For GS/OS
System Software
Version 5.0

Apple® IIGS® GS/OS®
Device Driver Reference

APPLE COMPUTER, INC.

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

Simultaneously published in the United States and Canada.

Contents

Figures and tables

Introduction **The Device Level in GS/OS**

One of the principal goals of GS/OS® is to provide application programmers with access to a wide variety of hardware devices while insulating programmers (and users) from the low-level details of hardware control. The device level in GS/OS is responsible for meeting this goal.

The device level consists of

- the GS/OS interface to FSTs for device access through file systems
- the GS/OS interface to applications for direct device access
- the GS/OS interface to device drivers
- a set of low-level system service calls available to device drivers
- the collection of drivers that are provided with GS/OS

Part I of this reference describes the application interface to GS/OS for direct device access: It documents all device calls and describes the individual GS/OS device drivers that applications can call.

Part II of this reference describes the GS/OS interface to drivers: It shows how to design and write a device driver, documents all calls a driver must accept, and describes how a driver can get information and services it needs from GS/OS.

Appendixes to this reference describe how GS/OS generated drivers interact with slot-based firmware I/O drivers and what errors GS/OS can return.

What is the device level?

As described in the introduction to *GS/OS Reference,* GS/OS consists of three interface levels: the application level, the file system level, and the device level. Figure I-1 is a generalized diagram of GS/OS, showing how the device level relates to the rest of the system.

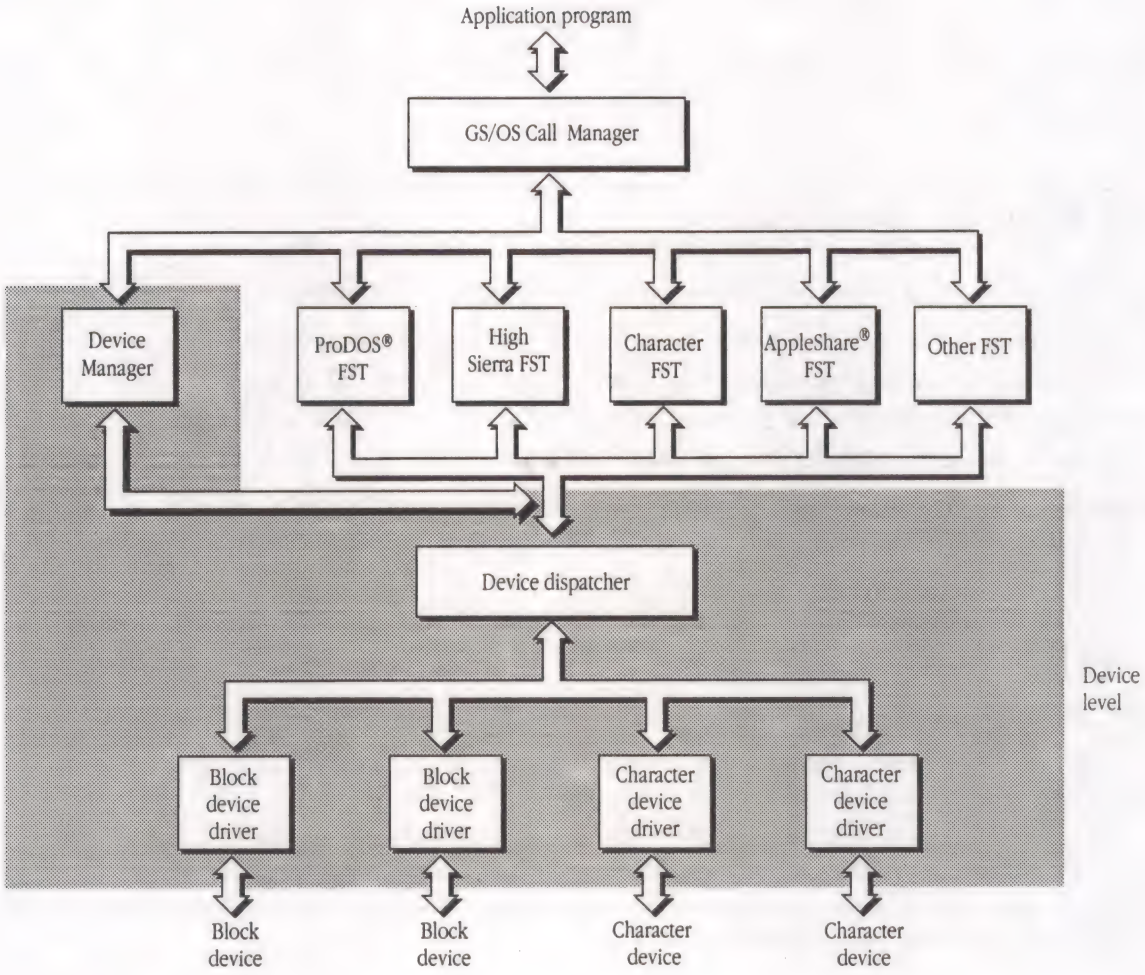
In general, the device level sits between the file system level and hardware devices, translating the device I/O calls made by a file system translator (FST) into the calls that access data on peripheral devices. Note also that part of the device level (the Device Manager) extends upward into the level occupied by file system translators. By making calls through the Device Manager, applications can access devices at a high level, in a manner analogous to the way they access files.

Different components of the device level handle different device-access needs:

- File system translators, which convert file I/O calls into equivalent driver calls, go through the device dispatcher. Driver calls are described in Chapter 10.
- Applications that wish to access devices directly make device calls that go through the Device Manager. Device calls are described in Chapter 1. Like file I/O calls, device calls are translated into driver calls by the Device Manager.
- The device dispatcher itself makes other driver calls when setting up drivers or shutting them down. How the device dispatcher interacts with drivers is described in Chapter 8.
- GS/OS device drivers are the lowest level of GS/OS; they access device hardware directly. The individual drivers that accompany GS/OS are described in Chapters 2–7.
- The device level is extensible; you can write your own device driver for GS/OS. Device driver structure and design are described in Chapter 8; how drivers handle caching and configuration is discussed in Chapters 9 and 10.
- Device drivers that need access to system features and functions can make system service calls to GS/OS. System service calls are described in Chapter 11.

What GS/OS device drivers are and how the Device Manager, the device dispatcher, and the rest of GS/OS interact with them are the subjects of the rest of this chapter.

■ **Figure I-1** Device level in GS/OS



GS/OS drivers

A GS/OS driver is a program, executing from RAM, that directly or indirectly handles all input/output operations to or from a hardware device and also provides information to the system about the device. GS/OS drivers must be able to accept and act upon a specific set of calls from GS/OS.

Generally, each hardware device (or group of closely related devices) needs its own driver. Disk drives, printers, serial ports, and the console (keyboard and screen) can all be accessed through their drivers.

This section discusses the different driver classifications that GS/OS recognizes.

Block drivers and character drivers

There are two fundamental types of drivers, in terms of the kinds of devices they control:

- **Block drivers** allow access to **block devices,** such as disk drives, from which a certain number (one block) of bytes is read from or written to the device at a time, and on which any block within a file can be accessed at any time. Block devices are also called *random-access* devices because all blocks are equally accessible.
- **Character drivers** allow access to **character devices**, such as printers or the console, in which a single character (byte)—or a stream of consecutive characters—is read or written at a time, and access is available to only the current byte being read or written. Character devices are also called *sequential-access* devices because each byte must be taken in sequence.

GS/OS fully supports both types of drivers and includes drivers of each type. For example, the Console driver (see Chapter 8 of *GS/OS Reference*) is a character driver, and the AppleDisk 3.5 driver (see Chapter 3) is a block driver.

Loaded drivers and generated drivers

GS/OS also distinguishes between drivers on the basis of origin, in order to take advantage of the many existing device drivers (both built in and on peripheral cards) for the Apple® II family of computers:

- **Loaded drivers** are drivers that are written to work directly with GS/OS and that are usually loaded in from the system disk at boot time.
- **Generated drivers** are drivers that are *constructed* by GS/OS itself to provide a GS/OS interface to existing, slot-based firmware drivers in ports or on peripheral cards.

At boot time, GS/OS first loads and initializes all loaded drivers. Then, for slots that contain devices that do not have loaded drivers, GS/OS generates the appropriate character or block drivers. Generated drivers are discussed further in Chapter 7.

Because all generated drivers are created by GS/OS, any driver that you write for GS/OS will of course be a loaded driver. How to write a loaded driver is discussed in Part II of this reference.

Device drivers and supervisory drivers

It is simplest to assume that each hardware device is associated with only one driver and that each driver is associated with only one hardware device. It is only slightly more complex to have more than one device controlled by a single driver; a single block driver can access several disk drives, for example. In either case the driver accesses its hardware devices directly.

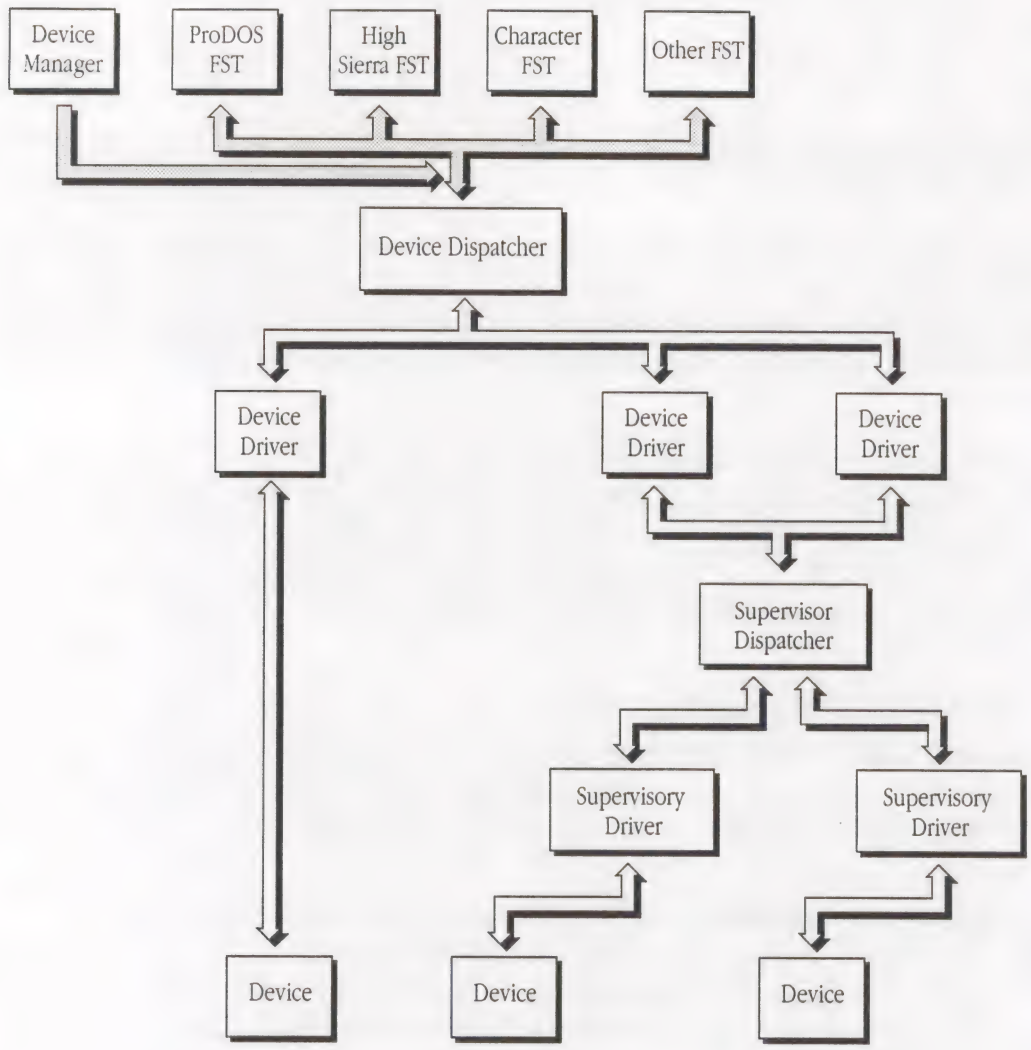
More complexity is possible, however. In some cases there are logical "devices" (hardware controllers such as a SCSI port) that must handle I/O requests from more than one driver (for example, a SCSI hard disk driver and a SCSI CD-ROM driver) and access more than one type of device. To handle those situations, GS/OS allows for special drivers that arbitrate calls from individual device drivers and dispatch them to the proper individual devices.

Therefore, GS/OS also defines these two types of driver:

- A **device driver** is a driver that accepts the standard set of **driver calls** (device I/O calls made by an FST or by an application through the Device Manager). A device driver can conduct I/O transactions directly with its device or indirectly through a supervisory driver.
- A **supervisory driver** (or **supervisor**) arbitrates use of a hardware controller by several device drivers in cases where a single hardware controller conducts I/O transactions with several devices. A supervisory driver does not accept I/O calls directly from FSTs or the Device Manager; it accepts only **supervisory-driver calls** from its individual device drivers.

The presence of supervisory drivers adds more layers to the GS/OS device level. Because more than one supervisory driver can be active at a time, there is a **supervisor dispatcher** to route the requests of device drivers to the proper supervisory driver. The supervisor dispatcher relates to supervisory drivers much as the device dispatcher relates to device drivers. This device-level driver hierarchy is diagrammed in Figure I-2.

■ **Figure I-2** Driver hierarchy within device level



Supervisory drivers and their accompanying device drivers are always loaded drivers, but they can be character drivers, block drivers, or both; that is, a single driver does not have characteristics that restrict it to being solely a block or character device.

Supervisory drivers are closely tied to their device drivers. During the boot sequence all supervisory drivers are loaded and started before any device drivers. This procedure ensures that when a loaded device driver is started, its supervisory driver will be available to it. Other than that, GS/OS is not concerned with the rules of arbitration between a supervisory driver and its loaded device drivers.

Besides simplifying the device interface for applications and providing increased hardware independence, the use of supervisory drivers allows individual device drivers to be added to the system without requiring the replacement or revision of existing drivers.

The differences between device drivers and supervisory drivers are explained more fully in Chapter 8. The rest of the discussion in this chapter concerns device drivers only.

How applications access devices

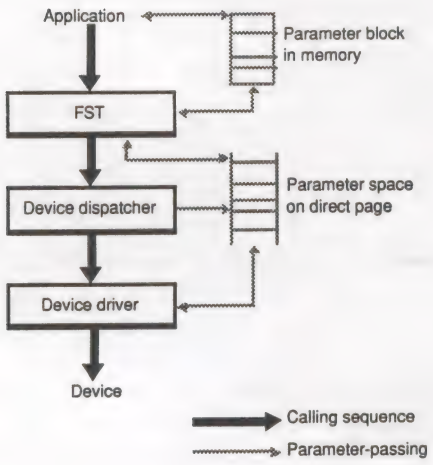
When an application makes a call that results in any kind of I/O, device access occurs. That device access is either indirect, through an FST, or direct, through the Device Manager.

Through an FST

Device access through a file system translator is completely automatic and transparent to the application. When an application performs file I/O by making a standard GS/OS call (as described in Chapter 3 of *GS/OS Reference*) such as Create, Read, or Write, the GS/OS Call Manager passes the call along to the appropriate FST, which converts it to a driver call and sends it to the device dispatcher, which routes it to the appropriate device driver. The device driver in turn accesses the device and performs the requested task.

In most cases the application does not know what device is being accessed. It might not even know which file system is being used. Figure I-3 shows the schematic progress of a typical GS/OS call from application to device, including how parameters are passed.

■ **Figure I-3** Diagram of GS/OS call



High-level calls pass parameters differently than low-level calls do. When an FST receives a call from an application, it converts the parameter block information into data on the GS/OS **direct page**; that conversion makes the data available to low-level software, including drivers. The call then passes through the device dispatcher and to the driver. After the call has been completed, the driver puts any return information into the direct-page parameter space; the FST transfers that information back to the application's parameter block and returns control to the application.

Through the Device Manager

A typical Apple IIGS® application does not need to make any calls to access devices directly. File calls made by the application pass through an FST and are automatically converted into the correct driver calls, which read or write the desired data. The application need not be concerned with the specific device, or even the specific file system, used to store the data.

However, there are times when a particular process is specific to a particular type of device. If your application needs to do something that specific, such as taking user input from the console in text mode, you will need to know how to make a specific driver perform a specific action. That's where device calls come in.

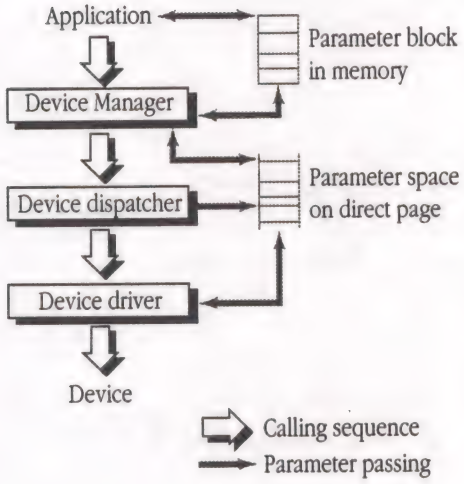
Device calls are application-level GS/OS calls, just like all the calls discussed in Chapter 10 of *GS/OS Reference*. Your application sets up a parameter block in memory and makes the call as described in Chapter 3 of *GS/OS Reference*. The only difference from a normal file-access call is that the device calls are routed through the Device Manager rather than through an FST. See Figure I-4.

The Device Manager converts the call into a driver call and sends it to the device dispatcher, which passes it on to the device driver; the driver then acts on it accordingly.

The Device Manager is similar to an FST but is limited in its support of GS/OS system calls and is independent of any file system. It supports only those GS/OS calls that provide an application with direct access to a peripheral device or device driver, while providing an FST-like interface between the application and the device dispatcher.

The Device Manager handles only six GS/OS calls: DInfo, DStatus, DControl, DRead, DWrite, and DRename. Extensions to DStatus and DControl allow device-specific functions to be called. All other application-level GS/OS calls that access devices must pass through an FST. Device calls are documented in detail in Chapter 1 of this reference.

■ **Figure I-4** Diagram of device call



When the Device Manager receives a device call from an application, it converts the parameter block information into data on the GS/OS direct page; that conversion makes the data available to low-level software, including drivers. The call then passes through the device dispatcher and to the driver. After the call has been completed, the driver puts any return information into the direct-page parameter space; the Device Manager transfers that information back to the application's parameter block and returns control to the application.

How GS/OS communicates with drivers

Device drivers communicate with the operating system in two basic ways: by receiving driver calls from the device dispatcher and by making system service calls to GS/OS.

The device dispatcher

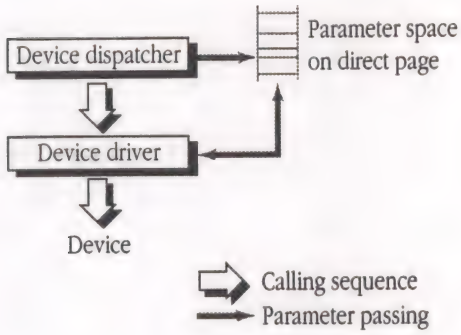
All calls to device drivers pass through the device dispatcher. The device dispatcher maintains a list of information about each driver attached to the system and thus knows where to transfer control to when it receives a driver call from an FST or the Device Manager.

The driver calls that the device dispatcher receives from FSTs or the Device Manager and passes on to drivers are Driver_Read, Driver_Write, Driver_Status, and Driver_Control. They are documented in Chapter 10. These particular driver calls have names that are very similar to the names of their equivalent device calls. The lower parts of Figures I-3 and I-4 diagram the call progress and parameter passing for these driver calls.

Note also that there is no equivalent driver call for the device calls DInfo and DRename; these calls are handled entirely by the device dispatcher by consulting its list of device information. DInfo must subsequently make a Driver_Status call to determine the volume size if a block device's size is dynamic.

The device dispatcher and other parts of GS/OS also make driver calls that are not translations of device calls and are concerned with setting up drivers to perform I/O and with shutting them down afterward. These other driver calls are Driver_Startup, Driver_Open, Driver_Close, Driver_Flush, and Driver_Shutdown and are documented in Chapter 10. Figure I-5 shows the progress of such a driver call; note that Figure I-5 is also identical to the lower part of Figures I-3 and I-4.

■ **Figure I-5** Diagram of driver call



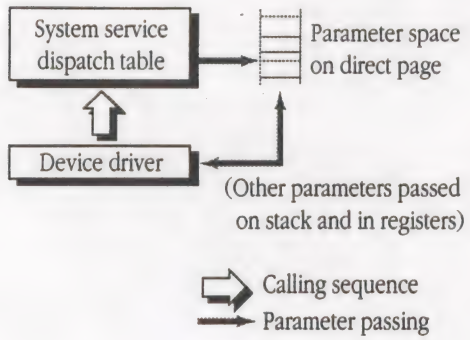
System service calls

GS/OS provides a standardized mechanism for passing information and providing services among its low-level components such as FSTs and device drivers. That mechanism is the **system service call.**

System service calls exist for various purposes: to perform disk caching, to manipulate buffers in memory, to set system parameters such as execution speed, to send a **signal** to GS/OS, to call a supervisory driver, or to perform other tasks. Not all drivers need all of these services, but each is useful in a particular situation. If you are writing a device driver, consult Chapter 11 to see what system service calls are available to your driver and what each does.

Drivers make system service calls through jumps to locations specified in the system service dispatch table. Parameters are passed back and forth through registers on the stack and through the same direct-page space used for driver calls. See Figure I-6.

■ **Figure I-6** Diagram of system service call



Driver features

This section describes some of the notable features that GS/OS drivers can have. See the referenced chapters for more information.

Configuration

GS/OS drivers can be configurable, meaning that the user can customize and store certain driver settings. For example, for a driver that controlled a serial port, such parameters as bits per second, parity, stop bits, and so on could be customized and stored.

Many users will never need to configure drivers. Others will use the capability when adding a peripheral device or adjusting device driver or system default settings. As a device driver writer, you can choose which user-configurable features you want in your driver.

The specific formats in which configuration options are to be presented to the user, how the chosen settings are to be stored, and how the options are to be set up by the driver the first place are specific to the individual driver. However, the overall format in which the configuration parameters are to be stored in the device driver and what calls are used to set or modify those parameters are defined in Chapters 8 and 10.

Cache support

Caching is the process by which frequently accessed disk blocks are kept in memory to speed subsequent accesses to those blocks. On the Apple IIGS computer the user can control what the maximum cache size can be. It is the driver, however, that is responsible for making caching work. GS/OS block drivers should support caching.

The GS/OS cache is a **write-through** cache. That is, when an FST issues a Write call to a device driver, the driver writes the same data to the block in the cache and the equivalent block on the disk. Never does the block in the cache contain information more recent than that in the disk block. Also, like most caching implementations, the GS/OS cache uses a least recently used **(LRU)** algorithm: Once the cache is full, the least recently used (accessed) block in the cache is sacrificed for the next new block that is written.

Cache memory is obtained and released by GS/OS on an as-needed basis. Only as individual blocks are cached is the necessary amount of memory (up to the maximum set by the user) assigned to the cache. The size of a block in the cache is essentially unrestricted, limited only by the maximum size of the cache itself.

Drivers implement caching by making system service calls. Caching is described in Chapter 9; system service calls are documented in Chapter 11.

Terms and conventions

Terms introduced in this book are printed in **bold** type where defined and are listed in the glossary.

Assembly-language labels, entry points, programs and subroutine names, and filenames that appear in text passages are printed in Apple Courier typeface (for example, `DoWItem` and `MENU.PAS`). There is one exception: The names of Apple IIGS system software routines such as toolbox calls and operating system calls (for example, NewModalDialog and QUIT) are printed in normal type.

The following words mark special messages to you:

- *Note:* Text set off in this manner presents sidelights or interesting information.

