the options:

- **Stack/Inline system calls (J):** The Exerciser lets you make a call with either of two methods. The first is a stack-based call: you push the parameter buffer address and the command number onto the stack and then call the appropriate GS/OS entry point. The second method is the (more familiar) inline call: you call the appropriate GS/OS entry point and immediately follow with the command number and the parameter buffer pointer. (ProDOS 8 uses the inline call method.)

  In the Exerciser, you toggle between stack-based calls and inline calls by pressing **J.**

- **System call classes (K):** GS/OS includes the concept of call classes. Although up to eight classes are possible, only classes 0 (ProDOS 16-compatible calls) and 1 (standard GS/OS calls) are currently defined.

  By pressing **K** followed by either the arrow or number keys, you can select which class of call to make.

- **Maximum/Minimum parameter counts (P):** Many GS/OS calls accept a variable number of parameters. For each call, there is a minimum and a maximum permitted value for the parameter count (parameter `pCount`).

  By pressing **P** at either the main screen or the parameter-setup screen (see Figure C-2), you set the default `pCount` to either the minimum or maximum for the call being issued. (Only standard GS/OS calls use the parameter `pCount`.) Then, if you want something other than the minimum or maximum, you can reset `pCount` to the desired value at the parameter-setup screen.

The lower part of the main screen always displays the current settings for the method, class, and `pCount` options. The method and class are also displayed on the top line of the parameter-setup screen (see Figure C-2).

Making GS/OS calls

You make GS/OS calls from the Exerciser by entering call numbers on the main screen. The number you enter is displayed at the bottom of the screen. You can clear the number at any time by pressing zero twice in succession.

After entering the number, press the Return key. The parameter-setup screen for the call you selected is displayed (Figure C-2). Enter a value (or select the default provided by pressing the Return key) for each parameter; each time you press Return, the cursor moves downward one position in the parameter block. The cursor does not stop at any parameter that is a result-only value (that has no input value).

Figure C-2 Parameter-setup screen

```
$1C-Get Dir Entry              class 1 inline call               esc: main menu
-------------------------------------------------------------------------------
         p_count:      $000F    input
         ref_num:      $0006    input
        reserved:      $0000    result
            base:      $0000    input
    displacement:      $0001    input
     name_buffer:  $000146AA    result
FINDER.DEF

       entry_num:      $0001    result
       file_type:      $00C9    result
             eof:  $00000000    result
     blocks_used:  $00000038    result
   create          $57090100    result    Tu 22Dec87  901
   time and date:  $03000B15    result
   modification    $58113400    result    We 20Jan88 1752
   time and date:  $04000013    result
          access:      $00E3    result
        aux_type:  $00000100    result
     file_sys_id:      $0001    result
     option_list:  $00014850    result
-------------------------------------------------------------------------------
Press return to exit to main menu   Error $0000: call successful
```

Note: If, while you are entering parameters, you wish to abort the call, press the Escape key—it returns you to the main screen.

Pathnames and other text strings are passed to and from GS/OS in buffers referenced by pointers in the parameter blocks. Therefore, to enter or read a pathname, you must provide a buffer for GS/OS to read from or write to. In most cases, the Exerciser sets up a default buffer, pointed to by a default pointer parameter (see, for example, the Create call). The contents of the location referenced by that pointer are displayed on the screen, below the parameter block. For convenience, you can directly edit the displayed string on the screen; you needn't access the memory location itself.

After you have entered all the required parameters, press the Return key once more to execute the call. If everything has gone right, the parameter list now contains any results returned by GS/OS, and the message " $0000 call successful" appears at the bottom of the screen. If a GS/OS error occurs, the proper error number and message are displayed instead. In addition, if an error occurs, a small "c" appears at the lower right corner of the screen, which indicates that the microprocessor's carry bit has been set.

Other commands

The Exerciser has several other useful features.

- **List Directory (L,N):** There are two items on the main screen that help you catalog a disk. The first is the List command, which catalogs either a target directory or all online devices (see next item). The second is the **N** option, which allows you to specify how many levels to display of subdirectories and files within the target directory.

  From the main screen, select the levels you want by pressing **N** and then using the number keys or vertical arrow keys to specify the desired number of levels. You can select any number from $00 to $40. Press Return to enter your selection.

  Pressing **L** repeatedly toggles the List command between listing a directory and listing devices. Press **L** until "Catalog a directory" appears after "L - " on the main screen. Then press Return to execute the command (press Escape to abort it).

- **List Online Devices (L):** The List Online Devices command allows you quick access to the device numbers, device names, and volume names of any devices currently connected to the system.

  Pressing **L** repeatedly toggles the List command between listing a directory and listing devices. Press **L** until "List Devices Online" appears after "L - " on the main screen. Then press Return to execute the command (press Escape to abort it). The device-list screen appears (Figure C-3).

  `Dev #` on the screen is the actual hex value that you would use for `devNum` in the parameter list for a device call. `Device Name` and `Volume Name` are the names as known to the system. If the device is a drive with a volume that has been removed, the `Status` field will read "Offline".

Figure C-3 Device-list screen

```
L - List Devices Online                                          esc: main menu
--------------------------------------------------------------------------------

Dev #  Device Name                   Volume Name                          Status

$0001  .APPLEDISK3.5A                :SYSTEM.DISK
$0002  .APPLEDISK3.5B                :SYSTEM.TOOLS
$0003  .CONSOLE
$0004  .APPLEDISK5.25A                                                   Offline
$0005  .APPLEDISK5.25B                                                   Offline
$0006  .SCSI1                        :SCUSI
$0007  .DEV2
$0008  .DEV3

--------------------------------------------------------------------------------
Press return to continue:
```

- **Modify Memory (M):** By using the Modify Memory command, you can inspect and change the contents of any memory location.

  When you press **M** the Exerciser prompts you for a full three-byte address. Enter it and press Return; the Exerciser gives you an 80-column display of one memory page (256 bytes), with 16 bytes of data per line (Figure C-4). The page contains the address you entered, and the inverse-video cursor highlights the byte at that address. Using the arrow keys, you can move through the display; pressing > or < displays the next or previous page.

  To modify the contents of a memory location, move the cursor to it and retype the hexadecimal value you want it to contain. Table C-1 lists the hexadecimal values for all ASCII characters.

  Press **U** to undo a keypress that has modified the data in memory.

Figure C-4 Modify-memory screen

```
M - Modify the contents of memory                                   esc: main menu

    data_buffer:  $000146AA    value

01/4600- 20 20 20 20 00 00 00 00  9A 00 00 00 20 20 20 20      ........
01/4610- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/4620- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/4630- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/4640- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/4650- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/4660- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/4670- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/4680- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/4690- 20 20 20 20 20 20 00 00  00 00 9A 00 0A 00 46 49
01/46A0- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20        ........FI
01/46B0- 4E 44 45 52 2E 44 45 46  20 20 20 20 20 20 20 20    NDER.DEF
01/46C0- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/46D0- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/46E0- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20
01/46F0- 20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20

Commands: arrow keys, >, <, U, 0..F
```

- **Visit the Monitor (R):** The Monitor program is a firmware tool for debugging and executing programs. It is described in the *Apple IIGS Firmware Reference*. With the Monitor, you can inspect and modify the contents of memory, assemble and disassemble code in a limited manner, and execute code in memory.

  You can temporarily leave the Exerciser to use the Monitor program by pressing **R** from the main screen. The command functions exactly like the Control Panel "Visit Monitor" command and the BASIC command "call -151". When you are ready to return to the Exerciser, press Control-Y.

- **Quit the Exerciser (Q):** To leave the Exerciser—and return to the Finder or other startup program—press **Q** from the main screen. Of course, you can also quit by selecting the GS/OS Quit call ($29 on the main screen), filling out its parameters on the parameter-setup screen, and executing it.

Table C-1 ASCII table

A = ASCII character; D = decimal value; H = hexadecimal value; B = binary value

A	D	H	B	A	D	H	B	A	D	H	B	A	D	H	B
nul	0	0	00000000	sp	32	20	00100000	@	64	40	01000000	`	96	60	01100000
soh	1	1	00000001	!	33	21	00100001					a	97	61	01100001
stx	2	2	00000010	"	34	22	00100010	A	65	41	01000001	b	98	62	01100010
etx	3	3	00000011					B	66	42	01000010	c	99	63	01100011
eot	4	4	00000100	#	35	23	00100011	C	67	43	01000011				
				$	36	24	00100100	D	68	44	01000100	d	100	64	01100100
enq	5	5	00000101	%	37	25	00100101	E	69	45	01000101	e	101	65	01100101
ack	6	6	00000110	&	38	26	00100110					f	102	66	01100110
bel	7	7	00000111	'	39	27	00100111	F	70	46	01000110	g	103	67	01100111
bs	8	8	00001000					G	71	47	01000111	h	104	68	01101000
ht	9	9	00001001	(	40	28	00101000	H	72	48	01001000				
				)	41	29	00101001	I	73	49	01001001	i	105	69	01101001
lf	10	A	00001010	*	42	2A	00101010	J	74	4A	01001010	j	106	6A	01101010
vt	11	B	00001011	+	43	2B	00101011					k	107	6B	01101011
ff	12	C	00001100	,	44	2C	00101100	K	75	4B	01001011	l	108	6C	01101100
cr	13	D	00001101					L	76	4C	01001100	m	109	6D	01101101
so	14	E	00001110	-	45	2D	00101101	M	77	4D	01001101				
				.	46	2E	00101110	N	78	4E	01001110	n	110	6E	01101110
si	15	F	00001111	/	47	2F	00101111	O	79	4F	01001111	o	111	6F	01101111
dle	16	10	00010000	0	48	30	00110000					p	112	70	01110000
dc1	17	11	00010001	1	49	31	00110001	P	80	50	01010000	q	113	71	01110001
dc2	18	12	00010010					Q	81	51	01010001	r	114	72	01110010
dc3	19	13	00010011	2	50	32	00110010	R	82	52	01010010				
				3	51	33	00110011	S	83	53	01010011	s	115	73	01110011
dc4	20	14	00010100	4	52	34	00110100	T	84	54	01010100	t	116	74	01110100
nak	21	15	00010101	5	53	35	00110101					u	117	75	01110101
syn	22	16	00010110	6	54	36	00110110	U	85	55	01010101	v	118	76	01110110
etb	23	17	00010111					V	86	56	01010110	w	119	77	01110111
can	24	18	00011000	7	55	37	00110111	W	87	57	01010111				
				8	56	38	00111000	X	88	58	01011000	x	120	78	01111000
em	25	19	00011001	9	57	39	00111001	Y	89	59	01011001	y	121	79	01111001
sub	26	1A	00011010	:	58	3A	00111010					z	122	7A	01111010
esc	27	1B	00011011	;	59	3B	00111011	Z	90	5A	01011010	{	123	7B	01111011
fs	28	1C	00011100					[	91	5B	01011011	|	124	7C	01111100
gs	29	1D	00011101	<	60	3C	00111100	\	92	5C	01011100				
				=	61	3D	00111101	]	93	5D	01011101	}	125	7D	01111101
rs	30	1E	00011110	>	62	3E	00111110	^	94	5E	01011110	~	126	7E	01111110
us	31	1F	00011111	?	63	3F	00111111					del	127	7F	01111111
								_	95	5F	01011111				

Appendix D **GS/OS System Disks and Startup**

This appendix lists the directories and principal files that make up a GS/OS system disk for the Apple IIgs computer. A typical system disk has all of these files plus others, which may be applications, desk accessories, utilities, initialization files, documents, or other data files.