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

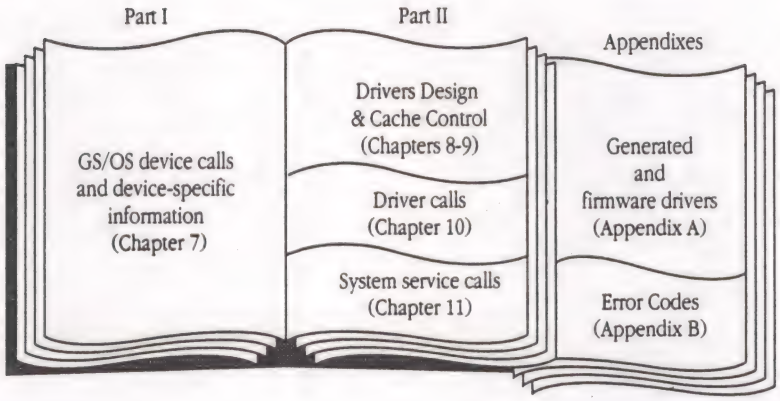
▲ **Warning** Text set off in this manner—with the word **Warning**—indicates potential serious problems. ▲

Code-list convention

The source code listings in these chapters and the driver examples in the appendixes are in assembly language. In addition to the 65C816 syntax and notation, please note the following conventions:

- Toolbox calls are in **boldface**.
- Reserved words are in *italics*.
- Names of functions, procedures, types, and user-defined constants begin with lowercase letters.
- Boolean values (such as TRUE and FALSE) are all CAPITAL letters.

Part I **Using GS/OS Device Drivers**



Chapter 1 GS/OS Device Call Reference

This chapter explains how to call device drivers and documents the GS/OS® **device calls:** application-level calls that give applications direct access to devices by bypassing all file systems.

This chapter repeats the device-call descriptions of *GS/OS Reference* but provides more complete documentation; in particular, it describes all the standard DStatus and DControl subcalls.

This chapter describes only standard GS/OS (class-1) device calls; for descriptions of how GS/OS handles equivalent ProDOS® 16 (class-0) device calls, see Appendix B of *GS/OS Reference.*

How to make a device call

Your application makes GS/OS device calls just like it makes any other application-level GS/OS calls—it sets up a parameter block in memory and executes either an in-line or a stack-based call method (either directly or with a macro). Chapter 3 of *GS/OS Reference* describes all the methods for making GS/OS calls.

All device calls are handled by the Device Manager and are listed in Table 1-1. The rest of this chapter documents how the device calls work.

■ **Table 1-1** GS/OS device calls

Call number	Name
$202C	DInfo
$202D	DStatus
$202E	DControl
$202F	DRead
$2030	DWrite
$2036	DRename

The diagram accompanying each call description in this chapter is a simplified representation of the call's parameter block in memory. The width of the parameter block diagram represents 1 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 the following description of **minimum parameter count**.
- **Size and type:** Each parameter is also identified by size (word or longword) and type (input or result, and value or pointer). A word is 2 bytes; a longword is 4 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 diagram.

Each parameter is described in detail after the diagram. Additional important notes, call requirements, and principal error results follow the parameter descriptions.

DInfo ($202C)

Description

DInfo returns certain attributes of a device known to the system. The information is in the device's device information block (DIB). The Device Manager makes a call to the device dispatcher to obtain the pointer to the DIB and then returns the requested parameters from the DIB. If the `pCount` parameter is greater than 3, the DInfo call actually issues a DStatus call with a status code of 0 to the device to obtain the current block count. This ensures that any dynamic parameters in the DIB are updated.

Parameters

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

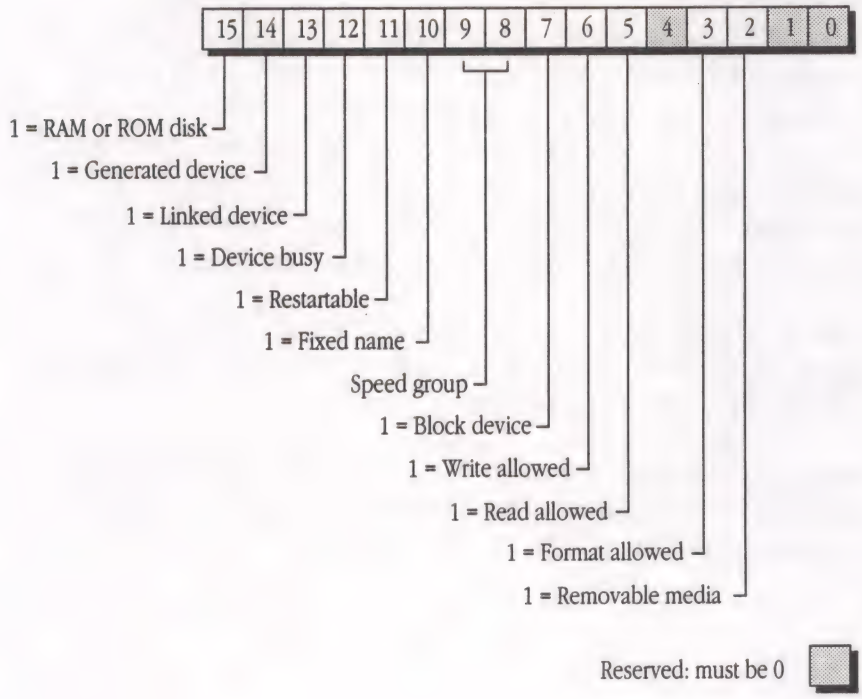
`pCount` Word input value: the number of parameters in this parameter block. Minimum is 2; maximum is 11.

`devNum` Word input value: a nonzero device number. GS/OS assigns device numbers in sequence 1, 2, 3,... as it loads or creates the device drivers. Because the device list is dynamic, there is no fixed correspondence between devices and device numbers. To get information about every device in the system, make repeated calls to DInfo with `devNum` values of 1, 2, 3,... 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 corresponding to the device number. The maximum size of the device-name string is 32 bytes, so the maximum size of the returned value is 34 bytes. The buffer size should thus be 36 bytes.

`characteristics`
Word result value: Individual bits in this word give the general characteristics of the device. This is its format:

■ **Figure 1-1** Device characteristics word



In the device characteristics word, *linked device* means that the device is one of several partitions on a single, removable medium. *Device busy* is maintained by the device dispatcher to prevent reentrant calls to a device.

Speed group defines the speed at which the device requires the processor to be running. Speed group has these binary values and meanings:

Setting	Speed
$0000	Apple IIGS normal speed
$0001	Apple IIGS fast speed
$0002	Accelerated speed
$0003	Not speed dependent

Restartable defines whether or not the device driver is to be purged or is to remain in memory when switching between a ProDOS 8 application and a GS/OS application. If this bit is a 1, the driver is restartable and will not be purged when quitting from a GS/OS application program to a ProDOS 8 application program. If this bit is a 0, the driver is not restartable and will be purged.

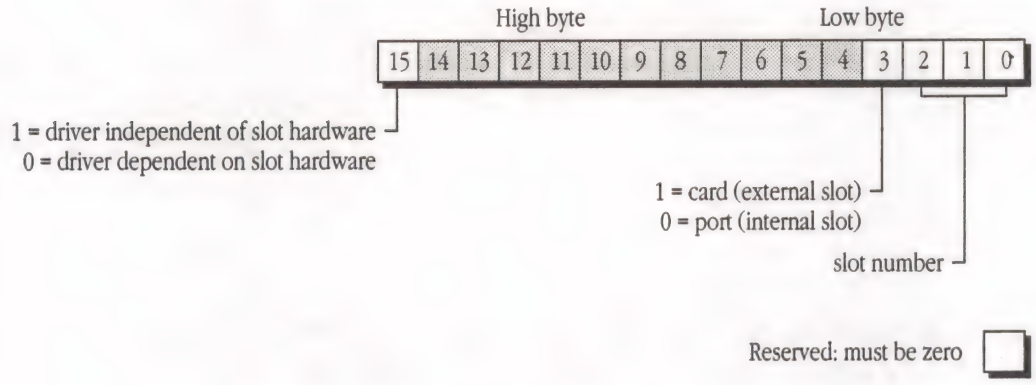
Device drivers are always loaded from disk and thus may contain preinitialized data. This data may be modified during the normal execution of the device driver. In order to make these device drivers restartable, the device driver Shutdown call must be modified to reset the variables that have been modified during device driver execution, so that subsequent Startup calls to the driver will function properly. This is an additional task for the device driver Shutdown call and does not in any way diminish previous requirements on the driver Shutdown call.

Fixed name defines whether or not the device driver name can be changed in the memory-resident DIB. If this bit is a 1, the DRename call will not alter the device driver name.

`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 contains 0.

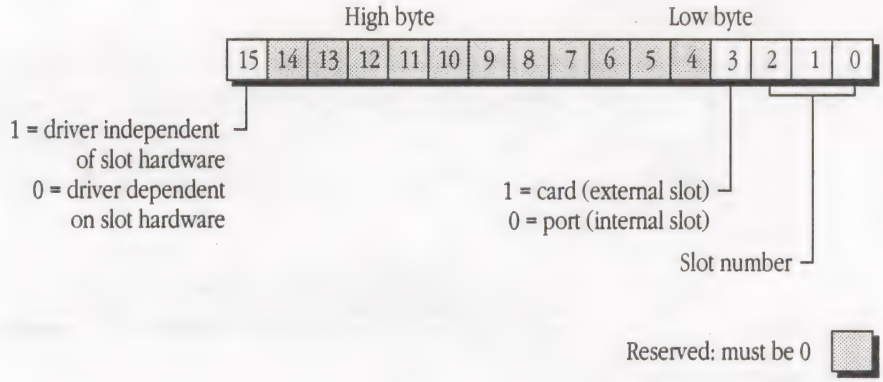
`slotNum` Word result value: slot number of (1) the device hardware or (2) the resident firmware (port) associated with the device. Bits 0 through 2 define the slot (valid values are $1 through $7), and bit 3 indicates whether it is an internal port (controlled by firmware within the Apple® IIGS® computer) or an external slot containing a card with its own firmware.

For a given slot number, either the external slot or its equivalent internal port is active (switched in) at any one time; bit 15 indicates whether or not the device driver must access the peripheral card's I/O addresses. For more information on those addresses, see the *Apple IIGS Hardware Reference*.



`unitNum` Word result value: unit number of the device within the given slot. Because different drivers permit different numbers of devices per slot, the value of this parameter is driver specific; it has no direct correlation with the GS/OS device number or any other device designation used by the system.

`version` Word result value: version number of the device driver. This parameter has the same format as the SmartPort version parameter. These are its fields:



- *Note:* This parameter has a format different from that of the version parameter returned from the GS/OS GetVersion call.

`deviceIDNum` Word result value: an identifying number associated with a particular type of device. Device ID may be useful for Finder™-like applications when determining what type of icon to display for a certain device. These are the currently defined device IDs:

ID	Description	ID	Description
$0000	Apple 5.25 drive (includes UniDisk™, DuoDisk®, Disk II®, and Disk IIc drives)	$000F	ROM disk
$0001	ProFile™ (5 MB)	$0010	File server
$0002	ProFile (10 MB)	$0011	(Reserved)
$0003	Apple 3.5 drive (includes UniDisk 3.5 drive)	$0012	Apple Desktop Bus™
$0004	SCSI device (generic)	$0013	Hard disk drive (generic)
$0005	SCSI hard disk drive	$0014	Floppy disk drive (generic)
$0006	SCSI tape drive	$0015	Tape drive (generic)
$0007	SCSI CD-ROM drive	$0016	Character device (generic)
$0008	SCSI printer	$0017	MFM-encoded disk drive
$0009	Modem	$0018	AppleTalk network (generic)
$000A	Console	$0019	Sequential-access device
$000B	Printer	$001A	SCSI scanner
$000C	Serial LaserWriter®	$001B	Other scanner
$000D	AppleTalk® LaserWriter	$001C	LaserWriter SC
$000E	RAM disk	$001D	AppleTalk main driver
		$001E	AppleTalk file server
		$001F	AppleTalk RPM driver

`headLink` Word result value: This parameter holds 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 represent separate partitions on a single disk. A value of 0 indicates that no link exists.

`forwardLink` Word result value: This parameter holds 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 represent separate 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 (extended DIB), if provided.

Errors

$11 Invalid device number
$53 Parameter out of range

DStatus ($202D)

Description

DStatus returns status information about a specified device. DStatus is really four or more calls in one. Depending on the value of the status code parameter (`statusCode`), DStatus can return several classes of status information.

Parameters

Offset		No.	Size and type
$00	pCount	—	Word input value (minimum = 5)
$02	devNum	1	Word input value
$04	statusCode	2	Word input value
$06	statusList	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.

`statusCode` Word input value: a number indicating the type of status request being made. Each status code corresponds to a particular DStatus subcall (DStatus subcalls are described later in this section).

`statusList` Longword input pointer: points to a buffer in which the device returns its status information. The format of the data in the status buffer depends on the status code. See individual DStatus subcall descriptions for more information.

`requestCount` Longword input value: specifies the number of bytes to be returned in the status list. The call can 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 is always less than or equal to the request count.

Buffer size

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

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

DStatus subcalls

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

Table 1-2 lists the currently defined values for `statusCode` and the subcalls invoked. Following the DStatus error listings, each of the status subcalls is described individually.

■ **Table 1-2** DStatus subcalls

Status code	Subcall name
$0000	GetDeviceStatus
$0001	GetConfigParameters
$0002	GetWaitStatus
$0003	GetFormatOptions
$0004	GetPartitionMap
$0005–$7FFF	(Reserved)
$8000–$FFFF	(Device-specific subcalls)

Errors

$11 Invalid device number
$53 Parameter out of range

GetDeviceStatus

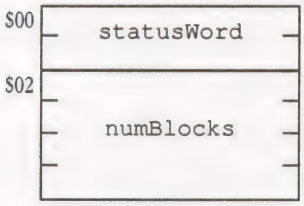
`statusCode` = $0000

Description

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

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

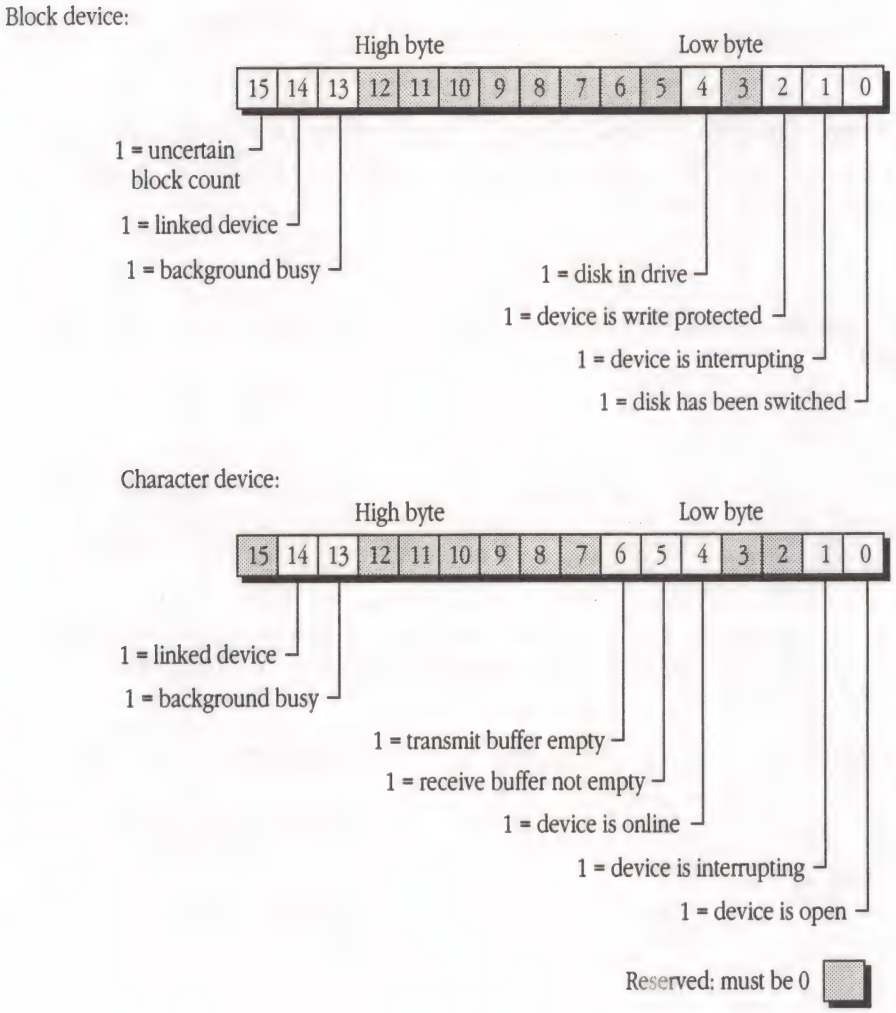
Parameters



Offset	No.	Size and type
$00	Word	Status word (see following definition)
$02	Longword	Number of blocks on device

The device status word has two slightly different formats, depending on whether the device is a block device or a character device. Figure 1-2 shows the bits of the device status word for both types of devices.

■ **Figure 1-2** Device status word



The *linked device* bit indicates that this device is one of several in a linked list of DIBs that has been constructed to keep track of a changing number of partitions (such as for a CD-ROM drive) and must dynamically adjust the linked list as new CDs are inserted.

The *background busy* bit is set if the device is currently executing a background task.

The *receive buffer not empty* bit is set whenever the receive buffer contains data.

The *volume on line* bit is set whenever the device is mounted.

The *device is interrupting* bit is set whenever the device is requesting interrupt service.

The *device is open* bit (character devices only) is set whenever a Driver_Open call is made to the character device driver.

To maintain future compatibility, the driver must return 0 in all reserved bit positions for the status word, because reserved bits may be assigned new values in the future.

GetConfigParameters

`statusCode` = $0001

Description

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

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

Parameters

Offset		No.	Size and type
$00	`length`	Word	Length of list (in bytes)
$02	`configParamList`	—	Configuration list

GetWaitStatus

`statusCode` = $0002

Description

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

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

Parameters

Offset		No.	Size and type
$00	`waitMode`	Word	Wait/no-wait status of device

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

GetFormatOptions

`statusCode` = $0003

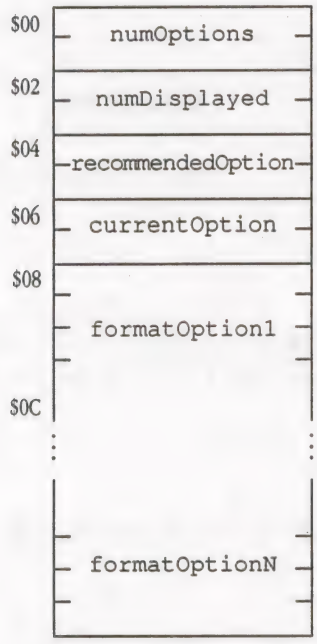
Description

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

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

The GetFormatOptions subcall returns the list of formatting options for a particular device. Devices that do not support media variables return a transfer count of 0 and generate no error. Character devices do nothing and return no error from this call. If a device does support media variables, it returns a status list consisting of a four-word header followed by a set of entries, each of which describes a formatting option.

Parameters



Offset	No.	Size and type
$00 numOptions	Word	Number of format option entries in list
$02 numDisplayed	Word	Number of options to be displayed
$04 recommendedOption	Word	Recommended default formatting option
$06 currentOption	Word	Option with which currently on-line media was formatted ($0000 = unknown)
$08 formatOption1	(16 bytes)	First format option entry
formatOptionN	(16 bytes)	Last format option entry

Of the total number of options in the list, zero or more can be displayed on the initialization dialog presented to the user when initializing a disk (see the calls Format and EraseDisk in Chapter 10 of *GS/OS Reference*). The options to be displayed are always the first ones in the list. (Undisplayed options are available so that drivers can provide FSTs with logically different options that are actually physically identical and therefore needn't be duplicated in the dialog box.)

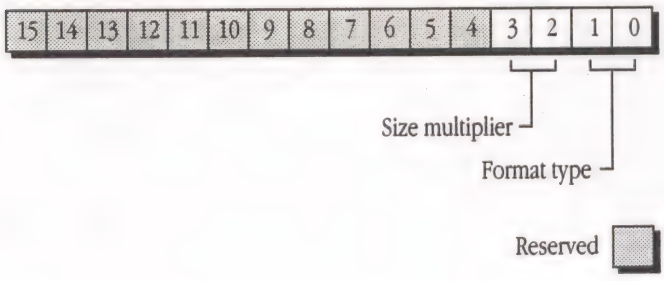
Each format option entry consists of 16 bytes, containing these fields:

Offset		Size	Description
$00	formatOptionNum	Word	Number of this option
$02	linkRefNum	Word	Number of linked option
$04	flags	Word	(See following definition)
$06	blockCount	Longword	Number of blocks supported by device
$0A	blockSize	Word	Block size in bytes
$0C	interleaveFactor	Word	Interleave factor (in ratio to 1)
$0E	mediaSize	Word	Media size (see flags description)

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

Bits within the flags word are defined as follows:

■ **Figure 1-3** Flags word



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

00 Universal format

01 Apple format

10 Non-Apple format

11 (Not valid)

Size multiplier is used, in conjunction with the parameter `mediaSize`, to calculate the total number of bytes of storage available on the device. Size multiplier can have these binary values and meanings:

00 `mediaSize` is in bytes

01 `mediaSize` is in kilobytes (KB)

10 `mediaSize` is in megabytes (MB)

11 `mediaSize` is in gigabytes (GB)

Example

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

The entire format options list looks like this:

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

GetPartitionMap

`statusCode` = $0004

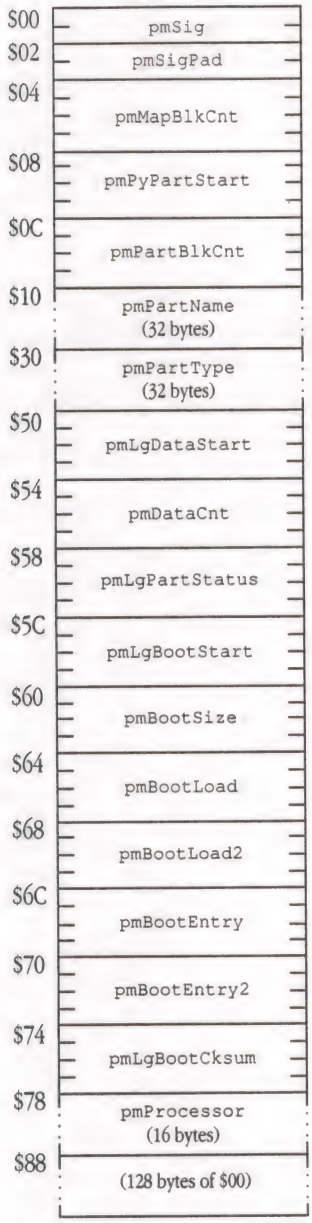
Description

This subcall returns the partition map for a partitioned disk or other medium in the status list. The structure of the partition information is device dependent. The map for Apple's implementation is described here. If you would like your partitioned device to be supported by the Apple Advanced Disk Utility (ADU), you must use a similar map.

There are times that you will want to alter the contents of the partition map of a partitioned device or verify its contents. This subcall returns the partition map of a head device (the first in a linked list of partitions) that you specify. If you specify any device other than the head device, error $11 (invalid device number) is returned.

The size of the partition map can vary, and the only way for an application to be sure that the entire partition map has been read is to validate the `pmMapBlkCnt` field in the map with the transfer count returned by the driver. If the value of `pmMapBlkCnt` times the value of `blockSize` is not equal to `transferCount`, then the application program must reissue the call with the appropriate `requestCount` value. If `requestCount` is not an integral multiple of `blockSize`, then an invalid byte count error is returned. The partition map is returned beginning with the data first entry. It is not possible to read a specific entry in the partition map; if you wish to reassign an entry to a different type operating system, use the AssignPartitionOwner DControl subcall. The partition map contains these fields:

■ **Figure 1-4** Partition map



The above figure shows a typical partition map. Note that all fields are stored in high-byte-first order; even though the partition signature is defined as $504D (ASCII "PM,"for "partition map"), it appears to the 6502 family of processors (including the 65C816 microprocessor) as $4D50.