In some very restricted instances, it may be possible to fit an application and its required system files onto a 800K diskette; most applications, however, require two 800K diskettes.

Application system disks

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

Table 2-1 shows the files that must be present on all application system disks.

Table D-1 Directories and files on a GS/OS system disk

Directory/File	Contents
Prodos	Required: A simple loader that loads the START.GS.OS file and executes it
Appletalk/	Contains AppleTalk setup files
Icons/	Contains Finder-related information
Finder.icons	Icons used by the Finder
Finder.def	Data used by the Finder
System/	Required: Contains GS/OS and other important system files
GS.OS	Required: The GS/OS operating system and the System Loader
START.GS.OS	Required: The GS/OS loader and program dispatcher
FSTS/	Required: Contains all File System Translators
System.setup/	Required: Contains setup files that execute at system startup
Tool.setup	Required: Initializes tool sets at startup
Drivers/	Contains GS/OS device drivers
Tools/	Contains RAM-based tool sets: required if RAM-based tools are needed
Fonts/	Contains font files: required if fonts are needed
Desk.accs/	Contains desk accessories: required if desk accessories are provided
Start	The program automatically executed at startup; this should usually be the Finder
Error.msg	Required: GS/OS error messages
P8	Required if ProDOS 8 applications will be run from GS/OS

System startup from ProDOS volumes

Disk blocks 0 and 1 on an Apple IIGS system disk contain the boot code. The boot code is functions identically for ProDOS 8, ProDOS 16 and GS/OS system disks. This allows ProDOS 8 system disks to boot on an Apple IIGS, and it also means that the initial part of the bootstrap procedure is identical for all three operating systems.

First, the boot firmware in ROM reads the boot code (blocks 0 and 1) into memory and executes it. For a system disk with a volume name represented by */,

1. The boot code searches the disk's volume directory for the first file named PRODOS with the file type $FF.
2. If the file is found, it is loaded and executed.

From this point on, the three operating systems behave differently. On a ProDOS 8 system disk, the file named PRODOS is the ProDOS 8 operating system. On a ProDOS 16 system disk, the PRODOS file is not the operating system itself; it is the operating system loader and program dispatcher. On a GS/OS system disk, the PRODOS file contains only a startup routine and file-system-specific routines that are used by the operating system loader and program dispatcher. The operating system loader and program dispatcher are contained in the file */SYSTEM/START.GS.OS.

When it receives control from the boot code, */PRODOS performs the following tasks -

1. Checks to make sure it's running on an Apple IIGS with ROM version 01 or greater.
2. Loads the file */SYSTEM/START.GS.OS.

   The START.GS.OS file is divided into two parts: GLoader and GQuit. GLoader is the operating system loader. It's temporary and is used only during system startup. GQuit is the program dispatcher. It contains the code used for starting and quitting ProDOS 8 and GS/OS applications.
3. Transfers control to GLoader.

When it receives control, GLoader performs the following tasks:

- Puts up the GS/OS splash screen and initializes the Apple IIGS tools and the Memory Manager.
- Relocates the GS/OS program dispatcher to an area in memory where it will reside permanently and relocates part of the */PRODOS file to an area in memory where it will reside permanently.
- Gets the name of the boot volume and the name of the start FST.
- Loads the GS/OS operating system and Apple IIGS System Loader (file */SYSTEM/GS.OS) and then installs the System Loader.
- Loads the file */SYSTEM/ERROR.MSG.

- Loads the start FST. The start FST must reside in the */SYSTEM/FSTS subdirectory, must have a file type of $BD, and must have the high bit of its auxiliary type set to 0.
- Initializes GS/OS and installs the start FST.
- Loads and installs the rest of the FSTs in the */SYSTEM/FSTS subdirectory. The files must be Apple IIGS load files of type $BD. If bit 15 of a file's auxiliary type is 1, the FST is not loaded.
- Sets prefix 0 to the boot volume name, and prefix 2 to */SYSTEM/LIBS.
- GLoader selects the application to run by taking the following steps:
  a. It first looks for a type $B3 file named */SYSTEM/START. Typically, that file should be the Finder, but it could be any Apple IIGS application. If START is found, it is selected.
  b. If there is no START file, GLoader searches the boot volume directory for a file that is either one of the following types:
     - a ProDOS 8 system program (type $FF) with the filename extension .SYSTEM
     - a GS/OS application (type $B3) with the filename extension .SYS16

  Whichever is found first is selected.

  Note If a ProDOS 8 system program is found first, but the ProDOS 8 operating system (file */SYSTEM/P8) is not on the boot volume, GLoader then searches for and selects the first ProDOS 16 application.
- Executes the file */SYSTEM/SYSTEM.SETUP/TOOL.SETUP. The TOOL.SETUP file must have file type $B6, and executes, in turn, every file (other than TOOL.SETUP) that it finds in the */SYSTEM/SYSTEM.SETUP subdirectory. The files must be Apple IIGS load files of type $B6 or $B7. If the high bit of a file's auxiliary type is 1, the setup file is not executed.
- Installs all desk accessories it finds in the */SYSTEM/DESK.ACCS subdirectory. The files must be Apple IIGS load files of type $B8 or B9. If Bit 15 of a file's auxiliary type is 1, the desk accessory is not loaded.

Finally, GLoader makes a standard GS/OS Quit call to launch the selected application. It is GQuit, not GLoader, that actually loads and launches the selected application.

System startup from non-ProDOS volumes

GS/OS supports booting from non-ProDOS volumes. Special boot blocks have to be written out to the boot volumes, as well as a boot file containing the startup routine and the file-system-specific routines required by GLoader and GQuit. The boot file is a replacement for the file PRODOS, which is used when booting from ProDOS volumes.

The boot blocks must load the boot file at location $2000 in bank $00 and then execute the boot file by doing a JMP $2000. The boot blocks must make sure that MSLOT ($07f8) is set up to contain the slot number of the boot device since this value will be needed by the boot file and GLoader. The boot file must contain the following routines: Startup, ReadInFile, GetBootName and GetFstName. These routines are described in the following sections.

The boot file must begin with a jump table that looks like this:

```
jump_table          start
                    jmp     startup                    ; 3 bytes
                    nop                                ; 1 byte of padding
                    dc      i2'readinfile'             ; offset into table = 4
                    dc      i2'getbootname'     ; offset into table = 6
                    dc      i2'getfstname'             ; offset into table = 8
                    dc      i2'xxxx-jump_table'        ; offset into table = 10
aux_value           ds      2                          ; offset into table = 12
                    end
```

The jump table must be the first thing in the boot file so that when the boot file is loaded, the table begins at location $2000. GLoader and GQuit use the table to call the routines in the boot file.

The entry at offset 10 must contain the size, in bytes, of the permanent part of the boot file. The permanent part of the boot file consists of the jump table, the ReadInFile routine, the GetBootName routine and any internal routines and/or data required by ReadInFile and GetBootName. The Startup and GetFstName routines are only used during boot time and so are temporary.

The boot file must be organized with the permanent code and data at the beginning of the load file and the temporary code and data at the end of the load file. GQuit uses the size specified in the jump table to determine how much of the boot file (beginning at location $2000) to save in memory for later use. When GQuit is quitting from a ProDOS 8 application into a GS/OS application, it needs to reload GS/OS. In order to do this, it relocates the saved portion of the boot file to location $2000, calls the GetBootName routine to verify that the boot volume is in the boot drive, and then calls the ReadInFile routine to read in the necessary files.

The entry at offset 12 must be set up by the Startup routine to contain the auxiliary type of the START.GS.OS file. GLoader uses this value when it puts up the splash screen.

Startup (boot file routine)

The Startup routine must perform the following tasks:

1. Determine that it is running on an Apple IIGS with ROM version 01 or greater, and if not, report a fatal error.
2. Set the e, m, and x flags in the processor status register to zero to enable full native mode.

3. Set the bank register to $00, set the direct register to $0000, and set the stack register to $01FF.
4. Obtain the boot slot number from MSLOT ($07f8) and save it in the permanent code area for later use by ReadInFile and GetBootName. Note that GLoader also uses MSLOT, so its contents must still be valid when control is transferred to GLoader.
5. Load the file */SYSTEM/START.GS.OS at location $6800 in bank $00.
6. Store the auxiliary type of the START.GS.OS file at offset 12 in the jump table.
7. Transfer control to GLoader by doing a JMP $6800.

The startup routine from the ProDOS boot file is included in this appendix as an example.

ReadInFile (boot file routine)

This routine finds the requested file, reads it into memory at the location specified, and returns the eof, file type and auxiliary type. The pathname of the requested file is returned as a GS/OS string; that is, it begins with a length word and the filenames are separated by colons. There is no leading or trailing colon.

The pathname does not include the volume name since this routine is called only to read from the boot volume. Also, the Startup routine should have saved the boot slot number in the permanent data area. For example, to load the file */SYSTEM/GS.OS, GLoader will call this routine with the partial pathname "SYSTEM:GS.OS".

Entry and exit are in full native mode. The direct register, data bank register, and language card state must be preserved.

The input parameters are as follows:

4 bytes	space for EOF result
2 bytes	space for auxiliary type result
2 bytes	space for file type result
4 bytes	pointer to partial pathname of file to read
4 bytes	pointer to buffer to read file into
2 bytes	return address

The output parameters are as follows:

4 bytes	EOF
2 bytes	auxiliary type

2 bytes file type

c = 0 if successful, c = 1 if error

A - contains error code if c = 1

GetBootName (boot file routine)

The GetBootName routine returns the name of the boot volume. The returned string must begin with a length word and must contain a leading colon but not a trailing colon. The maximum length of the volume name is 32 characters. If the volume name is longer than 32 characters, an error should be returned.

Entry and exit are in full native mode. The direct register, data bank register, and language card state must be preserved.

The input parameters are as follows:

4 bytes pointer to space for volume name

2 bytes return address

The output parameters are as follows:

c = 0 if successful, c = 1 if error

A - contains error code if c = 1

GetFSTName (boot file routine)

The GetFstName routine returns the filename of the FST that is associated with this boot file. For example, the ProDOS boot file returns the name "PRO.FST", which is the filename of the ProDOS FST. The returned string must begin with a length word and must not contain any separators. The maximum allowed length of the filename is 32 characters.

Entry and exit are in full native mode. The direct register, data bank register and language card state must be preserved.