`pmSig` Partition signature; always $504D to signify that this block contains a partition map.

`pmSigPad` Reserved; must be $00.

`pmMapBlkCnt` Number of blocks in the partition map.

`pmPyPartStart` First physical block of the partition.

`pmPartBlkCnt` Number of blocks in the partition map.

`pmPartName` A user-defined partition name. Can be any GS/OS legal name (which may or may not be the same name as used by the system). If the name is less than the maximum character length, it must be terminated with a null character (ASCII code $00). An empty name can be specified by setting the first byte to the null character.

`pmPartType` Partition type. This is an ASCII string of from 1 to 32 bytes in length; case is not significant. If the type is less than the maximum character length, it must be terminated with a null character (ASCII code $00). An empty name can be specified by setting the first byte to the null character.

Here are some example partition types:

- Apple_MFS
- Apple_HFS
- Apple_Unix_SVR2
- Apple_partition_map
- Apple_Driver
- Apple_PRODOS
- Apple_Free
- Apple_Scratch

`pmLgDataStart` First logical block of the data area.

`pmDataCnt` Number of blocks in the data area.

`pmPartStatus` Contains partition status information:

Bit	Meaning
0	Set if a valid partition map entry
1	Set if partition is already allocated; clear if available
2	Set if partition is in use; might be cleared after a system reset
3	Set if partition contains valid boot information
4	Set if partition allows reading
5	Set if partition allows writing
6	Set if boot code is position independent
7	Free for your use

`pmLgBootStart` First logical block of the boot code.

`pmBootSize` Size in bytes of the boot code.

`pmBootLoad` Specifies the memory address at which the boot code is to be loaded.

`pmBootLoad2` Additional boot load information.

`pmBootEntry` Specifies the memory address at which execution will begin after the boot code is loaded.

`pmBootEntry2` Additional boot entry information.

`pmBootCksum` Boot code checksum.

`pmProcessor` Processor type. This field contains an ASCII string of 1 to 16 bytes in length, upper- or lowercase characters. If either name is less than the maximum character length, it must be terminated with a null character (ASCII code $00). An empty name can be specified by setting the first byte to the null character.

Here are some examples of processor types:

- 68000
- 68030
- 6502
- 65C816
- 8080
- 80386

Device-specific DStatus subcalls

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

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

DControl ($202E)

Description

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

Parameters

Offset		No.	Size and type
$00	`pCount`	—	Word input value (minimum = 5)
$02	`devNum`	1	Word input value
$04	`controlCode`	2	Word input value
$06	`controlList`	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.

`controlCode` Word input value: specifies the type of control request being made. Each control request corresponds to a DControl subcall (DControl calls are described later in this section).

`controlList` Longword input pointer: points to a buffer that contains the control information for the device. The format of the data and the required minimum size of the buffer are different for different subcalls. See the individual subcall descriptions.

`requestCount` Longword input value: indicates the number of bytes to be transferred. For control subcalls that use a control list, this parameter gives the size of the control list. For control subcalls that do not use the control list, this parameter is not used.

`transferCount` Longword result value: for control subcalls that use a control list, this parameter indicates the number of bytes of information taken from the control list by the device driver. For control subcalls that do not use the control list, this parameter is not used.

Control-list buffer

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

For those subcalls that pass no information in the control list, the driver does not access the control list and verify that its length word is 0; instead, the driver ignores the control list entirely.

Subcalls

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

Table 1-3 lists the currently defined values for `controlCode`. Following the DControl error listings, each of the standard control subcalls is described individually.

■ **Table 1-3** DControl subcalls

controlCode	Subcall name
$0000	ResetDevice
$0001	FormatDevice
$0002	EjectMedium
$0003	SetConfigParameters
$0004	SetWaitStatus
$0005	SetFormatOptions
$0006	AssignPartitionOwner
$0007	ArmSignal
$0008	DisarmSignal
$0009	SetPartitionMap
$000A–$7FFF	(Reserved)
$8000–$FFFF	(Device-specific subcalls)

Errors

$11 Invalid device number
$21 Invalid control code
$53 Parameter out of range

ResetDevice

`controlCode` = $0000

Description

The ResetDevice subcall sets a device's configuration parameters back to their default values. Many GS/OS device drivers contain default configuration settings for each device they control; see Chapter 8, "GS/OS Device Driver Design," for more information.

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

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

FormatDevice

`controlCode` = $0001

Description

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

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

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

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

EjectMedium

`controlCode` = $0002

Description

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

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

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

SetConfigParameters

`controlCode` = $0003

Description

The SetConfigParameters subcall is used to send device-specific configuration parameters to a device. The configuration parameters are contained in the control list. The first word in the control list (`length`) indicates the length of the configuration list, in bytes. The configuration parameters follow the length word.

Parameters

Offset		No.	Size and type
$00	`length`	Word	Length of list (in bytes)
$02	`configParamList`	—	Configuration list

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

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

SetWaitStatus

`controlCode` = $0004

Description

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

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

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

Parameters

Offset		No.	Size and type
$00	waitMode	Word	Wait/no-wait status of device

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

SetFormatOptions

`controlCode` = $0005

Description

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

The SetFormatOptions subcall is used to set these media-specific formatting parameters prior to executing a FormatDevice subcall. SetFormatOptions does not itself cause or require a formatting operation. The control list for SetFormatOptions consists of two word-length parameters.

Parameters

Offset		No.	Size and type
$00	formatOptionNum	Word	Number of format option
$02	interleaveFactor	Word	Override interleave factor (if nonzero)

The format option number (`formatOptionNum`) specifies a particular **format option entry** from the driver's list of formatting options (returned from the DStatus subcall GetFormatOptions). The format option entry has this format:

Parameters

Offset		No.	Size and type
$00	`formatOptionNum`	Word	The number of this option
$02	`linkRefNum`	Word	Number of linked option
$04	`flags`	Word	File system information
$06	`blockcount`	Longword	Number of blocks supported by device
$0A	`blocksize`	Word	Block size in bytes
$0C	`interleaveFactor`	Word	Interleave factor (in ratio to 1)
$0E	`mediaSize`	Word	Media size

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

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

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

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

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

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

AssignPartitionOwner

`controlCode` = $0006

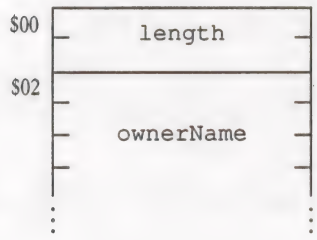
Description

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

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

Partition owner names are assigned by Apple Developer Technical Support and can be up to 32 bytes in length. Uppercase and lowercase characters are considered equivalent. The control list for this call consists of a GS/OS string naming the partition owner.

Parameters



Offset	No.	Size and type
$00	Word	Length of name (in bytes)
$02	—	Partition owner name

Block devices with nonpartitioned media and character devices do nothing with this call and return no error.

ArmSignal

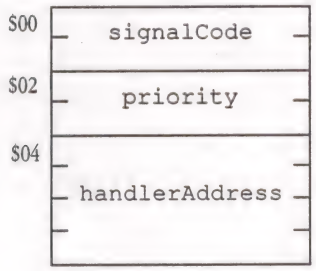
`controlCode` = $0007

Description

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

An ArmSignal call is issued by application programs to inform the driver to call an application-supplied interrupt handler routine at the location specified in `handlerAddress`. The SIGNAL system service call provides the `signalCode` and `priority` values to GS/OS.

Parameters



Offset	No.	Size and type
$00 `signalCode`	Word	ID for this handler and its signals
$02 `priority`	Word	Priority for this handler's signals
$04 `handlerAddress`	Longword	Pointer to signal handler's entry

`signalCode` The `signalCode` parameter contains the ID of the condition that the driver will pass to GS/OS when the condition occurs. The `signalCode` ID is assigned by the caller and must match a unique number defined by the device driver. The only subsequent use of the `signalCode` number is as an input to the DControl subcall DisarmSignal. A device driver should bind only one signal handler to each of its defined `signalCode` numbers.

`priority` The `priority` parameter is the signal priority the caller wishes to assign to the signal condition; $0000 is the lowest priority, and $FFFF is the highest priority.

`handlerAddress` The `handlerAddress` parameter is the entry address of the caller's signal handler. Control is passed to this address when GS/OS dispatches a queued signal associated with an occurrence of the signal condition. See Chapter 9 of the *GS/OS Reference* for a description of the signal handler execution environment.

An example may help to clarify how this subcall works. Imagine that we've defined three different values for the `signalCode` parameter, each representing a unique occurrence within the computer (for example a hardware interrupt).

`signalCode`	Signal source
$0000	Break sequence detected
$0001	Transition detected on the CTS input line
$0002	Transition detected on the DCD input line

An application program needs to be signaled when a break occurs or when the DCD line changes state. If both events happen simultaneously, the application program needs to be informed about the DCD transition first. The application program doesn't care about the CTS transition. Here's how it would call a driver to set up things:

First, the application program calls ArmSignal twice to bind its two signal handlers to the driver's signal sources. The first call uses these parameter values:

`signalCode`	$0002 (ID for detecting DCD transition)
`priority`	$FFFF
`handlerAddress`	Address of handler code

The second ArmSignal call uses these parameter values:

`signalCode`	$0000 (ID for detecting break)
`priority`	$FFFE
`handlerAddress`	Address of handler code

Now that the driver's signal sources have been bound to the signal handlers, several events can occur. This is how GS/OS handles these events:

If DCD changes state:

1. When the signal source detects that DCD has changed state, it issues a SIGNAL system service call with a priority of $FFFF in the A register and the address of the DCD-handling routine in the X and Y registers.
2. When GS/OS is no longer busy and is able to dispatch signals, it calls the application's DCD-handling routine.

If a break occurs:

1. When the signal source notices that a break has occurred, it issues a SIGNAL system service call with a priority of $FFFE in the A register and the address of the break-handling routine in the X and Y registers.
2. When GS/OS is no longer busy and is able to dispatch signals, it calls the application program's break-handling routine.

If DCD changes state and a break occurs:

1. When the signal source notices that a break has occurred, it issues a SIGNAL system service call passing a priority of $FFFE in the A register and the address of code to be called when a break is detected in the X and Y registers. The signal source also notices that DCD has changed state, so it makes another SIGNAL system service call passing a priority of $FFFF in the A register and the address of code to be called on a DCD transition in the X and Y registers.
2. When GS/OS is no longer busy and is able to dispatch signals, it first calls the application program's DCD-handling code, since the DCD signal was given a higher priority by the application when it made the ArmSignal calls. When GS/OS dispatches the next signal, it calls the application program's break-handling code unless another higher-priority signal event has occurred.

DisarmSignal

`controlCode` = $0008

Description

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

Parameters

Offset		No.	Size and type
$00	`signalCode`	Word	Signal handler's ID

SetPartitionMap

`statusCode` = $0009

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

Device-specific DControl subcalls

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

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

DRead ($202F)

Description This call performs a device-level read on a specified device. It transfers data from a character device or block device to a caller-supplied buffer.

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 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 nonblock devices, the parameter must be set to 0.

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

Character devices

You must first open a character device (with an Open call) before reading characters from it with DRead; otherwise, DRead returns error $23 (device not open).

If the parameter `blockSize` is not 0 on a DRead call to a character device, DRead returns error $58 (not a block device).

Block devices

DRead does not support caching. From block devices, DRead always reads data directly from the device, not from the cache (if any). Furthermore, the block being read will not be copied into the cache.

The request count should be an integral multiple of block size; if it is not, the call returns error $2C (invalid byte count). If the block number is outside the range of possible block numbers on the device, the call returns error $2D (invalid block number).

Errors

$11 Invalid device number
$23 Device not open
$2C Invalid byte count
$2D Invalid block number
$53 Parameter out of range
$58 Not a block device

DWrite ($2030)

Description This call performs a device-level write to a specified device. The call transfers data from a caller-supplied buffer to a character device or block device.

Parameters

Offset		No.	Size and type
$00	pCount	—	Word input value (minimum = 6)
$02	devNum	1	Word input value
$04	strPtr	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 nonblock devices, the parameter is unused and must be set to 0.

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

Character devices

You must first open a character device (with an Open call) before writing characters to it with DWrite (or Write); otherwise, DWrite returns error $23 (device not open).

If the parameter `blockSize` is not 0 on a DWrite call to a character device, DWrite returns error $58 (not a block device).

Block devices

DWrite does not support caching. When writing to block devices, DWrite does not also write the blocks into the cache, if there is one.

The request count should be an integral multiple of block size; if it is not, the call returns error $2C (invalid byte count). If the block number is outside the range of possible block numbers on the device, the call returns error $2D (invalid block number).

Errors

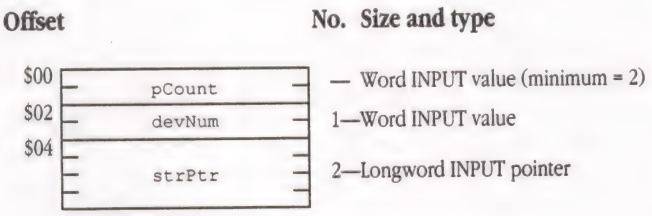
$11 Invalid device number
$23 Device not open
$2C Invalid byte count
$2D Invalid block number
$53 Parameter out of range
$58 Not a block device

DRename ($2036)

Description

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

Parameters



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

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

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

Errors

$11 Invalid device number
$53 Parameter out of range
$67 Device with same name exists

Chapter 2 **The SCSI Driver**

This chapter describes the GS/OS SCSI (Small Computer System Interface) drivers. There are three SCSI block drivers (the Apple HD SC hard disk driver, the AppleCD SC® CD-ROM driver, and the Apple Tape Backup 40SC driver) and one SCSI character driver (the Apple Scanner driver). These drivers support a similar set of calls and provide a common standard driver interface for SCSI devices.

This chapter also describes the SCSI Manager. This supervisory driver manages the SCSI bus and arbitrates between all SCSI device drivers and SCSI peripherals.

Device calls to the SCSI driver

All Apple SCSI drivers support these standard GS/OS device calls:

- DInfo
- DStatus
- DControl
- DRead
- DWrite
- DRename

The SCSI driver also supports additional device-specific DStatus and DControl subcalls. Because the device-specific DStatus and DControl subcalls for the SCSI driver follow the industry standard format for SCSI calls, this chapter provides descriptions of only the Apple-unique commands. All other SCSI commands are identical to the American National Standards Institute (ANSI) SCSI standard and can be found in the ANSI SCSI document. This and other documents should be on your bookshelf if you plan on writing your own driver. They are

- *AppleCD SC Developer's Guide*
- ANSI X3.131-1986, *Small Computer System Interface (SCSI) specification*
- SCSI-2 proposed specification

The Apple publication can be obtained from

Apple Programmer's and Developer's Association (APDA™)
Apple Computer, Inc.
20525 Mariani Avenue, Mail Stop 33-G
Cupertino, CA 95014-6299
800-282-APDA

The other two publications can be obtained from

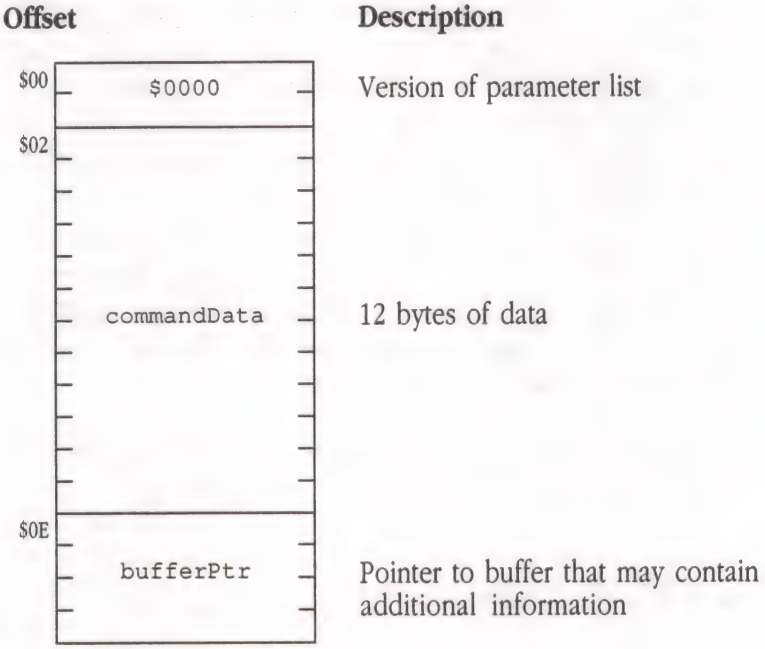
Global Engineering Documents
2805 McGaw Avenue
Irvine, CA 92714
714-261-1455 or 800-854-7179

The SCSI-2 document is necessary because it documents the device support that has been added to the SCSI standard since the release of the original SCSI specification.

DStatus ($202D)

The SCSI driver supports all standard status subcalls. See Chapter 1 of this reference for a description of the DStatus call.

All of the device-specific SCSI driver DStatus subcalls use the same format for the status list (the buffer pointed to by `statusList` in the DStatus call).



- *Note:* Use this status list for all status calls unless you wish to use data-chaining commands. See the data-chaining status list in the section "Data Chaining" later in this chapter.

`Version` Contains version of the parameter list. If you are using data-chaining commands, this version must be 1.

`commandData` Contains subcall-specific information.

`bufferPtr` Contains a pointer to a buffer that contains the source data to be transferred by the call or to the destination buffer where return data is to be placed by the call.

The `commandData` parameter and the contents of the data buffer pointed to by `bufferPtr` are different for each subcall.

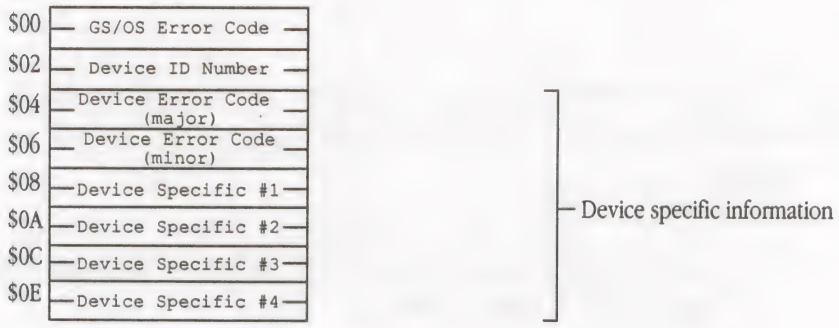
ReturnLastResult (DStatus subcall)

`controlCode` = $0005

Description This subcall provides an application with a means to query the driver for the results of the last device-specific call issued, provided that no other call has been issued since. Because the Apple SCSI drivers utilize auto-sensing (automatically issuing a Request Sense call when an error occurs), this is the only way to get accurate sense data from the device when using device-specific status and control subcalls. You can use this method after having issued an asynchronous call to verify that the call performed the requested action. For example, if an asynchronous format call is issued to a device, the application has no immedate means of verifying the success of the format. Issuing a ReturnLastResult subcall will indicate the success or failure of the format.

Parameters This call returns `requestLength` bytes up to 16 bytes maximum. The returned information format is shown in Figure 2-1.

■ **Figure 2-1** ReturnLastResult subcall return data



`GS/OS Error Code`

Reflects, as closely as possible, the problem encountered while performing the last requested transaction. GS/OS error codes are listed in Appendix B.

`Device ID Number`

Contains the SCSI device number.

`Device Error Code (major)`

Contains device-specific error information defined by and generated by the device. This word will be $00 if the request was successfully completed. The values are defined in Table 2-1.

■ **Table 2-1** Device error codes (major) definitions

Bit								
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Error**
r	r	0	0	0	0	0	r	Good; no error
r	r	0	0	0	0	1	r	Check condition
r	r	0	0	0	1	0	r	Good; condition met
r	r	0	0	1	0	0	r	Busy
r	r	0	1	0	0	0	r	Good; intermediate
r	r	0	1	0	1	0	r	Good; intermediate, condition met
r	r	0	1	1	0	0	r	Reservation conflict
r	r	1	0	1	0	0	r	Queue full

r = reserved

`Device Error Code (minor)`

A combination of several bits and the sense key, it provides a much more exact definition of the error status of the device. The bits and key are defined in Table 2-2.

■ **Table 2-2** Device error codes (minor) definitions

Bit	Value	Error
7	1	Indicates that the last command read a file mark. This is used only for sequential-access devices.
6	1	Indicates an end-of-medium condition. It is used in sequential-access and printer devices. This includes end-of-tape, beginning-of-tape, and out-of-paper conditions. Not used in direct-access devices.
5	1	Indicates that the requested logical block length did not match the logical block length of the data on the medium. (Refer to the ANSI X3.131-1986 SCSI specification for a discussion of logical block lengths.)
4	–	Reserved.
3–0	$0–$B	Sense key. These bits show the details of the device error code. The 16 possible sense keys are

Key	Definition
$0	No sense
$1	Recovered error
$2	Not ready
$3	Medium error
$4	Hardware error
$5	Illegal request
$6	Unit attention
$7	Data protect
$8	Blank check
$9	Vendor unique
$A	Copy aborted
$B	Aborted command
$C	Equal
$D	Volume overflow
$E	Miscompare
$F	Reserved

`Device Specific #1` and `Device Specific #2`

Used to form a longword (ordered from MSB to LSB); contain information about the failure. A few sample values are shown below. There are more than a hundred failure codes listed in the ANSI SCSI specification; refer to it for a complete list of failure codes.

Code	Error
1	The unsigned block address associated with the sense key.
2	The difference (residue) of the requested length minus the actual length in either bytes or blocks, as determined by the command. (Negative values are indicated by two's complement notation.)

`Device Specific #3`

Reserved; must be $00.

`Device Specific #4`

Contains the extended sense key that was returned by the device. This error information is device specific and defined by the vendor.

ReadTOC (DStatus subcall)

`controlCode` = $80C1

Description This subcall applies to the AppleCD SC only.

The ReadTOC subcall reads the table of contents information from the CD-ROM and returns it to the host. The format for this subcall is shown in Figure 2-2.

Parameters

■ **Figure 2-2** ReadTOC subcall format

Bit / Byte	7	6	5	4	3	2	1	0
$0	Opcode							$C1
$1	Flags							$00
$2	Track number							
$3	Reserved							$00
$4	Reserved							$00
$5	Type							
$6–$B	Reserved							$00

`Opcode` $C1

`Flags` $00

`Track number` Requested track number, 01–99 in binary-coded decimal (BCD)

Reserved Must be $00

`Type` Indicates the type of TOC requested by the call. The valid values and their meanings are

Value	Definition
$00	Requests the first and last user track number in BCD.
$01	Requests the starting address of the lead-out area (minutes, seconds, and frames) in BCD format.
$02	Requests the starting address of each track, starting with the track number specified in `Track number` (subcall byte $02). The quantity of track addresses transferred is equal to the value of `Allocation length` or the number of available track address bytes, whichever is less.

Reserved Must be $00

The three return data formats are shown in Figures 2-4 through 2-6. The parameters and the definitions are listed following each figure.

■ **Figure 2-3** ReadTOC subcall return data (TOC type $00)

Bit / Byte	7	6	5	4	3	2	1	0
$0	First User track number in BCD							
$1	Last User track number in BCD							

`User track number`
Two bytes that contain the first and last user track numbers on the disc.

■ **Figure 2-4** ReadTOC subcall return data (TOC type $01)

Bit / Byte	7	6	5	4	3	2	1	0
$0	Lead-out area starting address (minute) in BCD							
$1	Lead-out area starting address (second) in BCD							
$2	Lead-out area starting address (frame) in BCD							

`Lead-out area starting address`
Contains the starting address of the lead-out area (minutes, seconds, and frames) in BCD format.