The input parameters are as follows:

4 bytes pointer to space for FST name

2 bytes return address

The output parameters are as follows:

none

Sample boot file startup routine

The following sample code shows part of the ProDOS boot file startup routine.

```
startup     start
            using   pb_data

            longa   on
            longi   on

; At onset, we don't know what machine we are being run on.
; If we're not being run on an Apple //GS we must hang with an
; error message.

; The following code will run on both the 65816 and 6502
; processors to ensure 8-bit processing.

            tsx                             ;save stack pointer
            lda     #$3030
            nop                             ;required filler
            pha                             ;push $30 on stack
            plp                             ;and retrieve for full 8-bit mode

            longa   off
            longi   off

            txs                             ;restore stack

; The above code looks like the following for a 6502...
; (this code essentially does nothing on a 6502)

; tsx
; lda  #$30
; bmi  nop   ;will never be taken
; pha
; plp
; txs

            lda     romin                   ;bank in rom
            sec                             ;go into //GS id routine with c set
            jsr     idroutine               ;are we on a //GS?
            bcs     show_err                ;no
            cpy     #$01                    ;is rom revision 01 or greater?
            bcc     show_err                ;no

; At this point we must be in emulation mode on a //GS.

            pea     $0000                   ;ensure direct page at $0000
            pld

            phk                             ;set data bank to bank $00
            plb
```

```
;The id_sp routine reads MSLOT and sets up information used by ReadInFile

            jsr     id_sp

            clc
            xce                             ;begin native mode
            rep     #$30                    ;begin 16-bit mode
            longa   on
            longi   on

            lda     #$01ff                  ;set stack to $01ff
            tcs

; Now read in the */SYSTEM/START.GS.OS file.

            pha                             ;push 8 bytes for results
            pha
            pha
            pha
            pea     start_path|-16          ;push pointer to partial pathname
            pea     start_path              ;of file to be loaded
            pea     0                       ;push address of where file should
            pea     start_loc               ;be loaded
            jsr     readinfile              ;find the file and read it in
            bcc     read_ok                 ;branch if no error
            pha                             ;push error number
            pea     $0000                   ;push address of
            pea     startup_err             ;error message
            ldx     #$1503                  ;call SysFailMgr tool call to
            jsl     $e10000                 ;report the error - doesn't return

read_ok     anop
            pla                             ;ignore filetype
            pla                             ;get auxtype
            sta     |aux_value              ;and store at end of jump table
            pla                             ;ignore eof
            pla

; Now transfer control to START.GS.OS

            jmp     start_loc

;-----------------------------------------------------------------------

show_err    anop
            longa   off
            longi   off

; Enter this routine with c=1 for wrong system error.
; Enter this routine with c=0 for wrong rom error.

            php                             ;save 'c' around setup stuff
            lda     romin                   ;rom in for monitor's home routine
            sta     clr80vid                ;disable 80 column hardware
            sta     clraltchar              ;switch in primary character set
            sta     clr80col                ;disable 80 column store
```

```
            jsr     init                ;text pg 1, text mode, 40 col window
            jsr     setvid              ;does a 'pr#0' (puts in coutl in csw)
            jsr     setnorm             ;white chars on black background
            jsr     home                ;clear screen
            plp                         ;which message gets shown?

            ldy     not_a_gs            ;get length of message
            bcs     print_it
            ldy     wrong_rom           ;get length of message

print_it    anop
            lda     not_a_gs,y          ;get character
            bcs     print_it2
            lda     wrong_rom,y         ;get character

print_it2   anop
            sta     screen,y            ;store directly to screen
            dey
            bne     print_it

hang        jmp     hang                ;done

            end

;-----------------------------------------------------------------

pb_data     data

;firmware entry points

idroutine   equ     $fe1f               ;//GS id routine
setvid      equ     $fe93               ;reset output to screen
setnorm     equ     $fe84               ;normal white text on blk background
init        equ     $fb2f               ;text pg 1, text mode, 40 col window
home        equ     $fc58               ;home cursor and clear to end of screen

;soft switches

clr80col    equ     $c000               ;disable 80 column store
clr80vid    equ     $c00c               ;disable 80 colume hardware
clraltchar  equ     $c00e               ;normal lc, flashing uc
romin       equ     $c081               ;enable rom read

;misc. equates

screen      equ     $05a8               ;left center of 40 column screen
mslot       equ     $07f8               ;slot # of boot device
start_loc   equ     $006800             ;where START.GS.OS is loaded

;strings

start_path  dc      i2'18'
            dc      c'SYSTEM:START.GS.OS'

fst_name    dc      i2'7'
            dc      c'PRO.FST'          ;name of start FST
```

```
startup_err dc      i1'40'
            dc      c'Unable to load START.GS.OS file. Error=$'

            msb     on

not_a_gs    dc      i1'35'
            dc      c'GS/OS REQUIRES APPLE IIGS HARDWARE '

wrong_rom   dc      i1'38'
            dc      c'GS/OS needs ROM version 01 or greater '

            end
```

Appendix E **Apple Extensions to ISO 9660**

This appendix describes a protocol through which file-type information can be added to CD-ROM files or other files in the ISO 9660 format (which does not recognize file typing). With this protocol, ProDOS and Macintosh files can be stored on compact discs—as valid ISO 9660 files—while retaining all information related to file type.

You may need to read this appendix if you are

- an Apple Developer working with ISO 9660
- a publisher of authoring tools for ISO 9660 discs
- a publisher of ISO 9660 discs
- a publisher of ISO 9660 receiving system software

High Sierra support: ISO 9660 is the international file system standard for CD-ROM; it is based on the original High Sierra format, but is not identical to it. The protocol described in this appendix is meant to apply to the ISO 9660 file system; however, the High Sierra FST (See Chapter 10 of this volume) supports the protocol for High Sierra-formatted files also.

What the Apple extensions do

Creating an ISO 9660 CD-ROM disc containing ProDOS files or Macintosh hierarchical file system (HFS) files can have great advantages: the large storage capacity of compact discs means cost savings and greater convenience when distributing large amounts of data, and the position of ISO 9660 as an international standard means that the files will be accessible on a large variety of machines. Unfortunately, both the HFS and ProDOS file systems require information that the ISO 9660 file system does not support: ProDOS requires a file type and an auxiliary file type, and HFS requires a file type, a file creator, and, frequently, an icon resource.

This appendix defines a protocol that extends the ISO 9660 specification. The protocol is designed to both solve existing compatibility problems and allow for future expansion; at present, it has two principal features:

- It permits inclusion of HFS-specific or ProDOS-specific information in files, without corrupting the ISO 9660 structures. Discs created using the protocol are valid ISO 9660 discs and should function normally on non-Apple receiving systems.
- It defines a mechanism for preserving filenames across translations form ProDOS to ISO 9660 and back, and gives suggestions for optimum translations of Macintosh filenames.

The protocol uses the `systemIdentifier` field in the Primary Volume Descriptor for global information, and the `systemUse` field in the directory record for file-specific information.

The protocol identifier

Discs that have been formatted with the Apple extensions to ISO 9660 are identified by their **protocol identifier,** which has the following characteristics:

Location: `SystemIdentifier` field in the Primary Volume Descriptor.

Size: 32 bytes. It is the entire contents of the `SystemIdentifier` field.

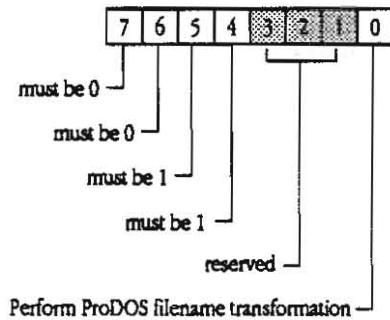
Contents: "APPLE COMPUTER, INC., TYPE: " followed by the **protocol flags.** In hexadecimal, the protocol identifier looks like this:

```
41 50 50 4C 45 20 43 4F 4D 50 55 54 45 52 2C 20
49 4E 43 2E 2C 20 54 59 50 45 3A 20 3x 3x 3x 3x
```

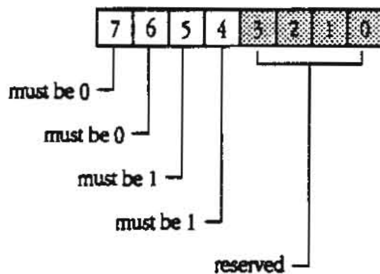
The protocol identifier is considered valid if its first 28 bytes match the first 28 characters above.

Protocol flags: 4 bytes of nibble-encoded information (represented as "3x" in the previous example). Nibble encoding is necessary in order to guarantee that the bytes represent legal ISO 9660 ***a-characters*** (printable characters). The flag bytes are numbered 0–3; flag-byte 0 is the byte following the space ($20). The bits of each flag byte are numbered 0–7, 0 being the least significant. The flag bytes are presently defined as follows:

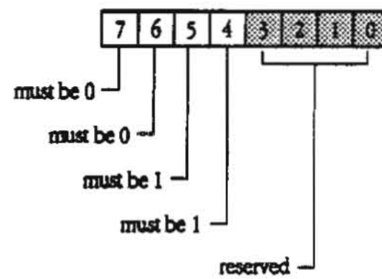
flag-byte 0:



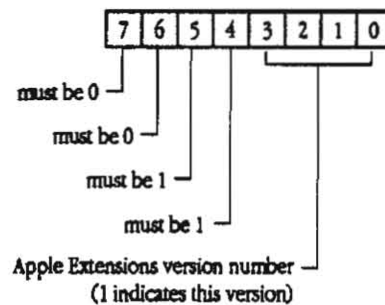
flag-byte 1:



flag-byte 2:



flag-byte 3:



The Directory Record SystemUse Field

Directory records in the ISO 9660 specification have the following format:

```
byte        DirectoryRcdLength
byte        XARlength
struct      ExtentLocation
struct      DataLength
struct      RecordingDateTime
byte        FileFlags
byte        FileUnitSize
byte        InterleaveGapSize
long        VolumeSequenceNum
byte        FileNameLength
char        FileName[FileNameLength]
byte        RecordPad
char        SystemUse[SystemUseLength]
```

The `RecordPad` field is present only if needed to make `DirectoryRcdLength` an even number. If `RecordPad` is present, its value must be zero ($00).

The `SystemUse` field is an optional field; if it is present, its length (equal to `SystemUseLength`) must be an even number.

The `SystemUse` field, when present, must begin with a signature word , followed by a one-byte `SystemUseID`, followed by file-specific information. The signature word allows a receiving system to ensure that it can interpret the following data correctly, and the `SystemUseID` determines the type and format of the information that follows.

The Apple signature word (`AppleSignature`) is defined as "B A" ($42 41).

Receiving systems must perform a simple calculation to determine if the `SystemUse` field is present in any given directory record. It is present if

`DirectoryRcdLength - FileNameLength` > 34

Receiving systems should first verify that the `SystemUse` field is present, then check for `AppleSignature` before interpreting the `SystemUseID`.

SystemUseID

`SystemUseID` can have the values shown in Table E-1.