■ **Figure 2-5** ReadTOC subcall return data (TOC type $02)

Byte \ Bit	7	6	5	4	3	2	1	0
$0	Reserved				Control			
$1	Starting address (minute) in BCD							
$2	Starting address (second) in BCD							
$3	Starting address (frame) in BCD							

`Control` Contains the type of tracks requested in this call. The tracks can be of several types: data; audio with preemphasis; audio without preemphasis. The options are

Bit 3	Bit 2	Bit 1	Bit 0	Definition
0	0	X	0	Two audio channels without preemphasis
0	0	X	1	Two audio channels with preemphasis
1	0	X	0	Four audio channels without preemphasis
1	0	X	1	Four audio channels with preemphasis
0	1	X	0	Data track
0	1	X	1	Reserved
1	1	X	X	Reserved
X	X	0	X	Digital copy protected
X	X	1	X	Digital copy protected

`Starting address`

Returns the starting address of each track for a range of tracks, starting from the track specified in the `Track number` byte. Addresses are returned in ascending sequential order until the number of bytes specified in the allocation length have been transferred or until all available data has been transferred to the host.

ReadQSubcode (DStatus subcall)

`controlCode` = $80C2

Description This subcall applies to the AppleCD SC only.

This subcall returns the Q Subcode data of either data tracks or audio tracks. The format for this subcall is shown in Figure 2-6.

Parameters

■ **Figure 2-6** ReadQSubcode subcall format

Byte \ Bit	7	6	5	4	3	2	1	0
$0	Opcode							$C2
$1	Flags							$00
$2	Reserved							$00

`Opcode` $C2

`Flags` $00

Reserved Must be $00.

The ReadQSubcode return data format is shown in Figure 2-7.

■ **Figure 2-7** ReadQSubcode subcall return data (TOC type $02)

Bit / Byte	7	6	5	4	3	2	1	0
$0	Reserved				Control			
$1	Track number							
$2	Index number							
$3	Address (minute)							
$4	Address (second)							
$5	Address (frame)							
$6	Address (abs. minute)							
$7	Address (abs. second)							
$8	Address (abs. frame)							

`Control` Contains the type of tracks requested. See the ReadTOC subcall for a description of this field.

`Track number` Specifies the current track number, between 1 and 99 in BCD notation.

`Index number` Specifies the index number assigned to the current track.

`Address` Contains the relative running time in minutes, seconds, and frames from the beginning of the track.

`Address (abs)` Contains the absolute running time in minutes, seconds, and frames from the beginning of the disc.

ReadHeader (DStatus subcall)

`controlCode` = $80C3

Description This subcall applies to the AppleCD SC only.

The ReadHeader subcall returns 4 bytes of header information for the specified logical block address on the CD-ROM. The format for this subcall is shown in Figure 2-8.

Parameters

■ **Figure 2-8** ReadHeader subcall format

Bit / Byte	7	6	5	4	3	2	1	0
$0	Command code							$C3
$1	Command flags							$00
$2	Block address							
$3–$B	Reserved							$00

`Opcode` $C3

`Flags` $00

`Block address` Contains the number of the requested logical block.

Reserved Must be $00.

The return data format is shown in Figure 2-9.

■ **Figure 2-9** ReadHeader subcall return data

Byte \ Bit	7	6	5	4	3	2	1	0
$0	Logical block address (MSB)							
$1	Logical block address							
$2	Logical block address							
$3	Logical block address (LSB)							
$4	Mode							

`Logical block address`
Four bytes that contain the absolute address (minutes, seconds, and frames from the beginning of the disc) in BCD format of the requested logical block.

`Mode` Contains the mode of the requested block.

AudioStatus (DStatus subcall)

`controlCode` = $80CC

Description This subcall applies to the AppleCD SC only.

The AudioStatus subcall returns the audio play status and the starting address of the next track. The format for this subcall is shown in Figure 2-10.

Parameters

■ **Figure 2-10** AudioStatus subcall format

Bit / Byte	7	6	5	4	3	2	1	0
$0	Opcode							$CC
$1	Flags							$00
$2–$B	Reserved							$00

`Opcode` $CC

`Flags` $00

Reserved Must be $00.

The AudioStatus return data format is shown in Figure 2-11.

■ **Figure 2-11** AudioStatus subcall return data

Bit / Byte	7	6	5	4	3	2	1	0
$0	Status							
$1	Reserved				Play mode			
$2	Reserved					Control		
$3	Address (abs. minute)							
$4	Address (abs. second)							
$5	Address (abs. frame)							

`Status` Reflects the current audio status of the AppleCD SC. The possible returned values and their meanings are

Value	Meaning
$00	The device is in AudioPlay mode following the execution of an AudioSearch subcall or AudioPlay subcall.
$01	The device is in pause mode.
$02	The device is in mute mode following the execution of an AudioSearch subcall or AudioPlay subcall.
$03	The device has just completed an AudioPlay operation.
$04	An error occurred during AudioPlay operation.
$05	AudioPlay operation not requested.

`Play mode` Determines which recorded audio channels are sent to which output channels. See the description of the AudioSearch subcall later in this chapter for an explanation of this field.

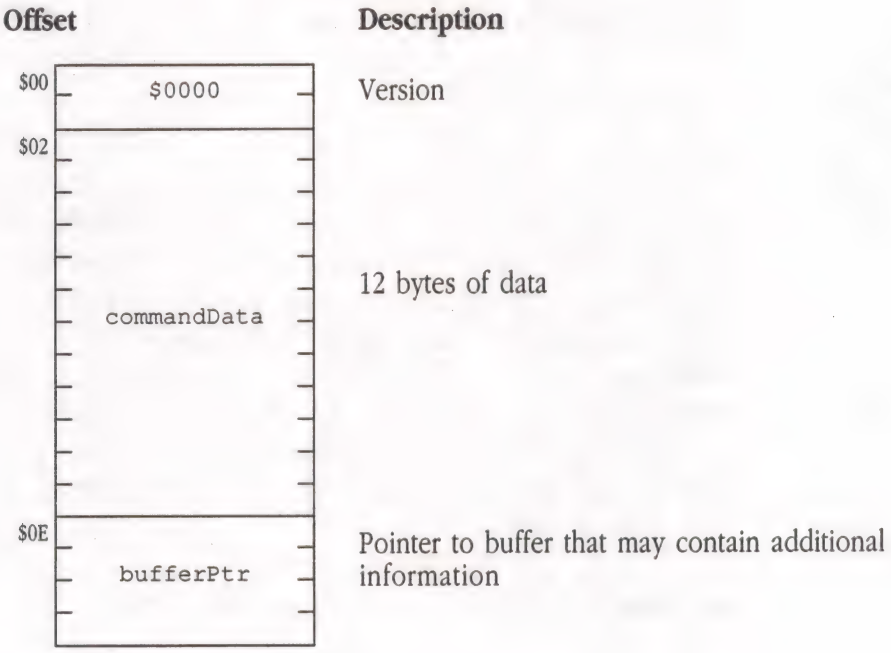
`Control` Shows the kind of information that is on the current track. See the ReadTOC subcall for a description of the `Control` field bits.

`Address` Last 3 bytes of return data; contains the current absolute address (minutes, seconds, and frames from the beginning of the disc) in BCD format.

DControl ($202E)

The SCSI driver supports all standard device-specific subcalls. Please see Chapter 1, "GS/OS Device Call Reference," for a description of the general format of the DControl call.

All of the SCSI device-specific subcalls use this same format for the control list (the buffer pointed to by `controlListPtr`):



`Version` Contains version of the parameter list. If you are using data-chaining commands, this version must be $0001. Otherwise, it should be $0000.

`commandData` Contains subcall-specific information.

`bufferPtr` Contains a pointer to a buffer that contains the source data to be transferred by the call or to the destination buffer where return data is to be placed by the call.

The `commandData` parameter and the contents of the data buffer pointed to by `bufferPtr` are different for each subcall.

AssignPartitionOwner (DControl subcall)

`controlCode` = $0004

Description

This subcall provides FSTs the ability to assign a new FST owner to a device partition. It does this by writing an ASCII string to the partition map, but it does not alter any other information on the map.

The AssignPartitionOwner subcall is supported by block devices that allow partitioned media. This call is initiated by an FST as a result of the EraseDisk system call in order to label the partition with the FST type.

Parameters

`Partition type`

The input parameter consists of an ASCII string indicating the partition type. The string can be up to 32 bytes in length (not case sensitive). If the string is less than 32 bytes in length, it must be terminated with a null character. A partition type can be cleared by setting the first byte of the string to $00 (the null character). The driver reassigns the current partition to the new owner. This subcall does not reassign physical block allocation within a device partition descriptor. Block devices utilizing nonpartitioned media and character devices return with no error. Figure 2-12 shows an example of a `Partition type` ASCII string.

■ **Figure 2-12** Example partition type ASCII string

Partition owner names are assigned by Apple Developer Technical Support and can be up to 32 bytes in length—uppercase and lowercase characters are considered equivalent. The control list for this call consists of a GS/OS ASCII string containing the partition owner name. Here are some examples of partition type names:

- Apple_MFS
- Apple_HFS
- Apple_Unix_SVR2
- Apple_partition_map
- Apple_Driver
- Apple_ProDOS
- Apple_Free
- Apple_Scratch

AudioSearch (DControl subcall)

`controlCode` = $80C8

Description

This subcall applies to the AppleCD SC only.

The AudioSearch subcall provides a means for positioning the optical pickup at an address and returns a status byte if and when the address is found. The format for this subcall is shown in Figure 2-13.

Parameters

■ **Figure 2-13** AudioSearch subcall format

Bit / Byte	7	6	5	4	3	2	1	0
$0	Opcode							$C8
$1	Flags							$00
$2	Play flag							
$3	Play mode							
$4	Search address (MSB)							
$5	Search address							
$6	Search address							
$7	Search address (LSB)							
$8	Type							
$9–$B	Reserved							$00

`Opcode` $C8

`Flags` $00

`Play flag` $00 = pause after search complete
$01 = play after search complete

`Play mode` Determines which recorded audio channels are sent to which audio output connectors. All the possible combinations are shown in Table 2-3.

■ **Table 2-3** Play modes

play mode	Definition
$00	Play with muting on, no audio output
$01	Play right channel only through right output
$02	Play left channel only through right output
$03	Play both channels through right output
$04	Play right channel through left output
$05	Play right channel through both outputs
$06	Play right channel through left output, left channel through right output
$07	Play right channel through left output, both channels through right output
$08	Play left channel through left output
$09	Play left channel through left output, right channel through right output (stereo)
$0A	Play left channel through both outputs
$0B	Play left channel through left output, both channels through right output
$0C	Play both channels through left output
$0D	Play both channels through left output, right channel through right output
$0E	Play both channels through left output, left channel through right output
$0F	Play both channels through both outputs

`Search address`
Address at which the search begins.

`Type` Specifies in which format the values in the `Search address` field are given. The valid values and their meanings are

Type	Address format
$00	Address is in logical-block address format.
$40	Address is in absolute minutes, seconds, and frames format.
$80	Address is in track-number format.
$C0	Reserved.

Reserved Must be $00.

The three return data formats are shown in Figures 2-14 through 2-16. The parameters and the definitions are listed following each figure.

■ **Figure 2-14** AudioSearch subcall return data for search type $00

Bit / Byte	7	6	5	4	3	2	1	0
$0	Logical block address (MSB)							
$1	Logical block address							
$2	Logical block address							
$3	Logical block address (LSB)							

`Logical block address`
Specifies the block address at which the AudioPause or AudioPlay subcall commences.

■ **Figure 2-15** AudioSearch subcall return data for search type $01

Bit / Byte	7	6	5	4	3	2	1	0
$0	Reserved							$00
$1	Address (abs. minute)							
$2	Address (abs. second)							
$3	Address (abs. frame)							

Reserved Must be $00.

`Address` Specifies the absolute address (minutes, seconds, and frames from the beginning of the disc) in BCD format, at which the AudioPause or AudioPlay subcall commences. Valid values for these parameters are 00–99 minutes, 00–59 seconds, and 00–74 frames.

■ **Figure 2-16** AudioSearch subcall return data for search type $02

Bit / Byte	7	6	5	4	3	2	1	0
$0-$2	Reserved							$00
$3	Track number							

Reserved Must be $00.

`Track number` Specifies the absolute track address in BCD format at which the AudioPause or AudioPlay subcall commences. Limits of the parameter are from 01 to 99 in BCD.

AudioPlay (DControl subcall)

`controlCode` = $80C9

Description This subcall applies to the AppleCD SC only.

This subcall allows you to position the optical pickup at a specified address and to play audio through the audio output.

You can also use the AudioPlay subcall to release the drive from a pause state after the execution of an AudioSearch subcall or from a pause state after the execution of an AudioPause subcall. The AudioPlay subcall can also be used to set a particular completion address or play mode. The format for this subcall is shown in Figure 2-17.

Parameters

■ **Figure 2-17** AudioPlay subcall format

Bit / Byte	7	6	5	4	3	2	1	0
$0	Opcode							$C9
$1	Flags							$00
$2	Stop flag							
$3	Play mode							
$4	Playback address (MSB)							
$5	Playback address							
$6	Playback address							
$7	Playback address (LSB)							
$8	Type							
$9–$B	Reserved							$00

`Opcode` $C9

`Flags` $00

`Stop flag` Indicates whether the address supplied in the `Playback address` field is a start address or a stop address. The valid values and their meanings are
$00 = `Playback address` is used as a start address
$10 = `Playback address` is used as a stop address

`Play mode` See the description of the AudioSearch subcall earlier in this chapter for a description of these bits.

`Playback address`
Contains the address at which the drive begins or stops, depending on the value in the `Stop flag` field.

`Type` Specifies in which format the values in the `Address` field are given. The valid values and their meanings are

Type	Address format
$00	Address is in logical-block address format.
$40	Address is in absolute minutes, seconds, and frames format.
$80	Address is in track-number format.
$C0	Reserved.

Reserved Must be $00.

AudioPause (DControl subcall)

`controlCode` = $80CA

Description This subcall applies to the AppleCD SC only.

The AudioPause subcall temporarily stops audio play operation and enters the hold-track state, maintaining the same address.

These subcalls can terminate the AudioPause subcall: Rezero Unit, Read, Seek, Start/Stop Unit, Send Diagnostics, Prevent/Allow Media Removal, Read Extended, ExtendedSeek, Verify, Eject, ReadHeader, AudioPlay (stop bit = 0), AudioPause, and AudioScan.

These subcalls will not interrupt the AudioPause subcall: TestUnitReady, RequestSense, Inquiry, Reserve, Release, ModeSelect, ModeSense, ReadCapacity, ReadTOC, ReadQSubcode, AudioStatus, AudioPlay (stop bit = 1), and AudioStop.

The format of the AudioPause subcall is shown in Figure 2-18.

Parameters

■ **Figure 2-18** AudioPause subcall format

Bit / Byte	7	6	5	4	3	2	1	0
$0	Opcode							$CA
$1	Flags							$00
$2	Pause flag							
$3–$B	Reserved							$00

`Opcode` $CA

`Flags` $00

`Pause flag` Contains the flag that determines whether this subcall pauses the CD audio play or releases the drive from a pause. The valid values and their meanings are

Value	Definition
$00	Release pause
$10	Initiate pause

This subcall is valid only if audio play is in progress and only after an AudioSearch or AudioPlay subcall has executed.

Reserved Must be $00.

AudioStop (DControl subcall)

`controlCode` = $80CB

Description

This subcall applies to the AppleCD SC only.

The AudioStop subcall specifies the address at which the audio play terminates. The drive stops spinning, and the optical pickup remains in the approximate area of the `Stop address`. The format for this subcall is shown in Figure 2-19.

Parameters

■ **Figure 2-19** AudioStop subcall format

Bit / Byte	7	6	5	4	3	2	1	0
$0	Opcode							$CB
$1	Flags							$00
$2	Stop address (MSB)							
$3	Stop address							
$4	Stop address							
$5	Stop address (LSB)							
$6	Type		Reserved					
$7-$B	Reserved							$00

`Opcode` $CB

`Flags` $00

`Stop address` Specifies the address at which the drive shuts down. If this field contains $00, and the `Type` field also contains $00, the drive will perform a SCSI Rezero Unit SCSI call. See the ANSI X3.131-1986 SCSI specification for details on this call.

`Type` Specifies in which format the values in the `Stop address` field are given. The valid values and their meanings are

Type	Address format
$00	Address is in logical-block address format.
$40	Address is in absolute minutes, seconds, and frames format.
$80	Address is in track-number format.
$C0	Reserved.

Reserved Must be $00.

AudioScan (DControl subcall)

`controlCode` = $80CD

Description This subcall applies to the AppleCD SC only.

The AudioScan subcall performs a fast-forward or fast-reverse scan operation beginning at a specified address. Audio output remains enabled normally. The format for this subcall is shown in Figure 2-20.

Parameters

■ **Figure 2-20** AudioScan subcall format

Bit / Byte	7	6	5	4	3	2	1	0
$0	Opcode							$CD
$1	Flags							$00
$2	Direction							
$3	Reserved							$00
$4	Start address (MSB)							
$5	Start address							
$6	Start address							
$7	Start address (LSB)							
$8	Type							
$9-$B	Reserved							$00

`Opcode` $CD

`Flags` $00

`Direction` Determines in which direction the scan takes place. The valid values and their meanings are
$00 = fast forward
$10 = fast reverse

Reserved Must be $00.

`Start address` The scan begins at the address specified in this field.

`Type` Specifies in which format the values in the `Start address` field are given. The valid values and their meanings are

Type	Address format
$00	Address is in logical-block address format.
$40	Address is in absolute minutes, seconds, and frames format.
$80	Address is in track-number format.
$C0	Reserved.

Reserved Must be $00.

Data chaining

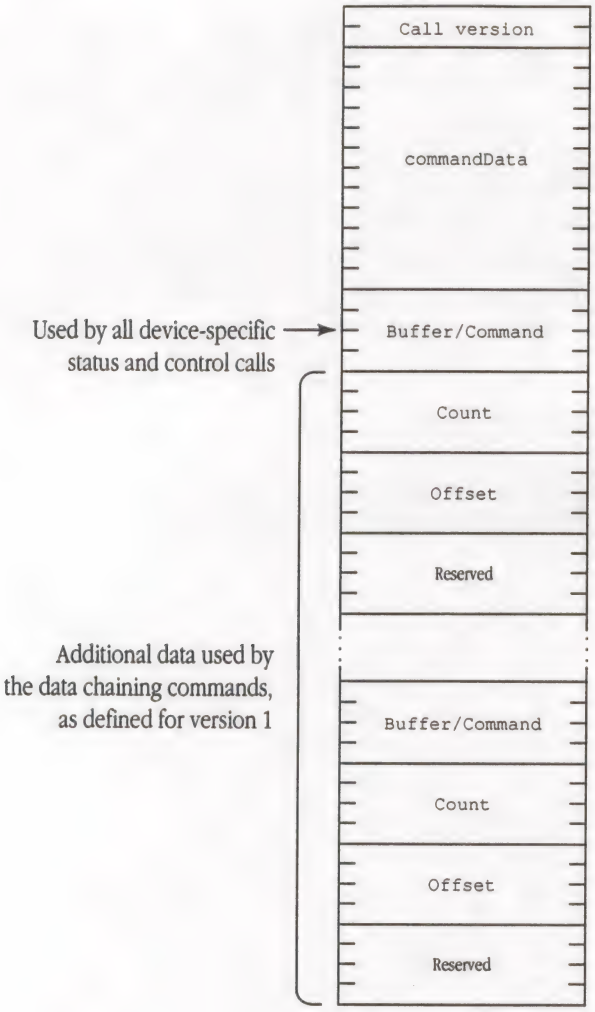
There is a special class of status codes specific to the SCSI driver. These commands instruct the SCSI Manager to move data *very* quickly from location to location by use of data chaining. The commands are explained in this section.

Data chaining is instituted by means of four new commands to the SCSI Manager: DCMove, DCLoop, DCSpecial, and DCStop. The DCMove command provides an address to move data to at a very fast rate. The DCLoop command provides a means for repeatability at a very small cost in code. The DCSpecial command provides a means for jumping to an auxiliary routine. The DCStop command terminates the data-chaining structure.

To issue a single command, use the status list shown in Figure 2-21.

The `bufferPointer` parameter on direct page points to this structure, and the `requestCount` parameter, also on direct page, contains the total number of bytes requested by the command.

■ **Figure 2-21** Status list using data-chaining commands



`Call version`

To maintain compatibility between previously written SCSI drivers and GS/OS, this field is provided to distinguish between versions of SCSI device-specific commands. The first SCSI driver release (GS/OS system software version 4.0 and previous versions) supports version $0000 commands only and uses only the `Call version` field, the `Command data` field, and the `Buffer Command` field. The current version of the driver supports version $0001 commands and uses all fields.

`commandData`

The 12-byte command data as outlined in the definition of the driver command being issued.

`Buffer/command`

Contains a pointer to the source or destination data buffer, depending on the command.

The data-chaining commands are each composed of four longword fields and are in low-byte-first format. By using the commands in various combinations, data can be gathered or distributed depending on the direction of transfer.

The address contained in the field is also used to determine which of the three data chaining commands will be used and are interpreted in one of three ways:

1. If the value is in the range of $00000001–$FFFFFFFE, the command is a DCMove command, and the value is treated as a buffer address.
2. If the value is $FFFFFFFF, the command is a DCLoop command.
3. If the value is $00000000, the command is a DCSpecial command.

`Count`

Depending on what is in the `Buffer/command` field, the next field is interpreted in one of three ways:

1. If a DCMove command is indicated, this value is a request byte count.
2. If a DCLoop command is indicated, this value is a count for the number of times to loop through the buffer entries minus 1.
3. In the case of a DCSpecial command, this field is interpreted in one of two ways: It either terminates a DCLoop command or calls a user-defined routine.

Value	Definition
$00000000	DCStop command; execution halts
Other	Address of a routine to call

The user-defined routine can perform an action such as a graphics page flip or any other task required in midtransfer.

`Offset` — Has three different definitions depending on the contents of the `Buffer/Command` field. The definitions are

1. If a DCMove command is indicated, this value is added to the buffer address (in the `Buffer/Command` field), so that the buffer address points to a new buffer for the next pass.
2. If a DCLoop command is indicated, this value is the number of entries (not the number of *bytes*) to go back to the beginning of the loop.
3. If the command is a DCSpecial command, this field is $00000000, which represents a DCStop command, and terminates the loop. If the command is not a DCSpecial command, this field can be used by the called routine for any purpose.

Reserved — This field is reserved and must contain $00000000 in all cases.

- *Note:* The total number of bytes transferred (the number of bytes requested times the loop count plus any other counts) may not be larger than the `Count` used when issuing the command. If the number of bytes transferred is larger than `Count`, a time-out error is returned, the data-chaining structures are modified, and the data returned is unreliable.

Table 2-4 contains the data-chaining instructions.

Table 2-4 Data-chaining commands

Instruction	Buffer/command	Count	Offset	Pointer
DCMove	Buffer address	Buffer size	Value added to buffer address	Null
DCLoop	$FFFFFFFF	Loop count	Offset to another instruction	Null
DCStop	$00000000	$00000000	$00000000	Null
DCSpecial	$00000000	Address	Application defined	Null

Some data-chaining examples

Some examples will help show how data chaining can be implemented. Two examples for using these commands, one for sending data and the other for receiving data, are given here.

Sending data

In this example, a printer is connected to the SCSI interface, and the pointer requires a header to be sent prior to sending the image to the printer. The image resides in memory as contiguous pages and is rather large.

Rather than breaking up the data to be printed and collating it with copies of the header information, the application could instead use the following data-chaining commands.

Command	Opcode	Count	Offset	Pointer
DCMove	$00025C88	$00000200	$00000000	$00000000
DCMove	$001C0138	$00004000	$00004000	$00000000
DCLoop	$FFFFFFFF	$00000019	$FFFFFFFE(–2)	$00000000
DCStop	$00000019	$00000000	$00000000	$00000000

The first DCMove command is the pointer to the header information that is sent to the printer each time through the loop. It is 512 bytes in size, and we want to send the same header data on each pass of the loop.

The second DCMove command points to the first page image to be sent. Each page is 16 KB in size, and we want to send page 2—which is contiguous in memory—after page 1 on the next pass of the loop.