Table E-1 Defined values for `SystemUseID`

Value	Meaning
$00	(reserved)
$01	ProDOS file_type and aux_type follow
$02	HFS fileType and fileCreator follow
$03	HFS fileType, fileCreator follow (bundle bit set)
$04	HFS fileType, fileCreator, and ICN# resource (128-byte icon) follow
$05	HFS fileType, fileCreator, ICN# resource follow (bundle bit set)
$06	HFS fileType, fileCreator, Finder flags follow
$07-FF	(reserved)

Table E-2 defines the contents of the `SystemUse` field for each defined value of `SystemUseID`.

Table E-2 Contents of `SystemUse` field for each value of `SystemUseID`

Offset	Contents
SystemUseID=01 (ProDOS):	
$00-01	$42 41 (AppleSignature)
$02	$01 (SystemUseID)
$03	ProDOS file type
$04-05	ProDOS aux type (LSB-MSB)*
SystemUseID=02 (HFS):	
$00-01	$42 41 (AppleSignature)
$02	$02 (SystemUseID)
$03-06	HFS fileType (MSB-LSB)
$07-0A	HFS fileCreator (MSB-LSB)*
$0B	(Padding for even length)

SystemUseID=03 (HFS, bundle bit set):

$00-01	$42 41 (AppleSignature)
$02	$03 (SystemUseID)
$03-06	HFS fileType (MSB-LSB)*
$07-0A	HFS fileCreator (MSB-LSB)*
$0B	(Padding for even length)

SystemUseID=04 (HFS, icon):

$00-01	$42 41 (AppleSignature)
$02	$04 (SystemUseID)
$03-06	HFS fileType (MSB-LSB)*
$07-0A	HFS fileCreator (MSB-LSB)*
$0B-8A	HFS ICN# resource (MSB-LSB)*
$8B	(Padding for even length)

SystemUseID=05 (HFS, icon, bundle bit set):

$00-01	$42 41 (AppleSignature)
$02	$05 (SystemUseID)
$03-06	HFS fileType (MSB-LSB)*
$07-0A	HFS fileCreator (MSB-LSB)*
$0B-8A	HFS ICN# resource (MSB-LSB)*
$8B	(Padding for even length)

*SystemUseID=06 (HFS, Finder flags)**:*

$00-01	$42 41 (AppleSignature)
$02	$05 (SystemUseID)
$03-06	HFS fileType (MSB-LSB)*
$07-0A	HFS fileCreator (MSB-LSB)*
$0B-0C	HFS Finder flags (MSB-LSB)*

*(MSB-LSB) = the most significant byte occupies the lowest address, the least significant byte, the highest address; (LSB-MSB) = the least significant byte occupies the lowest address, the most significant byte, the highest address.

**To fill the Finder flags field here, premastering software can simply copy the finder flags as retrieved by the HFS call GetFInfo. Only bits 5 (always switch-launch), 12 (system file), 13 (bundle bit), and 15 (locked) are used. All other bits are either ignored or always set by the FST. See Macintosh technical note #40 for more details about the Finder flags.

Filename transformations

The rules governing permissible filenames are different under ISO 9660 than under either ProDOS or Macintosh HFS. Therefore, one problem with putting ProDOS or HFS files on an ISO 9660 disc is how to rename them. Ideally there should be a simple, reversible transformation that can be applied to a filename to make it a legal ISO 9660 name, and reversed to restore the original ProDOS or HFS name.

Such a transformation exists for ProDOS and is given here. There is none for HFS, but guidelines to minimize changes during transformation are listed.

ProDOS

Legal ProDOS filenames differ from legal filenames under ISO 9660 in these ways:

- ProDOS filenames allow multiple periods; ISO 9660 filenames do not.
- ISO 9660 requires that both of the separators *period* (.) and *semicolon* (;) occur in each filename, and that the semicolon be followed by a version number. (This requirement is for nondirectory files only.)

The following steps constitute a reversible transformation that preserves ProDOS filename syntax. That means that an authoring tool can apply the transformation to any ProDOS file to get a legal ISO 9660 filename, and that a receiving system can reverse the transformation to hide from an application the fact that a transformation has occurred. A user can therefore access the file using its original ProDOS filename.

When creating an ISO 9660 disc from ProDOS source files, the authoring tool must perform the following transformation on *all* filenames:

1. Replace all periods in the ProDOS filename with underscores. If the file is a directory file, that completes the transformation.
2. If the file is not a directory file, append the characters ".;1" to the filename. It is now a valid ISO 9660 filename.

After all filenames have been transformed, the authoring tool must set the ProDOS transformation bit in the protocol identifier, described earlier in this appendix.

Table E-3 shows some examples of the transformation.

Table E-3 ProDOS-to-ISO 9660 filename transformations

ProDOS filename	kind of file	ISO 9660 filename
PRODOS	standard	PRODOS.;1
BASIC.SYSTEM	standard	BASIC_SYSTEM.;1
SYSTEM	directory	SYSTEM
DESK.ACCS	directory	DESK_ACCS
START.GS.OS	standard	START_GS_OS.;1

Volume name: The ProDOS volume name becomes the ISO 9660 Volume Identifier in the Primary Volume Descriptor. It is a filename and, therefore, must be transformed like other ProDOS filenames. It must be transformed as a directory name (periods replaced with underscores).

In use, the receiving system can inspect the ProDOS transformation bit in the protocol identifier, and handle the necessary conversions such that the original ProDOS filenames can be used to refer to all files and directories on the volume. The receiving system performs the above transformation on user-supplied filenames before searching for them on disc, and reverses the transformation before presenting filenames to the user.

Remember that this transformation cannot be done on a file-by-file basis; it must be applied to every file and directory on a disc.

Macintosh HFS

Because HFS file naming rules are very flexible, most HFS filenames are illegal in the ISO 9660 specification. Furthermore, no reversible transformation is possible without degrading performance; unlike with ProDOS, there is no simple conversion from all valid Macintosh HFS filenames to valid ISO 9660 filenames. To make the transformations as consistent as possible, however, Apple recommends that authoring tools and receiving systems follow these guidelines when performing HFS-to-ISO 9660 transformations:

1. Convert all lowercase characters to uppercase.
2. Replace all illegal characters, including periods, with underscrores.
3. If the filename needs to be shortened, truncate the rightmost characters.
4. If the file is not a directory file, append the characters ".;1" to the filename.

Such a transformation is not reversible, but if it is followed the results, will at least be consistent across all files and discs.

ISO 9660 associated files

An associated file under ISO 9660 is analogous to the resource fork of an HFS file. The format of associated files is defined in the ISO 9660 specification; the Apple extensions do not change the format in any way. For clarity, however, this section restates the definition and gives an example.

An associated file has these characteristics:

- It is one of two identically named files in a directory; the associated file has exactly the same file identifier as its counterpart.
- It resides immediately before its counterpart in the directory.
- It has the associated bit set in the file flags byte of the directory record.

The associated file is equivalent to the resource fork of an HFS file; its counterpart is equivalent to the data fork of the same HFS file.

For example, if the file "ANYFILE.;1" has an associated file, two adjacent directory records will be named "ANYFILE.;1". The first one (the resource fork) will have the associated bit set, the second one (the data fork) will have the associated bit clear.

Appendix F **GS/OS Error Codes and Constants**

This appendix lists and describes the the errors that an application can receive as a result of making a GS/OS call.

Column 1 in Table F-1 lists the GS/OS error codes that an application can receive. Column 2 lists the predefined constants whose values are equal to the error codes; the constants are defined in the GS/OS interface files supplied with development systems. Column 3 gives a brief description of what each error means.

Table F-1 GS/OS errors

Code	Constant	Description
$01	`badSystemCall`	bad GS/OS call number
$04	`invalidPcount`	parameter count out of range
$07	`gsosActive`	GS/OS is busy
$10	`devNotFound`	device not found
$11	`invalidDevNum`	invalid device number (request)
$20	`drvrBadReq`	invalid request
$21	`drvrBadCode`	invalid control or status code
$22	`drvrBadParm`	bad call parameter
$23	`drvrNotOpen`	character device not open
$24	`drvrPriorOpen`	character device already open
$25	`irqTableFull`	interrupt table full
$26	`drvrNoResrc`	resources not available
$27	`drvrIOError`	I/O error
$28	`drvrNoDevice`	no device connected
$29	`drvrBusy`	driver is busy
$2B	`drvrWrtProt`	device is write protected
$2C	`drvrBadCount`	invalid byte count
$2D	`drvrBadBlock`	invalid block address
$2E	`drvrDiskSwitch`	disk has been switched

Table F-1 GS/OS errors (continued)

Code	Constant	Description
$2F	drvrOffLine	device off line or no media present
$40	badPathSyntax	invalid pathname syntax
$43	invalidRefNum	invalid reference number
$44	pathNotFound	subdirectory does not exist
$45	volNotFound	volume not found
$46	fileNotFound	file not found
$47	dupPathname	create or rename with existing name
$48	volumeFull	volume full error
$49	volDirFull	volume directory full
$4A	badFileFormat	version error (incompatible file format)
$4B	badStoreType	unsupported (or incorrect) storage type
$4C	eofEncountered	end-of-file encountered
$4D	outOfRange	position out of range
$4E	invalidAccess	access not allowed
$4F	buffTooSmall	buffer too small
$50	fileBusy	file is already open
$51	dirError	directory error
$52	unknownVol	unknown volume type
$53	paramRangeErr	parameter out of range
$54	outOfMem	out of memory
$57	dupVolume	duplicate volume name
$58	notBlockDev	not a block device
$59	invalidLevel	specified level outside legal range
$5A	damagedBitMap	block number too large
$5B	badPathNames	invalid path names for ChangePath
$5C	notSystemFile	not an executable file

stack: A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order. The term *the stack* usually refers to the particular stack pointed to by the 65C816 **stack pointer.**

standard Apple II: Any Apple II computer that is not an Apple IIGS. Since previous members of the Apple II family share many characteristics, it is useful to distinguish them as a group from the Apple IIGS. A standard Apple II may also be called an *8-bit Apple II,* because of the 8-bit registers in its 6502 or 65C02 microprocessor.

standard file: A named collection of data consisting of a single sequence of bytes. Compare **extended file, directory file.**

standard GS/OS calls: Also called *class 1 calls* or simply *GS/OS calls,* the primary set of **application-level calls** in GS/OS. They provide the full range of GS/OS capabilities accessible to applications. Besides GS/OS calls, the other application-level calls available in GS/OS are **ProDOS 16-compatible calls.**

System Loader: the program that loads all other programs into memory and prepares them for execution.

system service calls: Low-level calls in a common format used by internal components of GS/OS (such as FSTs), and also between GS/OS and device drivers.

unclaimed interrupt: An interrupt that is not recognized and acted upon by any interrupt handlers.

volume name: The name of the volume directory file on a disk or other medium. All pathnames on a volume start with the volume name. Volume names follow the same rules as other filenames, except that a volume name always starts with a pathname separator.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS computer when running a standard Apple II program.). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**