The DCLoop command executes a loop $1A times and goes back two instructions ($FFFFFFFE = –2) to the first DCMove command each time. After the loop executes the $1Ath time, it drops through to the next command, which is the DCStop command. This command signals the end of the transfer, and the call returns via the normal path to the application.

- *Note:* The first two commands will already have been executed once prior to the DCLoop command. The DCLoop command then instructs the SCSI Manager to do this $19 more times, for a total of $1A executions of the DCMove instructions.

Receiving data

In this example, an imaging device (an image scanner, for example) is connected to the SCSI interface and sends a large amount of image data, which we want to display. The video mapping in Apple II Hi-Res graphics mode is not structured in a way that allows an image to be read in directly and displayed; the data must first be remapped to the noncontiguous video memory in the Apple II computer. By structuring the data-chaining commands, a Hi-Res bitmap image can be read directly into video memory without having to first be manipulated.

Rather than using multiple unique Read calls to the device to achieve the correct mapping, and thereby spending a lot of time in CPU overhead, the application could instead use the following data-chaining commands:

Command	Opcode	Count	Offset	Pointer
DCMove	$00002000	$00000028	$00000080	$00000000
DCMove	$00002400	$00000028	$00000080	$00000000
DCMove	$00002800	$00000028	$00000080	$00000000
DCMove	$00002C00	$00000028	$00000080	$00000000
DCMove	$00003000	$00000028	$00000080	$00000000
DCMove	$00003400	$00000028	$00000080	$00000000
DCMove	$00003800	$00000028	$00000080	$00000000
DCMove	$00003C00	$00000028	$00000080	$00000000
DCLoop	$FFFFFFFF	$00000007	$FFFFFFF8(–8)	$00000000
DCSpecial	$00000000	`[AdjustProc]`	$00000000	$00000000
DCLoop	$FFFFFFFF	$00000002	$FFFFFFF6(–10)	$00000000
DCStop	$00000000	$00000000	$00000000	$00000000

The first DCMove command is the pointer to $2000, the first line of Apple II Hi-Res video memory. This increments by $80 on each pass through the loop.

The second DCMove command is the pointer to $2400, the second line of video memory. This also increments by $80 on each pass.

The third DCMove command is the pointer to $2800, the third line of video memory. This increments by $80 on each pass.

These values are duplicated for all eight DCMove commands, the last of which points to $3C00.

The first DCLoop command is set to loop seven times and goes back eight instructions ($FFFFFFF8 = –8) to the first DCMove command after each pass through the loop. When the loop executes the eighth pass (the initial pass plus seven loops back equals eight passes total), it drops through to the next command, which is the DCSpecial command. It contains the address of the application-defined `AdjustProc` routine.

The `AdjustProc` routine is called after each 64 scan lines have been drawn to the screen (8 scan lines times eight passes through the inner loop equals 64 scan lines) to adjust the pointers for the next group of 64 scan lines. There are a total of three groups of 64 scan lines on the video screen. The `AdjustProc` routine adjusts the buffer addresses within the data-chaining structure to point to the next group of 64 scan lines. This adjustment is required because of the interleaved nature of the Apple II Hi-Res video memory. (See the *Apple IIGS Hardware Reference* for more information on Apple IIGS video graphics.)

`AdjustProc` is shown here.

```
AdjustProc      Proc
        ldy     #0      ;init index
        ldx     #8      ;init counter
@loop
        lda     statusList+14,y       ;get the value in Field1
        sec
        sbc     #$3D8   ;adjust to first address of next group
        sta     statusList+14,y
        tya             ;adjust index
        clc
        adc     #16
        tay
        dex             ;decrement counter
        bne     @loop   ;loop until all 8 addresses are fixed

; now we fix up the count field in the first DCLoop instruction.
; This is located at (statusList+14) (beginning of DataChain structure)
;                 plus 80 ;(offset of DCLoop instruction within structure)
;                 plus 04 ;(offset of count within DCLoop instruction)

        lda     #7              ;adjust the count in the first DCLoop instruction
        sta     statusList+14+80+4
        rtl
        EndP
```

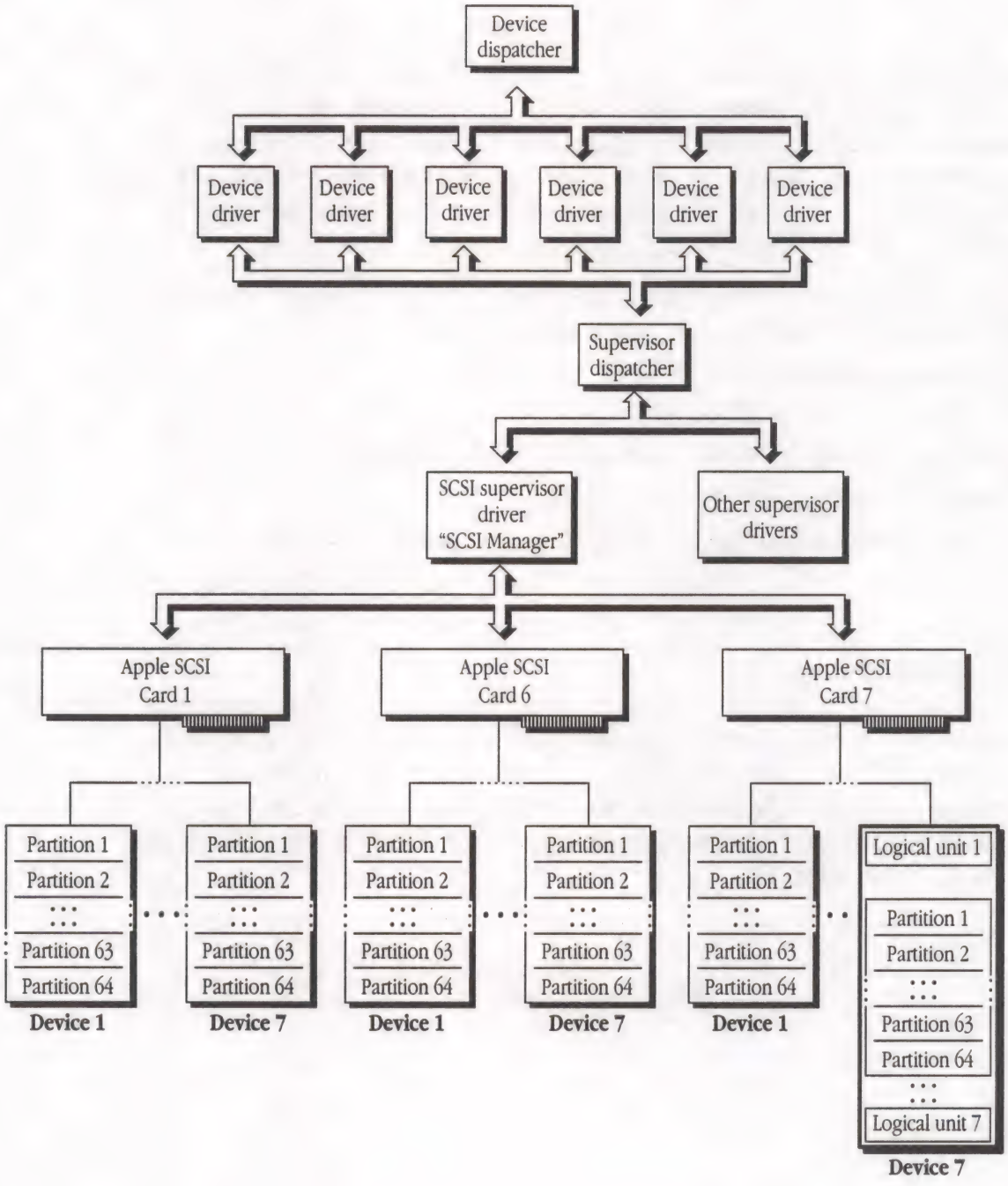
- *Note:* The SCSI Manager always checks data-chaining commands for validity prior to execution. If any command is unacceptable, the driver returns with the carry bit set, and error in the A register.

The SCSI Manager

The SCSI Manager is a supervisory driver that controls all communications on the SCSI bus by managing arbitration between SCSI device drivers and SCSI peripherals. All SCSI driver calls are issued to the SCSI Manager. If you plan on writing your own SCSI driver for use with the Apple SCSI Card, you must understand the SCSI Manager calls described in this section.

It is the job of the SCSI Manager to arbitrate and manage Apple SCSI peripheral cards, physical SCSI devices, and logical SCSI devices. However, it does not manage partitions; that task belongs to the drivers. In order to access a partition on a physical SCSI block device or a logical SCSI block device, the driver must translate logical block numbers to physical block numbers. Figure 2-22 shows the relationship of the SCSI Manager to drivers and devices in GS/OS.

■ **Figure 2-22** SCSI Manager



The SCSI Manager supports the Apple II SCSI Card and the Apple II High-Speed SCSI Card.

- *Note:* The SCSI Manager will support the original Apple II SCSI Card only if it has revision C firmware on the card. The Apple part number for this ROM is 341-0437-A.

You can plug seven cards into a system and connect seven devices to each card. Although the SCSI Manager does not "know" about partitions, room has been reserved to allow device drivers to access a maximum of 32 partitions. Therefore, even though there are only seven possible SCSI target devices on each SCSI bus, many more are available through multiple cards and partitions. Refer to Figure 2-22 for a pictorial representation of possible devices supported by the SCSI Manager.

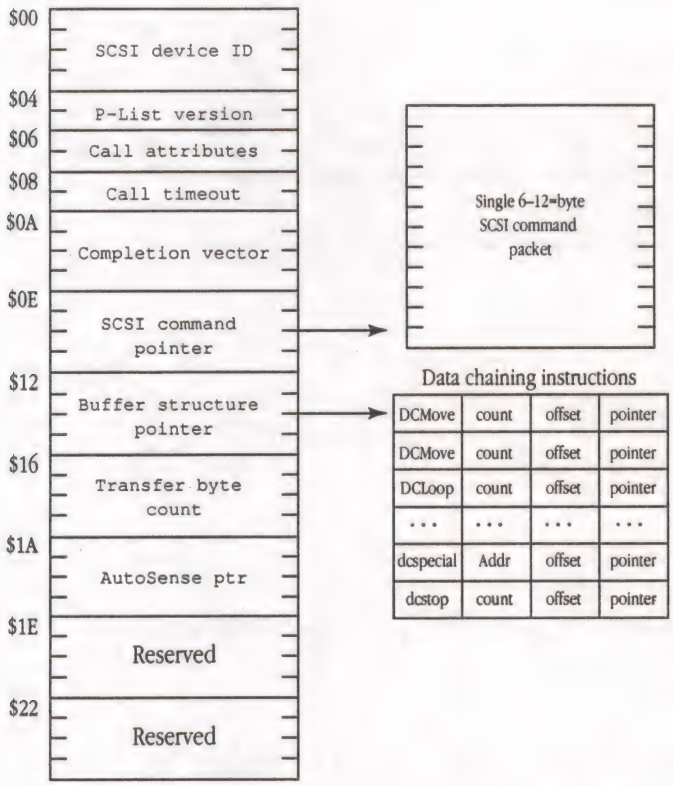
The SCSI Manager sends calls and data to SCSI devices using direct memory access (DMA). The following are the major areas the SCSI Manager controls:

- manages hardware resources
- supports startup and shutdown calls
- provides a call interface between SCSI drivers and the SCSI devices
- supports the built-in NCR 5380 SCSI controller IC
- provides low-level phase support, giving complete control of the SCSI bus

The SCSI data model

In addition to the required calls for supervisory drivers, there is one call to access SCSI target devices. The SCSI I/O call handles all communication between initiators and targets. Device drivers make the I/O call with a set parameter list, and the SCSI Manager will manage the communication between the system and the target. Figure 2-23 shows the format of the parameter list.

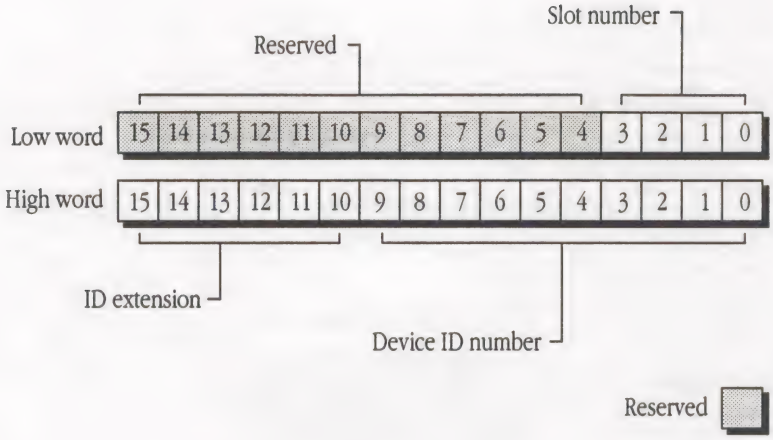
■ **Figure 2-23** The SCSI data model for the I/O call



`SCSI device ID`

Longword wide field that contains the ID given devices by the SCSI Manager. The ID is defined by the SCSI Manager for physical SCSI devices and logical units within SCSI devices. The driver-defined ID extension is used by block device drivers to differentiate partitions on block devices. The driver-defined ID extension is set to 0 and should not be modified by the driver unless the physical or logical device contains partitions on it. If the device contains partitions, the driver-defined ID extension must be unique for each partition on that device. When making a call to the SCSI Manager, the driver must set bits A through F to 0. See Figure 2-24 for a depiction of the device ID word.

■ **Figure 2-24** Device ID word



`P-List/version`
Contains a version word defining the type of parameter block. Must be set to $00.

`Call attributes`
A bit-encoded word describing how the I/O call is to be handled. Refer to the description of the SCSI I/O call later in this chapter for a description of the bits.

`Call timeout` Contains the number of 1/4-second intervals the SCSI driver waits before aborting the I/O call. If the call does not complete within this specified time, an error is returned to the driver.

`Completion vector`
When a call completes, the SCSI Manager calls the driver through this vector to signal completion of the call.

`SCSI command pointer`
Points to one or more SCSI command packets. Each packet can be 6, 10, or 12 bytes in size. The data contained in the command packet is defined in the ANSI X3.131-1986 SCSI specification.

`Buffer structure pointer`
Long pointer that contains the address of a list of data-chaining instructions. Refer to "Data Chaining" earlier in this chapter for a detailed description.

`Transfer byte count`
: Contains the total number of bytes requested.

`AutoSense ptr`
: Contains a pointer to a buffer where the returned sense information is placed as a result of a RequestSense device call. The sense information contains device-specific details of an error reported by the device.

SCSI Manager calls

The SCSI Manager accepts SCSI driver calls from the supervisor dispatcher and SCSI device drivers. Calls from device drivers are the only means of communicating with SCSI devices; all calls and data are passed to devices using the SCSI Manager I/O call.

The GS/OS SCSI Manager calls available to SCSI device drivers are

- RequestDevices
- ClaimDevices
- I/O

When a driver makes a call to the SCSI Manager, control is passed to the manager through a system service vector defined by GS/OS. When control is returned to the driver at the end of the call, the accumulator contains $0000 (no error) or an error code that the driver must interpret and translate into a valid GS/OS error code. If an error does occur, the carry bit will be set; otherwise, it will be clear.

RequestDevices ($0002)

Description

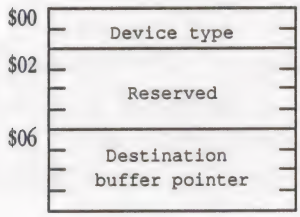
This call returns the number of devices the SCSI Manager found on line during the SCSI Manager startup for a specific SCSI peripheral device type.

The driver must provide a buffer of $704 bytes in size to hold a maximum of 49 devices (seven Apple SCSI peripheral cards with seven physical SCSI devices on each card; see Figure 2-22).

Parameters

A register:	SCSI Manager ID	$0000
X register:	Call opcode	$0002
GS/OS DirectPage:	`SIB pointer`	
	`Parameter list pointer`	

■ **Figure 2-25** RequestDevices input parameter list



`Device type` Word defining SCSI devices about which information is being requested. Bits 0–7 define the device type.

Type	Device
$00	Direct-access device (for example, magnetic disk)
$01	Sequential-access device (for example, magnetic tape)
$02	Printer device
$03	Processor device
$04	Write-once read-multiple device (for example, some optical disks)

[continued]

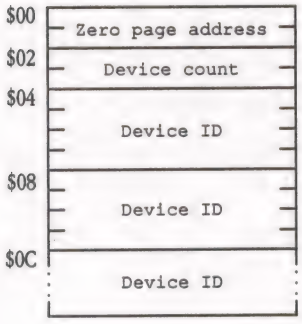
Type	Device [continued]
$05	Read-only direct-access device (for example, some optical disks)
$06	Scanner device
$07	Optical memory devices
$08	Changer devices (for example, CD jukeboxes)
$09	Communications devices
$0A–0F	Reserved
$10	Apple Tape Drive (and 3M Corp. MCD/40)
$11–1D	Reserved
$1E	Target device
$1F	Unknown device type

Reserved Must be 0.

`Destination buffer pointer`
Longword field; contains the pointer to a $704-byte buffer that contains the list of devices requested.

The return buffer contains all the information about all SCSI devices of the type requested. Figure 2-26 shows the format of the buffer, and each parameter is described immediately following the figure.

■ **Figure 2-26** RequestDevices return buffer format



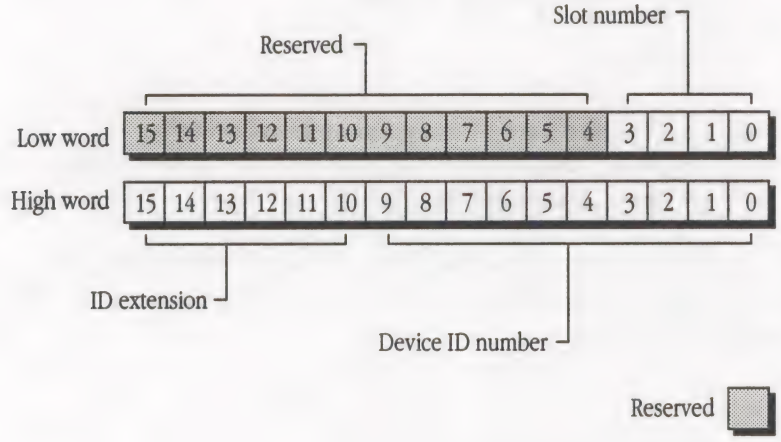
`Zero page address`

Contains the address of a scratch zero page that can be used by device drivers calling the SCSI Manager. This zero page is valid only during the call, and drivers should not count on data's remaining valid between calls to the driver.

`Device count` Word field that contains the number of target devices that match the specified device type returned to the caller.

`Device ID` Contains several subfields. Figure 2-27 shows the bits in the longword, and a description of each subfield follows the illustration.

■ **Figure 2-27** RequestDevices device ID longword



Word	Bits	Description
Low	$0–3	Slot number for this device. Slots $0–7 are internal slots; slots $8–F are external slots.
Low	$4–F	Reserved; must be 0.
High	$0–9	Device ID (defined by SCSI Manager).
High	$A–F	ID extension (defined by driver).

The high word contains the unit number assigned to the device by the SCSI Manager. Note that the device ID matches the slot and unit number definitions for DIBs. The device ID is assigned by the SCSI Manager to identify devices it finds on the SCSI bus. The driver-defined ID extension field is provided for support of SCSI block devices with multiple partitions. It is set to 0 by the SCSI Manager and can be changed by SCSI block device drivers that support partitions. It gives drivers a means to make each partition on a specific SCSI device unique. The SCSI device ID must be unique; multiple devices with the same SCSI device ID are not allowed. The first ID extension must be 0.

Errors

$FE01	Invalid GS/OS SCSI device type

ClaimDevices ($0003)

Description

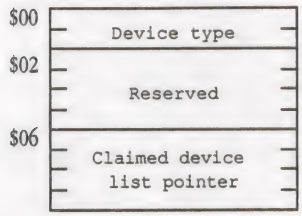
This call provides a means for drivers to request access to devices on the bus. This process is called *claiming* a device. The driver informs the SCSI Manager which devices are being claimed by setting the high bit in the device ID.

The parameter list for this call is identical to the parameter list for the RequestDevices call. The `bufferPointer` parameter points to a buffer that contains the same list as that returned from the RequestDevices call with the exception of the high-order bit of the device ID. For each device being claimed by the driver, the high-order bit of the device ID must be set by the driver to indicate a claimed device. If the bit is 0, the device is free and can be claimed by another driver. Drivers must claim a device in order to access the device. If the driver does not claim the device and accesses it, there may be a conflict; unpredictable (and potentially unpleasant) results may occur. The only way to guarantee that a driver has sole access to a device is to claim it.

Parameters

A register:	SCSI Manager ID	$0000
X register:	Call opcode	$0003
GS/OS DirectPage:	`SIB pointer`	
	`Parameter list pointer`	

■ **Figure 2-28** ClaimDevices parameter list



`Device type` Byte that defines SCSI devices. Bits 0–7 define the device type. See "RequestDevices ($0002)," earlier in this chapter, for a complete list of device types.

Reserved Must be $00.

`Claimed devices list pointer`
Longword pointer that contains the address of the $704-byte buffer that holds the list of devices of the type being requested.

The ClaimDevices return buffer is identical to the RequestDevices return buffer. Refer to "RequestDevices ($0002)," earlier in this chapter, for a description of the contents of the return buffer.

Errors $FE03 Device ID not found

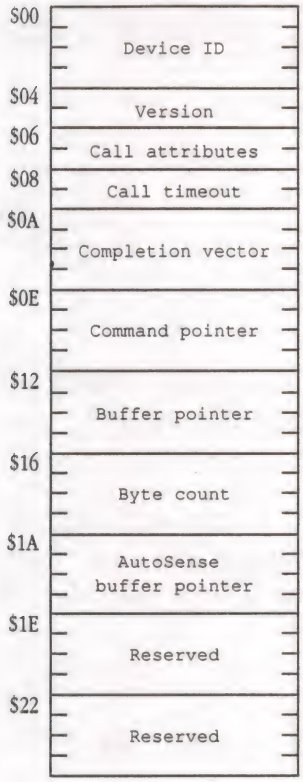
I/O ($0004)

Description All SCSI drivers issue the I/O call to the SCSI Manager each time the driver receives a driver call that accesses a SCSI device. Figure 2-29 shows the format of the call parameter list that is pointed to by the `statusList` pointer.

Parameters

A register:	SCSI Manager ID	$0000
X register:	Call opcode	$0004
GS/OS DirectPage:	Address of GS/OS direct page	

■ **Figure 2-29** I/O call parameter list

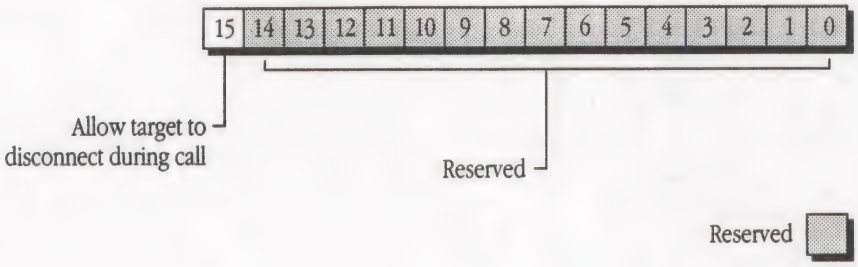


`Device ID` Contains a longword identifying the device requested. This is the ID given to the driver during the RequestDevices call.

`Version` Contains the block version number. The number identifies the version of the parameter block (this block) used by the SCSI Manager. Should be set to $0000 for GS/OS system software version 5.0.

`Call attributes`
If bit 15 is set, the SCSI Manager allows the device to disconnect if it chooses to do so during the call. If bit 15 is clear, the SCSI Manager will not allow the device to disconnect. The call attributes word is depicted in Figure 2-30.

■ **Figure 2-30** I/O call attributes word



`Call timeout` Word field defining the number of 1/4-second intervals the call has to complete before the manager returns due to a time out.

`Completion vector`
Address of a routine called via a JSL when a SCSI call completes. There can be only one completion routine pending for each SCSI device. The routine performs the function, executes, and returns to the SCSI Manager through an RTL in full native mode.

`Command pointer`
Longword field; contains pointer to the SCSI command block for the current call.

`Buffer pointer`
Contains a pointer to a buffer that contains the data-chaining instructions that are executed by the SCSI Manager. Each instruction occupies 16 bytes, and all commands are contiguous. The last instruction must be a DCStop command. See "Data Chaining" earlier in this chapter.

`Byte count` Contains the number of bytes requested in the call and contains the number of bytes actually transferred on exit from the call.

`AutoSense buffer pointer`
Longword field; pointer to a buffer used to hold the error information returned when auto sensing is enabled. If the pointer is $00000000, then auto sensing is disabled. If the pointer is nonzero, auto sensing is enabled, and the SCSI Manager will automatically issue a RequestSense call to the target when an I/O call fails. The RequestSense buffer must be a minimum of one page (256 bytes) in size. If the RequestSense call fails, the error code is in the range from $FE80 to $FEFF. The low byte of the error code contains the value of the SCSI status byte returned by the target device when the call failed.

Reserved — Reserved and must contain $00 when the call is issued.

Errors — A register: Error code

Code	Error
$FE01	Invalid GS/OS SCSI device type
$FE02	SCSI check condition from SCSI device
$FE03	Device ID not found
$FE05	Parameter list
$FE08	Device already busy
$FE11	Call time out
$FE80–$FEFF	Request Sense failed

Sparing disk blocks

The Apple SCSI drivers use block sparing to reassign a block that has a lowered read reliability. All failed write attempts are spared, and reads are spared only when the data can be retrieved. The following description should clarify what actually takes place.

When a read call is issued to the device specifying a particular block, the device reads the block, checking it for wholeness (a checksum is examined to see if the data is intact). If the block is damaged, an error is returned, and the read is attempted again. If more than one retry is needed before the intact data is returned, then the block is assigned a new location on the device, and the data is written to the new block. If the block is so damaged that the data cannot be read, then an error is returned to the caller, and the block is not spared; it is possible that a subsequent read will be successful. Unsuccessful read attempts never result in a block's being spared. If the block is irreparably damaged, then the next write attempt to the block results in the block's being spared and a new one assigned.

Chapter 3 **The AppleDisk 3.5 Driver**

The AppleDisk 3.5 driver is a loaded driver that communicates directly with the hardware to support one or two Apple 3.5 drives. The Apple 3.5 drive is a block device that reads 3.5-inch disks in formats compatible with the ProDOS or Macintosh® file systems and connects directly to the Apple IIGS disk port or to a compatible expansion card in a slot (see the *Apple IIGS Hardware Reference* for more information on disk drive expansion).

The AppleDisk 3.5 driver operates independently of the system speed. The driver supports several formatting options: 400 KB or 800 KB disks and either 2:1 or 4:1 interleave.

This chapter describes the GS/OS AppleDisk 3.5 driver. It gives general information on the driver and includes descriptions of any driver-specific implementation of the standard GS/OS device calls.

Device calls to the AppleDisk 3.5 driver

Applications can access the AppleDisk 3.5 driver either through a file system translator (such as ProDOS) or by making device calls. Applications can make these device calls to the AppleDisk 3.5 driver:

- DInfo
- DStatus
- DControl
- DRead
- DWrite
- DRename

The rest of this chapter describes the differences between the way the AppleDisk 3.5 driver handles these device calls and the way a standard driver handles these calls. Any calls or subcalls not discussed here are handled exactly as documented in Chapter 1.

DStatus ($202D)

This call is used to obtain current status information from the device or the driver. The AppleDisk 3.5 driver supports this standard set of DStatus subcalls:

Status code	Subcall name
$0000	GetDeviceStatus
$0001	GetConfigParameters
$0002	GetWaitStatus
$0003	GetFormatOptions
$0004	GetPartitionMap

GetDeviceStatus

This subcall returns a general status followed by a longword specifying the number of blocks supported by the device.

The driver returns a disk-switched condition under appropriate circumstances. For a description of those circumstances, see "Driver_Status ($0005)" in Chapter 10.

GetConfigParameters

With this subcall, the AppleDisk 3.5 driver has no parameters in its configuration parameter list and returns with a status list length word of 0 and a transfer count of $0000 0002.

GetFormatOptions

This subcall returns a list of formatting options that may be selected using the DControl subcall SetFormatOptions prior to issuing a FormatDevice call to a block device. The AppleDisk 3.5 driver returns format options as follows:

`transferCount` $0000 0038 (56 bytes returned in list)

`statusList` *Option list header:*

$0003	Three options in list
$0003	All three options to be displayed
$0001	Recommended default = option 1
$0000	Current media formatting unknown

Option entry 1:

$0001	Option 1
$0000	No linked option
$0004	Apple format/size in kilobytes
$0000 0640	Block count = 1600
$0200	Block size = 512 bytes
$0002	Interleave factor = 2:1
$0320	Media size = 800 KB

Option entry 2:

$0002	Option 2
$0000	No linked option
$0004	Apple format/size in kilobytes
$00000640	Block count = 1600
$0200	Block size = 512 bytes
$0004	Interleave factor = 4:1
$0320	Media size = 800 KB

Option entry 3:

$0003	Option 3
$0000	No linked option
$0004	Apple format/size in kilobytes
$00000320	Block count = 800
$0200	Block size = 512 bytes
$0002	Interleave factor = 2:1
$0190	Media size = 400 KB

DControl ($202E)

This call is used to send control information to the device or the device driver. The AppleDisk 3.5 driver supports this standard set of DControl subcalls:

Control code	Subcall name
$0000	ResetDevice
$0001	FormatDevice
$0002	EjectMedium
$0003	SetConfigParameters
$0004	SetWaitStatus
$0005	SetFormatOptions
$0006	AssignPartitionOwner
$0007	ArmSignal
$0008	DisarmSignal

Only the following subcalls are nonstandard for the AppleDisk 3.5 driver.

ResetDevice

This control call is used to reset a particular device to its default settings. This call has no function with the AppleDisk 3.5 driver and returns with no error.

SetConfigParameters

This call has no function with the AppleDisk 3.5 driver and returns with no error.

SetWaitStatus

All block devices, including the Apple 3.5 drive, operate in wait mode only. Setting the AppleDisk 3.5 driver to wait mode results in no error. If a call is issued to set the AppleDisk 3.5 driver to no-wait mode, then error $22 (invalid parameter) is returned.

SetFormatOptions

This control call sets the current format option as specified in the format option list returned from the GetFormatOptions subcall of DStatus.

AssignPartitionOwner

This call has no function with the AppleDisk 3.5 driver and returns with no error.

ArmSignal

This call has no function with the AppleDisk 3.5 driver and returns with no error.

DisarmSignal

This call has no function with the AppleDisk 3.5 driver and returns with no error.

DRead ($202F)

This call returns the requested number of bytes from the disk starting at the block number specified. The request count must be an integral multiple of the block size. Valid block sizes for this driver are $0200 and $020C (512 and 524) bytes per block.

DWrite ($2030)

This call writes the requested number of bytes to the disk starting at the block number specified. The request count must be an integral multiple of the block size. Valid block sizes for this driver are $0200 and $020C (512 and 524) bytes per block.

Chapter 4 **The UniDisk 3.5 Driver**

The UniDisk 3.5 drive is a block device that reads 3.5-inch disks in formats compatible with the ProDOS or Macintosh file systems and connects directly to the Apple IIGS disk port. The UniDisk 3.5 drive supports up to four total UniDisk 3.5 drives on the disk port.

This chapter describes the GS/OS UniDisk 3.5 driver, a GS/OS loaded driver that controls the UniDisk 3.5 drive. It has general information on the driver and includes descriptions of any driver-specific implementation of the standard GS/OS device calls.

Device calls to the UniDisk 3.5 driver

Applications access a UniDisk 3.5 device either by making a file call that goes through a file system translator (FST) or by making a GS/OS device call. The UniDisk 3.5 driver supports these standard device calls from an application:

- Dinfo
- DStatus
- DControl
- DRead
- DWrite
- DRename

The rest of this chapter describes the differences between the way the UniDisk 3.5 driver handles these device calls and the way a standard driver handles these calls. Any calls or subcalls not discussed here are handled exactly as documented in Chapter 1.

DStatus ($202D)

The UniDisk 3.5 driver supports the standard set of status subcalls. Only the following are implemented in a nonstandard way.

GetDeviceStatus

This call returns a general status followed by a longword specifying the number of blocks supported by the device.

The driver returns a disk-switched condition under appropriate circumstances. For a description of those circumstances, see "Driver_Status ($0005)" in Chapter 10, "GS/OS Driver Call Reference."

GetConfigParameters

The UniDisk 3.5 has no parameters in its configuration parameter list. GetConfigParameters returns a transfer count of $0000 0002 and a status list length word of $0000.

GetWaitStatus

Block devices operate in wait mode only. For UniDisk 3.5 devices, GetWaitStatus always returns a transfer count of $0000 0002 and a wait status value of $0000 in the status list.

GetFormatOptions

This call returns a list of formatting options that may be selected using a SetFormatOptions DControl subcall prior to issuing a Format DControl subcall to a block device.

The UniDisk 3.5 driver returns a format options list as follows:

`transferCount` $0000 0018 (24 bytes returned in list)

`statusList` *Option list header:*

$0001	One option in list
$0001	One displayed option
$0001	Default = option 1
$0000	Current media formatted with option 1

Option entry 1:

$0001	Option 1
$0000	No linked option
$0004	Apple format/size in kilobytes
$00000640	Block count = 1600
$0200	Block size = 512 bytes
$0004	Interleave factor = 4:1
$0320	Media size = 800 KB

DControl ($202E)

The UniDisk 3.5 driver supports the standard set of control subcalls. Only the following calls are implemented in a nonstandard way.

ResetDevice

This subcall has no function with the UniDisk 3.5 driver and returns with no error.

SetConfigParameters

This subcall has no function with the UniDisk 3.5 driver and returns with no error.

SetWaitMode

All block devices operate in wait mode only. Setting the UniDisk 3.5 driver to wait mode results in no error. If a call is issued to set the UniDisk 3.5 driver to no-wait mode, then error $22 (invalid parameter) is returned.

SetFormatOptions

The UniDisk 3.5 driver supports the format options listed under the GetFormatOptions DStatus subcall. This option (and possible future options) can be specified in the parameter `formatOptionNum` for this subcall. However, the UniDisk 3.5 driver does not support overriding interleave factors, so `interleaveFactor` for this call must be $0000.

AssignPartitionOwner

This call has no function with the UniDisk 3.5 driver and returns with no error.

ArmSignal

This call has no function with the UniDisk 3.5 driver and returns with no error.

DisarmSignal

This call has no function with the UniDisk 3.5 driver and returns with no error.

DRead ($202F)

This call returns the requested number of bytes from the disk starting at the block number specified. The request count must be an integral multiple of the block size.
Valid block sizes for the UniDisk 3.5 driver are $0200 and $020C (512 and 524) bytes per block. Issuing this call with a block size other than $0200 or $020C will result in error $22 (invalid parameter).

DWrite ($2030)

This call writes the requested number of bytes to the disk starting at the block number specified. The request count must be an integral multiple of the block size.
Valid block sizes for this driver are $0200 and $020C (512 and 524) bytes per block. Issuing this call with a block size other than $0200 or $020C will result in error $22 (invalid parameter).

Chapter 5 **The AppleDisk 5.25 Driver**

Apple 5.25 drives, UniDisk drives, DuoDisk drives, and Disk II drives are block devices that read 5.25-inch floppy disks and are used widely with the Apple II family of computers. Disks formatted under the ProDOS, Pascal, or DOS 3.3 file systems can be read from these devices. The drives can plug directly into the Apple IIGS disk port, or they can connect to interface cards in slots. Under GS/OS, these drives are controlled by the AppleDisk 5.25 driver.

The AppleDisk 5.25 driver is a loaded driver that supports up to 14 Apple 5.25 drives and operates with either an interface card in a slot or the built-in disk port. The AppleDisk 5.25 driver functions independently of the system speed and does not have the resident slot limitation inherent in the Apple IIGS computer. This means that, although the Apple IIGS computer normally allows Apple 5.25 drives to operate at accelerated speed in slots 4 through 7 only, the AppleDisk 5.25 driver permits Apple 5.25 drives to operate at accelerated speed in all slots (1 through 7), with either one or two Apple 5.25 drives per slot.

This chapter describes how the the AppleDisk 5.25 driver works and what device calls it accepts. It also describes the physical and logical formats used by the AppleDisk 5.25 driver on 5.25-inch media.

- *Note:* For convenience, in this chapter the term *Apple 5.25 drive* is used to refer to all manifestations of the 5.25-inch drive—including Apple 5.25, UniDisk, DuoDisk, and Disk II.

Limitations of 5.25-inch disk drives

The Apple 5.25 drive provides no means for detection of disk-switched errors. The AppleDisk 5.25 driver provides a simulation of disk-switched detection by forcing any FST interfacing to the Apple 5.25 drive to identify the volume currently on line. This simulation of disk-switched errors is adequate to prevent writing to the wrong volume, but it is not adequate to validate the integrity of the cache. Therefore, the AppleDisk 5.25 driver does not implement caching. Also, the status subcall GetDeviceStatus never returns a disk-switched status.

Device calls to the AppleDisk 5.25 driver

Applications can access the Apple 5.25 drive either through an FST or by making device calls. Applications can make these standard device calls to the AppleDisk 5.25 driver:

- DInfo
- DStatus
- DControl
- DRead
- DWrite
- DRename

The rest of this chapter describes how the AppleDisk 5.25 driver handles any of the above device calls differently than the standard ways documented in Chapter 1. Any calls or subcalls not discussed here are handled exactly as documented in Chapter 1.

DStatus ($202D)

This call is used to obtain current status information from the device or the driver. The AppleDisk 5.25 driver supports this standard set of status subcalls:

Status code	Subcall name
$0000	GetDeviceStatus
$0001	GetConfigParameters
$0002	GetWaitStatus
$0003	GetFormatOptions

The following descriptions show how the the AppleDisk 5.25 driver handles various DStatus subcalls differently than the standard descriptions given in Chapter 1 of this reference.

GetDeviceStatus

This call returns a general status word followed by a longword specifying the number of blocks supported by the device. Because there is no way to validate media insertion on an Apple 5.25 drive, bit 4 ("disk in drive") of the device status word is always set to 1.

GetConfigParameters

The AppleDisk 5.25 driver has no parameters in its configuration parameter list. It returns a length word of 0 in the status list and transfer count of $0000 0002 in the parameter block.

GetFormatOptions

This call returns a list of formatting options that you can select using the DControl subcall SetFormatOptions prior to issuing a format call to a block device. The AppleDisk 5.25 driver returns format options as follows:

`transferCount` $0000 0028 (40 bytes returned in list)

`statusList` *Option list header:*

$0002	Two options in list
$0001	Only one to be displayed
$0001	Recommended default = option 1
$0000	Formatting option of current media unknown

Option entry 1:

$0001	Option 1
$0002	This option linked to option 2
$0004	Apple format/size in kilobytes
$0000 0118	Block count = 280
$0200	Block size = 512 bytes
$0000	Interleave factor = n/a (fixed physical interleave)
$008F	Media size = 140 KB

Option entry 2:

$0002	Option 2
$0000	No linked options
$0004	Apple format/size in kilobytes
$0000 0230	Block count = 560
$0100	Block size = 256 bytes
$0000	Interleave factor = n/a (fixed physical interleave)
$008F	Media size = 140 KB

DControl ($202E)

This call is used to send control information to the device or the device driver. The AppleDisk 5.25 driver supports this standard set of DControl subcalls:

Control code	Subcall name
$0000	ResetDevice
$0001	FormatDevice
$0002	EjectMedium
$0003	SetConfigParameters
$0004	SetWaitStatus
$0005	SetFormatOptions
$0006	AssignPartitionOwner
$0007	ArmSignal
$0008	DisarmSignal

The rest of this chapter describes the differences between the way the AppleDisk 5.25 driver handles DControl subcalls and the way a standard driver handles these subcalls. See Chapter 1 for complete documentation of DControl.

ResetDevice

This call has no function for the AppleDisk 5.25 driver and returns with no error.

FormatDevice

This subcall is used to format a disk. The AppleDisk 5.25 driver ignores the control list.

EjectMedium

The Apple 5.25 drive does not have any mechanism for ejecting disks. This call has no function with the AppleDisk 5.25 driver and returns with no error.

SetConfigParameters

The AppleDisk 5.25 driver has no configuration parameters. This call has no function and returns with no error.

SetWaitStatus

All block device drivers, including the AppleDisk 5.25 driver, operate in wait mode only. Setting the AppleDisk 5.25 driver to wait mode results in no error; attempting to set the driver to no-wait mode results in error $22 (invalid parameter).

SetFormatOptions

Because only a single fixed physical interleave is supported, this call works with either format option but has no effect on the actual formatting of the media. This call returns with no error.

AssignPartitionOwner

This call has no function with the AppleDisk 5.25 driver and returns with no error.

ArmSignal

This call has no function with the AppleDisk 5.25 driver and returns with no error.

DisarmSignal

This call has no function with the AppleDisk 5.25 driver and returns with no error.

DRead ($202F)

This call returns the requested number of bytes from the disk starting at the block number specified. The request count must be an integral multiple of the block size. The AppleDisk 5.25 driver supports block sizes of 256 bytes (for DOS 3.3) and 512 bytes (for ProDOS and Pascal) and block counts of 560 and 280 blocks, respectively. Logical interleave on the disk varies with the block size.

△ **Important** In order to force disk-switched detection on an Apple 5.25 drive, the AppleDisk 5.25 driver returns a disk-switched error on any read or write request if there has not been a media access in the previous one second. If your application is accessing the AppleDisk 5.25 driver directly, the application has to handle the disk-switched error. The normal procedure is to retry once and only once. △

DWrite ($2030)

This call writes the requested number of bytes to the disk starting at the block number specified. The request count must be an integral multiple of the block size. The AppleDisk 5.25 driver supports block sizes of 256 bytes (for DOS 3.3) and 512 bytes (for ProDOS and Pascal) and block counts of 560 and 280 blocks, respectively. Logical interleaving on the disk varies with the block size.

△ **Important** In order to force disk-switched detection on an Apple 5.25 drive, the AppleDisk 5.25 driver returns a disk-switched error on any read or write request if there has not been a media access in the previous one second. If your application is accessing the AppleDisk 5.25 driver directly, the application has to handle the disk-switched error. The normal procedure is to retry once and only once. △

AppleDisk 5.25 driver formatting

The AppleDisk 5.25 driver supports only 35-track, 16-sector formatting. Media are formatted with a physical 1:1 interleave. Logical interleave is achieved by using one of two interleave translation tables. DOS 3.3 operates on 256-byte sectors; ProDOS and Pascal operate on 512-byte blocks consisting of two contiguous "logical sectors." Both ProDOS and Pascal use a common logical sector interleave of 2:1, while DOS 3.3 uses a logical sector interleave of 4:1.

Logical-to-physical-sector translations are shown in the interleave translation tables of Figure 5-1. The input block size to a media-access call controls which translation table is used.

■ **Figure 5-1** Apple 5.25 drive interleave configurations

ProDos or Pascal disks:

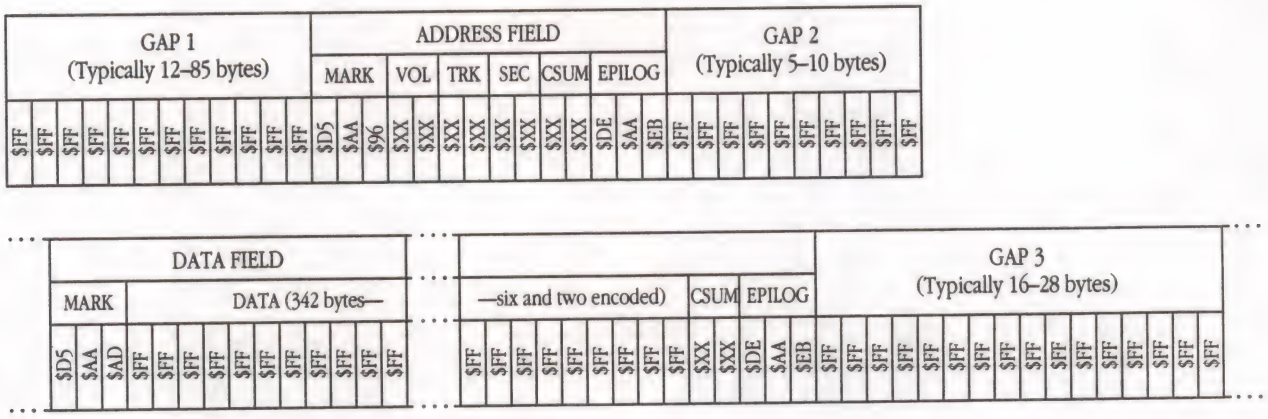
Logical sector address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Physical sector address	0	2	4	6	8	A	C	E	1	3	5	7	9	B	D	F

DOS 3.3 disks:

Logical sector address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Physical sector address	0	D	B	9	7	5	3	1	E	C	A	B	6	4	2	F

As Figure 5-2 shows, each sector consists of a self-synchronization gap, followed by the sector address field, followed by another self-synchronization gap, followed by the data field, and ending with a final gap. The sector address field contains the volume number, track number, sector number, and checksum for the sector. The data field contains 342 bytes of data and a checksum. Both the address field and the data field have beginning (mark) and ending (epilog) markers.

■ **Figure 5-2** Apple 5.25 drive sector format



Chapter 6 **The AppleTalk Drivers**

The AppleTalk drivers are loaded drivers that provide network protocol services, allowing Apple II programs to interact with devices connected to an AppleTalk network. Utilizing one of the two RS-232 serial ports, the AppleTalk drivers supply network printing and server access. You need the information in this chapter only if your application uses AppleTalk.

The AppleTalk driver is actually three drivers in one: the .AppleTalk driver, the .AFPn driver, and the .RPM driver. The three components are stand-alone drivers in themselves but are loosely connected to each other by being in the same driver file. These three drivers are described in this chapter, as are the calls to the drivers.

This chapter also describes the SCC Manager. This supervisory driver manages the AppleTalk port and arbitrates between all AppleTalk device drivers and AppleTalk network peripherals.

For complete information on AppleShare® and the Apple IIGS computer, you should also have the *AppleShare Programmer's Guide for the Apple II Family.*

The Remote Print Manager driver (.RPM)

The Remote Print Manager driver (.RPM) is a loaded character driver that communicates with the AppleTalk Remote Print Manager. The purpose of this driver is to provide a means of communication between a GS/OS application program and the older Pascal 1.1 firmware entry points for the AppleTalk slot. (For more information on the Pascal interface protocol, see the *Apple IIGS Firmware Reference*.)

Printing to the Remote Print Manager under system software version 4.0 was accomplished by means of a generated driver that was created by GS/OS at boot time. Generated drivers are created automatically for any slot that has valid firmware but no loaded driver. Since the current version of GS/OS (system software version 5.0 and later) has a number of loaded drivers for the AppleTalk slot, GS/OS no longer generates a driver for the Remote Print Manager. A new loaded .RPM driver has been written for system software 5.0.

Instead of making calls directly to the AppleTalk slot firmware, application programs should print to a network printer by using standard GS/OS Write calls to the .RPM driver. The .RPM driver uses special GS/OS generated device-management code to send the data to the AppleTalk slot. This is a much cleaner approach than writing directly to the slot and is the only approach supported by Apple. For this reason, GS/OS application developers should make use of it. Although a GS/OS application can print to a network printer using the .RPM driver, you should limit use of the .RPM driver to text-only output; use the Print Manager for all of your other printing needs.

The .RPM driver is part of the main .ATALK driver located in the *:SYSTEM:DRIVERS subdirectory.

About calls to the .RPM driver

The .RPM driver supports the standard driver calls. These calls are

- DInfo
- DStatus
- DControl
- DRead
- DWrite
- DRename

The remainder of this section describes the way the .RPM driver handles these calls differently than standard driver calls. For details of the standard calls and the requirements and capabilities of each, see Chapter 1, "GS/OS Device Call Reference."

DStatus ($202D)

Description The DStatus call is used to obtain status information from the device or the driver. The DStatus call supports several subcalls, which are described in this section. These are

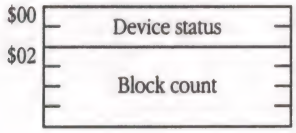
Code	Subcall name
$0000	GetDeviceStatus
$0001	GetConfigParameters
$0003	GetFormatOptions
$0004	GetPartitionMap
$8001	GetRPMParameters

GetDeviceStatus (DStatus subcall)

Description This status subcall returns the general device status word followed by a longword. This subcall is handled by the driver's generated device-management code.

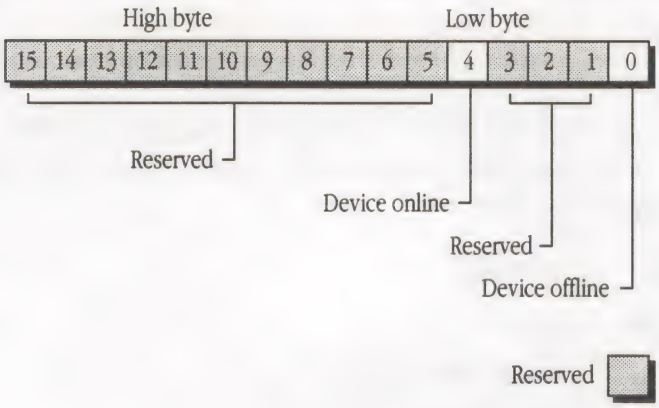
Return parameters The return data format is shown in Figure 6-1, and the individual fields are defined following the figure.

■ **Figure 6-1** GetDeviceStatus subcall return data



`Device status` Contains the device status word. Bit 0 is set if the device is open. Bit 4 is set if the device is on line.

■ **Figure 6-2** Device status word



`Block count` This field is not used by character devices and will always be equal to $00000000.

GetConfigParameters (DStatus subcall)

Description This status subcall is not valid for character drivers and will always return with no error and with the transfer count set to 0.

GetFormatOptions (DStatus subcall)

Description This status subcall is not valid for character drivers and will always return with no error and with the transfer count set to 0.

GetPartitionMap (DStatus subcall)

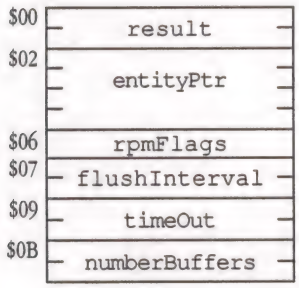
Description This status subcall is not valid for character drivers and will always return with no error and with the transfer count set to 0.

GetRPMParameters (device-specific subcall)

`statusCode` = $8001

Description This DStatus subcall returns the current parameters used by the .RPM driver.

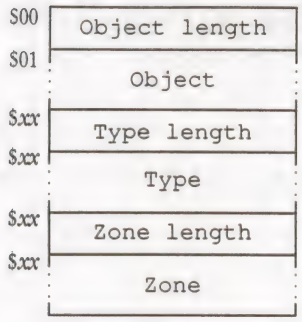
■ **Figure 6-3** GetRPMParameters subcall format



`result` The result from the PMSetPrinter AppleTalk call will be placed in this field. (See the *AppleShare Programmer's Guide for the Apple II Family* for more information on the PMSetPrinter call.)

`entity Ptr` Contains a longword pointer to a standard AppleTalk entity name. The name must be supplied before the call is made, and the buffer containing the name must be 100 ($64) bytes long. An entity name is a character string consisting of three Pascal strings: object, type, and zone concatenated together. The format of the entity name is shown in Figure 6-4 and is described following the figure.

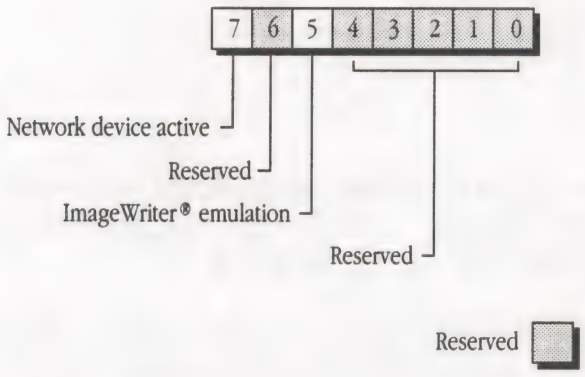
■ **Figure 6-4** GetRPMParameters entity name format



The first two fields of the entity name can be up to 32 characters plus 1 length byte. The third field can be up to 33 characters plus 1 length byte. This means that the supplied buffer must be 100 bytes in length.

`rpm Flags` Contains the flags byte for the .RPM driver, shown in Figure 6-5. The bits in the flags byte are defined in Table 6-1.

■ **Figure 6-5** .rpm Flags byte



■ **Table 6-1** .rpm Flags byte definitions

Bit	Definition
7	This bit is set when the specified printer is on the network. This bit should always be set on the Apple IIGS computer. Use the SetRPMParameters call to set this bit.
6	Reserved.
5	If this bit is set, the printer is a LaserWriter printer with the ImageWriter emulator installed. At the start of a print job, the printer is sent a command causing it to act like an ImageWriter printer. (See the *AppleShare Programmer's Guide to the Apple II Family* for more information on using the LaserWriter in emulation mode.)
▲ **Warning**	This call does not check for the presence of the ImageWriter emulator. Be sure it has been downloaded to the LaserWriter before printing using the .RPM driver. ▲
4–0	Reserved.
`flushInterval`	Word value that specifies the number of 1/4-second intervals the .RPM driver waits for new characters before flushing a less-than-full output buffer. The default value is one 1/4-second interval.
`timeOut`	Word value that specifies the number of 1/4-second intervals the .RPM driver waits for new characters to be sent from the application program before timing out and ending the print job. The default value is 30 seconds.
`numberBuffers`	Word value that specifies the number of 512-byte buffers the .RPM driver uses for storing outgoing information. The more buffers allocated, the faster the printing will be. As many buffers can be allocated as there is available memory space to support. The default number is 20 buffers (10 KB).

DControl ($202E)

Description The DControl call allows the .RPM driver to pass data and control information to devices on the network. The DControl call supports two control subcalls, which are described in this section. These are

Code	Subcall name
$0000	ResetDevice
$8001	SetRPMParameters

ResetDevice (DControl subcall)

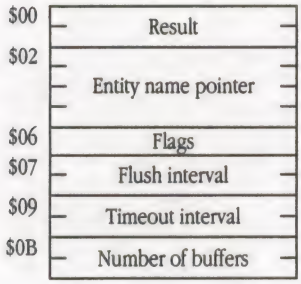
Description This control subcall resets the specified network device to its default settings. The only setting that is affected by this subcall is the wait/no-wait read mode. After a reset, the read mode will be set to wait. This subcall will always return with no error and with the transfer count set to 0.

SetRPMParameters (device-specific subcall)

`controlCode` = $8001

Description This subcall sets the parameters to be used by the .RPM driver. The individual parameters are described following Figure 6-6.

■ **Figure 6-6** SetRPMParameters subcall format



`result`	The result from the AppleTalk call PMSetPrinter is placed here. For information on the PMSetPrinter command, see the *AppleShare Programmer's Guide for the Apple II Family*.
`entity Ptr`	Contains a longword pointer to a standard AppleTalk entity name. See the GetRPMParameters status subcall for the format of `Entity name pointer`.
`rpm Flags`	Contains the flags byte for the .RPM driver. See the GetRPMParameters DStatus subcall for the format of the flags byte.
`flushInterval`	Word value that specifies the number of 1/4-second intervals the .RPM driver waits for new characters before flushing a less-than-full output buffer.
`timeOut`	Word value that specifies the number of 1/4-second intervals the .RPM driver waits for new characters to be sent from the application program before timing out and ending the print job.
`numberBuffers`	Word value that specifies the number of 512-byte buffers the .RPM driver uses for storing outgoing information. The more buffers allocated, the faster the printing will be. As many buffers can be allocated as there is available memory space to support. The default number is 20 buffers (10 KB).

DRead ($202F)

Description The DRead call for this driver is not valid because the printer is a write-only device. This call returns error $4E (invalid access).

DWrite ($2030)

Description The DWrite call is used to send print data to the driver. Data sent to the printer is buffered internally; not every DWrite call results in data's being printed. To send a less-than-full buffer to the printer, use the Flush call ($2015). Otherwise, the DWrite call for the .RPM driver is the same as the standard DWrite call for all drivers. Refer to Chapter 1, "GS/OS Device Call Reference," for specifics on the standard DWrite call.

The .AppleTalk driver

In the past, the only way for an application or driver to determine general AppleTalk parameters was to look at specific memory locations. With the .AppleTalk driver, an application or FST can determine these general AppleTalk values by making status calls to the driver.

The .AppleTalk driver is a loaded character driver that provides a common interface for determining general AppleTalk variables. The driver provides application programs with the following information:

- AppleTalk presence: An application can make the assumption that if this driver is present in the system, the AppleTalk protocols are also present.
- AppleTalk port: The driver can return which SCC channel the link access protocol (LAP) is using as the AppleTalk port.

The .AppleTalk driver is part of the main .ATALK driver in the *:SYSTEM:DRIVERS subdirectory. Also included in this driver file are the .RPM and .AFPn drivers. Refer to the sections in this chapter concerning the .AFPn and .RPM drivers for more information.

Protocol layer interaction

It is important that you, the developer, know the difference between AppleTalk and AppleShare. AppleTalk is the entire network system, and AppleShare is simply AppleTalk file service.

This driver neither uses nor provides direct communication with the AppleTalk protocol layers. An application must make AppleTalk protocol calls through the AppleTalk dispatch vector at $E11014. This is because AppleTalk uses an asynchronous calling scheme, and GS/OS is designed around a synchronous call strategy. For more information on AppleTalk protocol calls, see the *AppleShare Programmer's Guide for the Apple II Family*.

About calls to the .AppleTalk driver

The .AppleTalk driver supports the standard driver calls. These calls are

- DInfo
- DStatus
- DControl
- DRead
- DWrite

The remainder of this section describes the way the .AppleTalk driver handles these calls differently than standard driver calls. For details of the standard calls and the requirements and capabilities of each, see Chapter 1, "GS/OS Device Call Reference."

DStatus ($202D)

This call provides a means to interrogate the .AppleTalk driver and obtain status information. The .AppleTalk driver supports all the DStatus subcalls plus the GetPort device-specific subcall.

GetWaitStatus

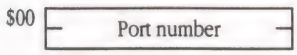
Description This status subcall always returns with $00, since the driver always operates in wait mode. Always set the request count to $02.

GetPort (device-specific subcall)

`statusCode` = $8001

Description This device-specific status subcall returns the port number that AppleTalk is currently using. Figure 6-7 illustrates the GetPort return data.

■ **Figure 6-7** GetPort return data



`Port number` The port number that AppleTalk is currently using:
$0001 = port 1 (printer port)
$0002 = port 2 (modem port)

DControl ($202E)

Description The DControl call allows the .AppleTalk driver to pass data and control information to devices on the network. The .AppleTalk driver supports most of the standard DControl subcalls. The exceptions are the device ArmSignal and DisarmSignal calls; these are not supported and always return error $21 (invalid control or status code).

DRead ($202F)

Description This call is never executed and instead returns error $4E (invalid access).

DWrite ($2030)

Description This call is never executed and instead returns error $4E (invalid access).

The AppleTalk Filing Protocol (.AFPn) driver

The AppleTalk Filing Protocol (.AFPn) driver is a loaded block driver that communicates with the AppleTalk ProDOS Filing Interface (the ProDOS file service) and also provides FSTs with a means of logically connecting AFP volumes to GS/OS.

AppleShare file service for the Apple IIGS computer is achieved through the .AFPn driver. The driver is actually 14 drivers in one; it has 14 device information blocks (DIBs), for a maximum of 14 different AppleShare volumes on line at one time, but it uses the same core routines for all 14 DIBs. The names of the devices are .AFP1 through .AFP14. (For information on DIBs and their structure, see Chapter 8, "GS/OS Device Driver Design.")

This driver provides these services:

- Maintains and updates DIBs for all AppleShare volumes, reporting the loss of a connection caused by either logging off or losing the session with a server. The driver treats these occurrences as disk-switch events. The only way for the driver to again hold valid data is if a new volume is mounted and assigned to this driver.
- Notifies users of a lost connection or server shutdown. The driver displays an appropriate dialog.
- Provides session information. This includes the session reference number, volume ID, volume name, server name, and zone name associated with a particular .AFPn driver.
- Handles volume eject. This is done by unmounting volumes and logging off the server if the last volume on the server has just been unmounted.

Interaction with ProDOS Filing Interface

The ProDOS Filing Interface (PFI) resides under GS/OS and provides AppleTalk file service for ProDOS 8. Instead of duplicating much of the code that is already in the PFI, the .AFPn driver relies heavily on the PFI to determine information about AFP volumes. The .AFPn driver relies on the PFI for the following information:

- new volumes that are mounted or unmounted
- volumes already mounted during an OS switch from ProDOS 8 to GS/OS, or vice versa
- names of AFP volumes and their session reference numbers and volume ID numbers
- attention messages coming in from a server

- *Note:* All DIBs are rebuilt whenever a switch from ProDOS 8 to GS/OS occurs. The DIBs are rebuilt to reflect the status of the volumes currently mounted, including configuration data and eject status.

The .AFPn drivers are part of the main .ATALK driver and are located in the *:SYSTEM:DRIVERS subdirectory of the GS/OS system disk 5.0. Also included in this driver file are the .RPM and .AppleTalk drivers. Refer to the .RPM and .AppleTalk driver sections in this chapter for more information on those drivers.

About calls to the .AFPn driver

The .AFP driver supports the standard list of driver calls. These calls are

- DInfo
- DStatus
- DControl
- DRead
- DWrite
- DRename

The remainder of this section describes the way the .AFP driver handles these calls differently than standard driver calls. For details of the standard calls and the requirements and capabilities of each, see Chapter 1, "GS/OS Device Call Reference."

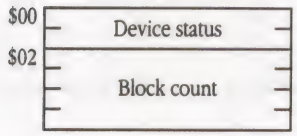
DStatus ($202D)

Description This call is used to request status information from the driver. The .AFP driver supports all the standard DStatus subcalls plus one additional device-specific subcall, GetEject. These subcalls are described in this section.

GetDeviceStatus

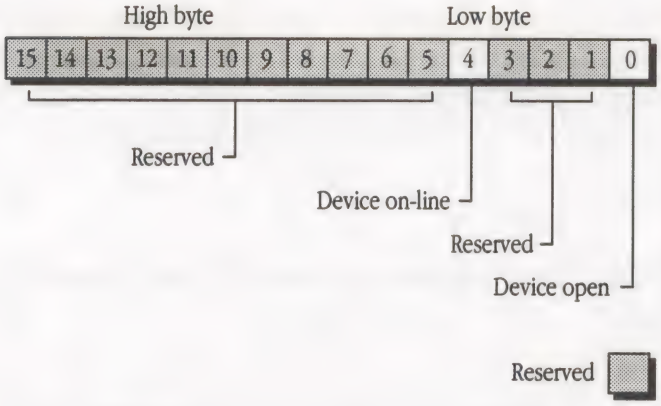
Description This status subcall returns the general device status word followed by a longword specifying the number of blocks supported by the device. The individual fields are defined below.

■ **Figure 6-8** GetDeviceStatus subcall return data



`Device status` Contains the device status word. Bit 0 is set if the disk has been switched. Bit 4 is set if the volume is on line. Bit 15 is set if the block count is unknown.

■ **Figure 6-9** GetDeviceStatus device status word

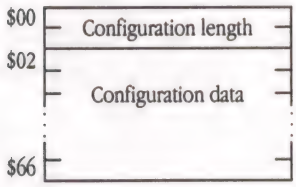


`Block count` Contains the size of the device in blocks. This value will always be equal to $007FFFFF for an AppleShare volume.

GetConfigParameters

Description This status subcall returns a length word as the first element in the status list that indicates the length of the configuration parameter list. The fields are defined following Figure 6-10.

■ **Figure 6-10** Device configuration status list



`Configuration length`
Word that contains the length of the `Configuration data` field in bytes.

`Configuration data`
Actual configuration data; will always be 101 ($65) bytes. The standard DIB settings returned in this field are the same ones specified in the SetRPMParameters subcall of the DControl call to the .RPM driver. The configuration data fields are described in Table 6-2.

■ **Table 6-2** Configuration data fields

Field	Length	Description
Device number	Word	Logical number of the device within the driver
Session number	Byte	Unique number assigned to each connection with a file server
Reserved	Byte	Not used under GS/OS; used for slot number and drive number under ProDOS 8
Volume name	28 bytes	Name associated with each volume

[continued]

■ **Table 6-2** Configuration data fields [continued]

Field	Length	Description
Volume ID number	Word	Number assigned to the volume on the server
Status word	Word	Device status word; see the section earlier in this chapter on the .AFPn driver DStatus subcall GetDeviceStatus for details
Server name	32 bytes	Unique name of the server (seen from the Control Panel)
Zone name	33 bytes	Name of the zone in which the server is located

◆ *Note:* The server name and zone name are returned as empty strings if the FILogin call is used rather than the FILogin2 call when logging onto the server.

GetFormatOptions

Description This status subcall is not valid for the .AFPn driver because the device cannot be formatted. This call always returns with no error and with the transfer count equal to 0.

GetPartitionMap

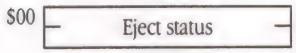
Description This status subcall is not valid for the .AFPn driver because the device cannot be formatted. This call always returns with no error and with the transfer count equal to 0.

GetEjectStatus (device-specific subcall)

`statusCode` = $8002

Description This device-specific status subcall returns the current eject status. The parameter list for this subcall is shown in Figure 6-11.

■ **Figure 6-11** GetEjectStatus parameter list



`Eject status` This field contains the eject status of the device—that is, whether or not the media may be ejected. The default is $0000 (volume may be ejected). This value is not reset to 0 when a Reset control call is issued.
$0000 = volume may be ejected
$8000 = volume may not be ejected

DControl ($202E)

Description This call is used to send data and control information to the device on the network. The .AFPn driver supports all the standard control calls plus two additional device-specific subcalls: DisplayMessages and SetEjectStatus. These subcalls are described in this section.

These are the only valid device-specific subcalls:

Code	Subcall name
$8001	DisplayMessages
$8002	SetEjectStatus

ResetDevice (DControl subcall)

Description This subcall is supported for compatibility only and actually does nothing. This subcall always returns with no error and with the transfer count equal to 0.

Format Device (DControl subcall)

Description This control subcall is not valid for the .AFPn driver because the device cannot be formatted. This call always returns with no error and with the transfer count equal to 0.

EjectMedium (DControl subcall)

Description This control subcall unmounts the volume and logs the user off the file server if the volume was the last volume connected to that server. If an error occurs while unmounting or logging off, error $88 (AppleTalk error) is returned. If the volume has been marked as unable to be ejected by the SetEjectStatus subcall, then no error is returned, and a disk switch occurs to simulate the volume being ejected and reinserted.

SetConfigParameters (DControl subcall)

Description This control subcall would normally set the current parameter list of the specified device to the given parameter list. Because the beginning of the AFP DIB configuration data contains critical information, the first 101 ($65) bytes of configuration data are ignored and are never set by this call. The first word of the status buffer must be a byte count that indicates the length of the configuration data. The configuration parameters must be contiguous to the byte count.

SetWaitStatus (DControl subcall)

Description This control subcall sets the current wait mode. A read from the .AFPn driver will always result in error $88 (AppleTalk error), so this call does nothing and always returns with no error and with the transfer count equal to 0.

SetFormatOptions (DControl subcall)

Description This control subcall is not valid for the .AFPn driver because the device cannot be formatted. This call always returns with no error and with the transfer count equal to 0.

AssignPartitionOwner (DControl subcall)

Description This control subcall is not valid for the .AFPn driver because the device cannot be formatted. This call always returns with no error and with the transfer count equal to 0.

ArmSignal (DControl subcall)

Description The .AFPn driver does not support any signaling, so this subcall always returns with error $21 (invalid control or status code) and with the transfer count equal to 0.

DisarmSignal (DControl subcall)

Description The .AFPn driver does not support any signaling, so this subcall always returns with error $21 (invalid control or status code) and with the transfer count equal to 0.

SetPartitionMap (DControl subcall)

Description This control subcall is not valid for the .AFPn driver because the device cannot be formatted. This call always returns with no error and with the transfer count equal to 0.

DisplayMessages (DControl subcall)

`controlCode` = $8001

Description This control subcall displays all pending server messages. This call is for internal use only; there is no need for a program to make this call.

SetEjectStatus (DControl subcall)

`controlCode` = $8002

Description This control subcall sets the current eject status. The status list is shown in Figure 6-12 and is defined following the figure.

■ **Figure 6-12** SetEjectStatus status list

◆ *Note:* This call is reserved for use by the AppleShare FST only.

Input parameters

`Eject status` Determines the eject status of the device—that is, whether or not the media may be ejected. The default is $0000 (volume may be ejected). This value is not reset to 0 when a Reset control call is made.
$0000 = volume may be ejected
$8000 = volume may not be ejected

DRead ($202F)

Description In order to maintain compatibility with system software version 3.2, this call normally returns error $88 (AppleTalk error). The only other errors returned by this call are error $2F (disk off line), error $2E (disk switched), and error $2C (bad request count).

DWrite ($2030)

Description This call has no effect on the driver and always returns error $2B (write protected).

The SCC Manager

The SCC Manager is a loaded supervisor driver that oversees and manages the AppleTalk protocols, GS/OS AppleTalk drivers, and any other GS/OS driver that requires the use of the serial communications controller (SCC). The driver provides the following services:

- both loads and initializes all the available AppleTalk protocols found in the *:SYSTEM:DRIVERS directory
- assigns a unique AppleTalk driver unit and slot number by determining in which slot AppleTalk resides
- provides arbitration of the two SCC channels for such services as AppleTalk, MIDI, modem, and printer drivers to share the SCC effectively

Calls to the SCC Manager

The SCC Manager is a supervisory driver that controls all communications through the serial ports by managing arbitration between serial device drivers and serial peripherals (including network devices). All serial driver calls are issued to the SCC Manager. If you plan on writing your own serial driver for use with the SCC, you must understand these SCC Manager calls described in this section:

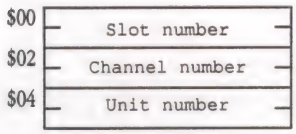
- AppleTalkClient
- GetChannelStatus
- SetChannelStatus

AppleTalkClient

`controlCode` = $0002

Description This call returns to the calling driver the AppleTalk slot number, channel number, and a unique unit number that the calling driver should use in its DIB.

Parameters



Offset	Size	Description
	Longword	AppleTalk slot number
	Longword	SCC channel number
	Longword	Unique AppleTalk unit number

`Slot number` Contains the AppleTalk slot number.

`Channel number`
Contains the SCC channel number.

`Unit number` Contains the AppleTalk unit number.

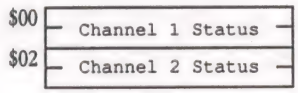
GS/OS drivers are expected to ask for a new unit number every time they start up. This means that when switching from ProDOS 8 back to GS/OS, the driver must reissue the AppleTalkClient call in order to receive a new unique unit number.

GetChannelStatus

`controlCode` = $0003

Description This call returns the current status of the SCC channels.

Parameters



Offset	Size	Description
	Longword	Channel 1 status
	Longword	Channel 2 status

`Channel 1 Status`

Contains the current status of channel 1:
$0000 = channel available
$0001 = channel being used
$0002 = channel being shared

`Channel 2 Status`

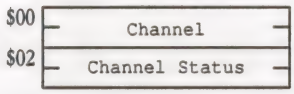
Contains the current status of channel 2:
$0000 = channel available
$0001 = channel being used
$0002 = channel being shared

SetChannelStatus

`controlCode` = $0004

Description This call sets the status of the specified SCC channels.

Parameters



Offset	Size	Description
$00	Longword	Channel
$02	Longword	Channel status

`Channel`

Indicates which channel of the SCC is being altered:
$0000 = channel 0
$0001 = channel 1

`Channel Status`

Indicates the desired status you wish to implement:
$0000 = channel available
$0001 = channel being used
$0002 = channel can be shared

The channel status is maintained only between the GS/OS startup and shutdown calls. After switching back from ProDOS 8 to GS/OS, the channel status for each channel, with the exception of the AppleTalk channel, is reset to 0 (channel available).

Errors

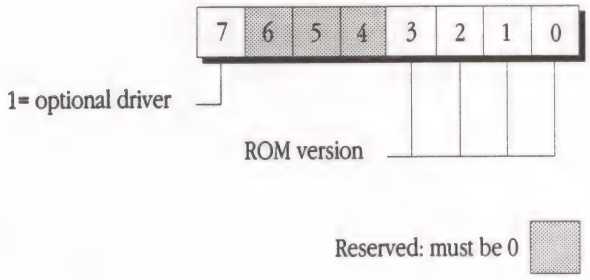
drvr_bad_parm	$22	Invalid parameter
drvr_no_resrc	$26	Resource not available

AppleTalk drivers

The SCC Manager loads and initializes all AppleTalk drivers found in the DRIVER directory on the boot disk. An AppleTalk driver is identified by a file type ($BB = driver file) and the high-order byte of the driver auxiliary type ($02 = AppleTalk driver). The low-order byte of the auxiliary type contains information about under what startup conditions the driver should be loaded. The auxiliary type is defined as follows:

High byte: The high byte must be $02.

Low byte: The low byte has the following format:



An AppleTalk driver will be loaded only if the current system ROM version is equal to or less than the ROM version field.

All AppleTalk drivers are loaded when the Apple IIGS computer is booted from a local disk drive. If the computer is booted over the network, only the drivers with the Optional Driver bit set will be loaded. Files are loaded in alphabetical (ASCII) order, not necessarily in the order they appear in the directory.

The SCC Manager uses the GS/OS loader to load the .AppleTalk driver. The manager then initializes the AppleTalk driver by jumping to the first byte of the driver. The X register contains the Apple IIGS ROM version, and the Y register contains the AppleTalk slot number. On exit from the initialization routine, the AppleTalk driver must set the accumulator to 0 and clear the carry bit.

Examples

The following driver code example allows a standard GS/OS driver to find the AppleTalk slot number and acquire a unique unit number for the slot.

```
slot_num            equ     $2E             ; Offset into DIB to slot #
unit_num            equ     $30             ; Offset into DIB to unit #

sup_drvr_disp  equ     $01FCA4         ; GSOS system service vector
drvr_dib_ptr   equ     $20             ; GSOS driver zero page
sup_parm_ptr   equ     $78             ; GSOS driver zero page


               phk
               plb

               rep     #$30
               longa   on
               longi   on

               lda     #$0000 ; No driver number (None)
               tax            ; Call number 0 (GetSupNum)
               ldy     #$0001 ; Supervisory ID 1 (SCC)
               jsl     sup_drvr_disp
               bcs     Error

               stx     sup_num         ; Store AppleTalk supervisor number
                       ; for later use

               lda     #ParmList       ; Store parameter block pointer
               sta     <sup_parm_ptr ; on zero page for supervisor's
               lda     #^ParmList      ; use
               sta     <sup_parm_ptr+2
               lda     sup_num         ; Supervisor driver number
               ldx     #$0002 ; Call number 2 (AppleTalkClient)
               ldy     #$0001 ; Supervisor ID 1 (SCC)
               jsl     sup_drvr_disp
               bcs     Error

               ldy     #slot_num       ; Slot number offset
               lda     Slot
               sta     [drvr_dib_ptr],y        ; Store slot # in our DIB
```

```
                ldy     #unit_num       ; Unit number offset
                lda     Unit
                sta     [drvr_dib_ptr],y        ; Store unit # in our DIB

                clc
Error           rts             ; C clear = no error
                                ; C set = no AppleTalk

sup_num         ds.w    1       ; Supervisor driver number goes here

ParmList                        ; Parameter list for AppleTalkClient
Slot            dc.w    0
Channel         dc.w    0
Unit            dc.w    0
```

Chapter 7 **GS/OS Generated Drivers**

At system startup, two kinds of device drivers are installed into the GS/OS device driver list: loaded drivers and generated drivers. GS/OS constructs a generated driver for each slot that does not have an associated loaded driver, so that all the device drivers supported by GS/OS can use the same standard interface.

With generated drivers, GS/OS allows applications to make standard GS/OS calls to access firmware-based device drivers (both built in and on peripheral cards) written for the Apple II family of computers.

This chapter describes the BASIC, Pascal 1.1, ProDOS, and SmartPort generated drivers and lists the device calls they support.

- *Note:* If you are writing a firmware driver for an Apple II peripheral card, read Appendix A, "Generated Drivers and Firmware Drivers." It explains how GS/OS recognizes and dispatches to firmware-based I/O drivers.

About generating drivers

At startup, GS/OS constructs a **device list,** a list of pointers to information about each installed device driver. GS/OS builds the list in this order:

1. It first installs all loaded drivers from the subdirectory *:SYSTEM:DRIVERS on the system disk.
2. For each slot *n* that does not have an associated loaded driver, GS/OS looks for a firmware I/O driver. It examines the appropriate firmware ID bytes in the $C*n*00 page of bank 0, and generates a GS/OS driver for any firmware driver it finds that uses BASIC, Pascal 1.1, ProDOS, SmartPort, or extended SmartPort protocols.

Generated drivers have two primary advantages over firmware drivers. These advantages are

- Peripheral card firmware is written in 6502 assembly-language code and is executable only in emulation mode on the Apple IIGS computer. However, generated drivers allow applications to access these drivers while running in native mode.
- Most firmware drivers cannot directly access memory banks other than bank $00; for these drivers, GS/OS double-buffers the data through bank $00 so that applications can access the drivers from anywhere in memory.

Each generated driver has an associated **device information block (DIB),** just like a loaded driver. The DIB contains device-specific information that can be used by the driver and by other parts of GS/OS.

Types of generated drivers

GS/OS generates drivers for four broad types of slot-resident, firmware-based I/O drivers:

- **BASIC and Pascal 1.1 drivers:** The Apple Super Serial Card and many third-party printer cards and parallel cards contain firmware drivers that conform to the Pascal 1.1 interface protocol. The Apple II Parallel Interface Card is a card that conforms to the BASIC interface protocol.

  A GS/OS character device driver is generated for slot-resident firmware I/O drivers that use the BASIC and Pascal 1.1 protocols (see the *Apple IIGS Firmware Reference* for more information). Each generated character device driver has a single DIB indicating that the driver supports only one device.

For BASIC firmware drivers, a BASIC generated driver is created. For Pascal 1.1 firmware drivers, a Pascal 1.1 generated driver is created. For firmware drivers that support both BASIC and Pascal 1.1 protocols, a Pascal 1.1 generated driver is created.

- **ProDOS drivers:** The Apple ProFile and several third-party hard disk drives include firmware-based drivers that conform to the ProDOS interface protocol on their controller cards.

  GS/OS generates a block device driver for slot-resident firmware I/O drivers that use the ProDOS interface (defined in the *ProDOS 8 Reference Manual*). One DIB is created for each logical ProDOS device; for example, a hard disk with two partitions is two logical devices and therefore has two DIBs.

- **SmartPort drivers:** The Apple II Memory Expansion Card (used as a RAM disk) is a peripheral card whose firmware driver follows the SmartPort protocol.

◆ *Note:* The Apple IIe UniDisk 3.5 card is not compatible with the Apple IIGS computer.

  Slot-resident firmware drivers that use the SmartPort protocol can in theory support up to 127 devices each, either character devices or block devices. See the *Apple IIGS Firmware Reference* for more information. GS/OS generates a DIB for each device interfaced to SmartPort. The device characteristics flag in the DIB indicates whether the device is a character device or a block device.

- **Extended SmartPort drivers:** An extended SmartPort driver has all the capabilities of a SmartPort driver and in addition supports direct memory transfer from any bank.

Device calls to generated drivers

All GS/OS generated drivers support these standard device calls:

- DInfo
- DStatus
- DControl
- DRead
- DWrite
- DRename

All generated drivers support the standard set of DStatus and DControl subcalls, although not all of those drivers perform meaningful actions with all of the subcalls. No generated drivers support driver-specific DStatus or DControl calls.

The rest of this chapter describes how generated drivers handle any of the above device calls differently than the standard ways documented in Chapter 1. Any calls or subcalls not discussed here are handled exactly as documented in Chapter 1.

DStatus ($202D)

Generated drivers support these DStatus subcalls:

- GetDeviceStatus
- GetConfigParameters
- GetWaitStatus
- GetFormatOptions

Only the following subcalls are implemented in a nonstandard way.

GetConfigParameters

Generated drivers have no configuration parameters. They always return no parameters, no errors, and a transfer count of $0000 0002 in the parameter block.

GetWaitStatus

Generated devices support wait mode only. A wait-mode value of $0000 is returned in the status list.

GetFormatOptions

This subcall applies only to block devices that implement the SmartPort interface with the new optional set of calls (see the *Apple IIGS Firmware Reference, 1MB Apple IIGS Update* for more information on the optional calls). The format of the options list is identical to that of the SmartPort specification and is returned unmodified in the status list.

DControl ($202E)

Generated drivers support these standard DControl subcalls:

- ResetDevice
- FormatDevice
- EjectMedium
- SetConfigParameters
- SetWaitStatus
- SetFormatOptions
- AssignPartitionOwner
- ArmSignal
- DisarmSignal

Only the subcalls described in this section are implemented in a nonstandard way.

ResetDevice

This call has no application with generated drivers and returns with no error.

SetConfigParameters

This call does not apply to generated drivers. Both generated character and block device drivers return with no error.

SetWaitStatus

All generated drivers support wait mode only. Attempting to set the mode to wait results in no error; attempting to set the mode to no wait results in error $22 (invalid parameter).

SetFormatOptions

This subcall applies only to block devices that implement the SmartPort interface with the new optional set of calls (see the *Apple IIGS Firmware Reference, 1MB Apple IIGS Update* for more information on the optional calls). The format of the options list is identical to that of the SmartPort specification and is passed directly to the device in the control list.

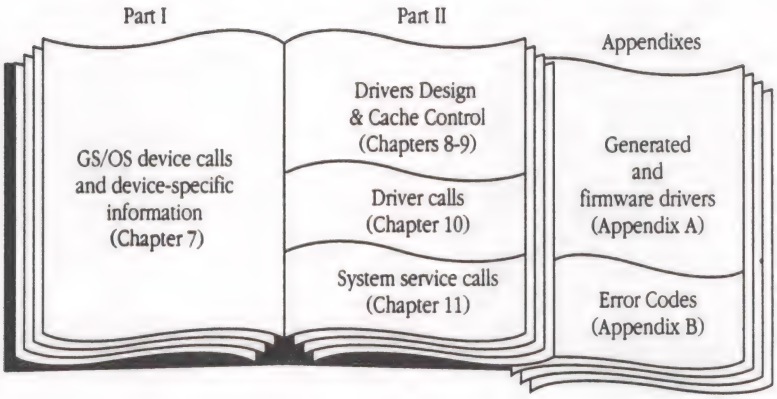
ArmSignal

This call has no application with generated drivers and returns with no error.

DisarmSignal

This call has no application with generated drivers and returns with no error.

Part II **Writing a Device Driver**



Chapter 8 **GS/OS Device Driver Design**

If you are planning to write a device driver for GS/OS, read this and the following chapters. GS/OS gives you a wide variety of capabilities to choose from in designing your driver; GS/OS drivers can

- access either block devices or character devices
- access devices either directly or through supervisory drivers
- respond to both a standard set of driver calls and any number of device-specific calls
- support multiple formatting options for their media
- be configurable by users or applications
- support caching of disk blocks to improve I/O performance
- include interrupt handlers
- include signal sources
- include signal handlers

This chapter describes the general structure of device drivers. Chapters 9 and 10 discuss additional concepts related to driver function and design. Driver calls, which every driver must handle, are described in Chapter 10. System service calls, which drivers can make to get information from GS/OS and to perform certain functions, are described in Chapter 11.

Driver types and hierarchy

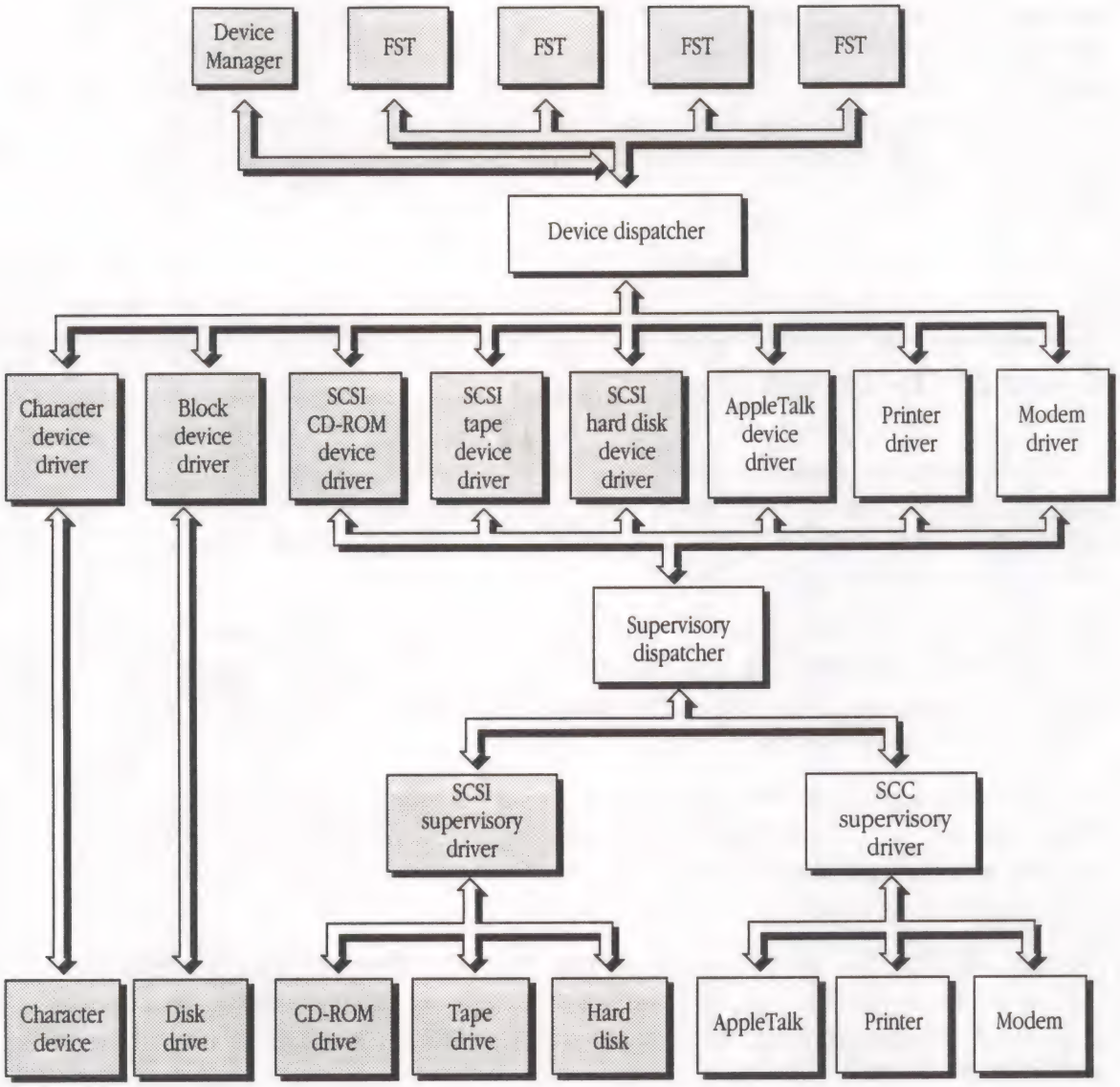
To summarize the discussion in the introduction to this reference, drivers can be classified in three ways:

- In relation to *devices,* there are two basic types of GS/OS drivers: block drivers and character drivers. Block drivers control hardware devices that handle data in blocks of multiple characters; character drivers control hardware devices that handle streams of individual characters.
- In relation to the GS/OS *initialization* routines, there is another classification of drivers: loaded drivers and generated drivers. Loaded drivers are loaded into memory at system startup or during execution; generated drivers are created by GS/OS to provide a GS/OS-compatible interface to slot-based firmware I/O drivers.
- In relation to the *hierarchy* of drivers and calls, there is another classification: device drivers and supervisory drivers. Device drivers accept driver calls directly from GS/OS and in turn access either a hardware device or a supervisory driver. The AppleDisk 3.5 driver is an example of a device driver. Supervisory drivers accept driver calls only through other device drivers and in turn access hardware devices. The SCSI Manager and the SCC Manager are both examples of supervisory drivers.

If you write a driver to work with GS/OS, it may be a block driver or a character driver, it may access hardware directly or go through a supervisory driver, but it must be a loaded driver. All loaded drivers, whether block drivers or character drivers, must accept (if not necessarily act on) the standard GS/OS driver calls documented in Chapter 10. Extensions to the standard calls are available for device-specific operations. Part I of this reference describes several examples of loaded and generated drivers.

Figure 8-1 shows how some specific device drivers and supervisory drivers might make up a particular configuration on the Apple IIGS computer.

■ **Figure 8-1** Hypothetical driver configuration



The diagram in Figure 8-1 includes examples of both block devices and character devices, and two hypothetical supervisory drivers: a SCSI supervisor and an SCC supervisor. Note that some block drivers can access their devices directly and don't need a supervisory driver. Note also that all SCSI device drivers must use the SCSI supervisory driver, and all drivers interfacing to the serial communications chip (SCC)—such as AppleTalk, printers, and modems—must use the SCC supervisory driver. The supervisor dispatcher is needed whenever there is one or more supervisory driver; the dispatcher routes calls to the proper supervisory driver.

Driver file types and auxiliary types

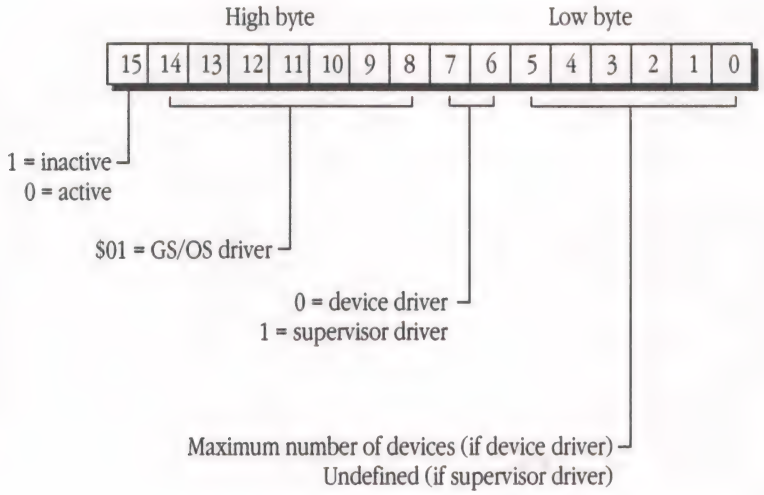
Loaded drivers are executable programs (load files). On disk, they should be in compacted format conforming to version 2.0 of object module format (OMF; see *GS/OS Reference*). All Apple IIGS driver load files must have a file type of $BB. They may also be in Express Load format.

The high-order byte of the auxiliary type field (`auxType`; see Figure 8-2) indicates the type of driver file and whether the driver is active (that is, whether it should be loaded and started up at boot time). If bit 15 of `auxType` is set (= 1), the driver is inactive; if bit 15 is clear (= 0), the driver is active.

The two high bits of the low-order byte of `auxType` indicate the type of GS/OS driver. Three types have been defined: device drivers, supervisory drivers, and boot drivers. The two remaining possible values are reserved. (For information on boot drivers, contact Apple Developer Support.)

The definition of the low six bits of the low byte of `auxType` depends on the driver type. For device drivers, those bits indicate the maximum number of devices supported by the driver; the device dispatcher uses that number to allocate memory for the device list. For supervisory drivers, the low six bits of `auxType` are not defined.

■ **Figure 8-2** Auxiliary type field for GS/OS drivers



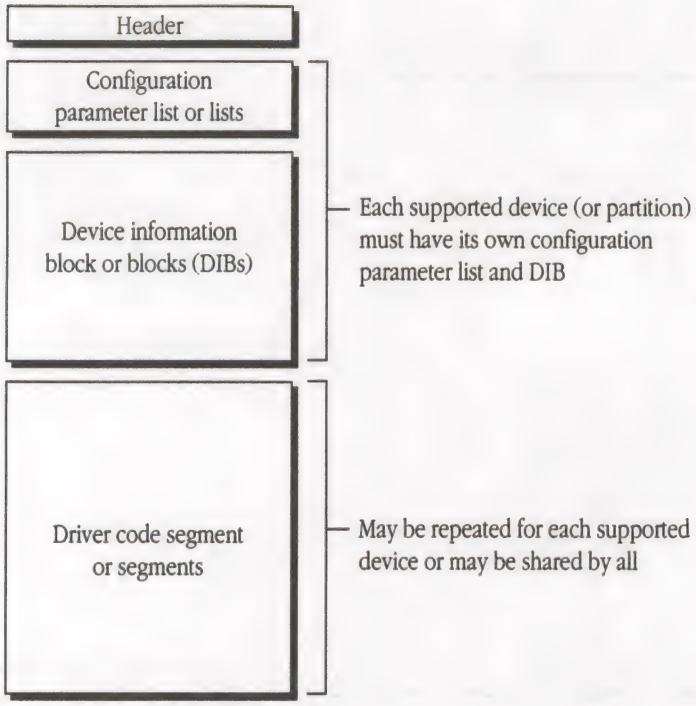
Device driver structure

A device driver consists of these basic parts, usually in this order:

- a driver header, which must always be the first part of the driver
- one current configuration list and one default configuration list for each DIB; for example, four DIBs result in eight configuration lists
- optionally, one or more DIBs (DIBs can be allocated dynamically by making calls to the Memory Manager)
- a format options table, if the driver can perform more than one type of formatting
- a driver code section

Figure 8-3 diagrams the general structure of a GS/OS device driver.

■ **Figure 8-3** GS/OS device-driver structure



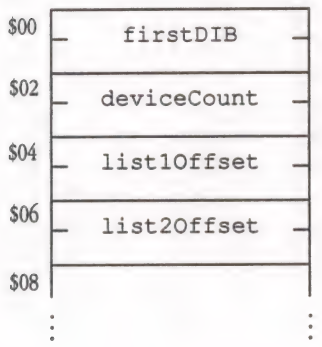
If the device driver supports more than one device, then one DIB, a current configuration list, and a default configuration list must be provided for each device. The current list is a list that reflects the current values of the driver parameters, and the default list is a list of default values. Each device may have its own individual configuration and default lists or may share those lists with other devices supported by the driver.

A driver always contains one DIB per device supported by the driver; multiple devices, even logical devices such as partitions on a disk, cannot share the same DIB. If several supported devices use the same configuration parameters, the driver need have only a single set of configuration parameters for them; offsets in the driver header can then reference the same configuration lists for each device.

The device driver header

The device driver header specifies where the configuration lists and DIBs are located. The device dispatcher needs that information when loading drivers and building the device list. Using an InitialLoad call to the System Loader (see Chapter 7 in *GS/OS Reference*), the device dispatcher loads only the driver's static load segment, which contains its code, DIBs, and configuration lists.

A device driver header has this format:

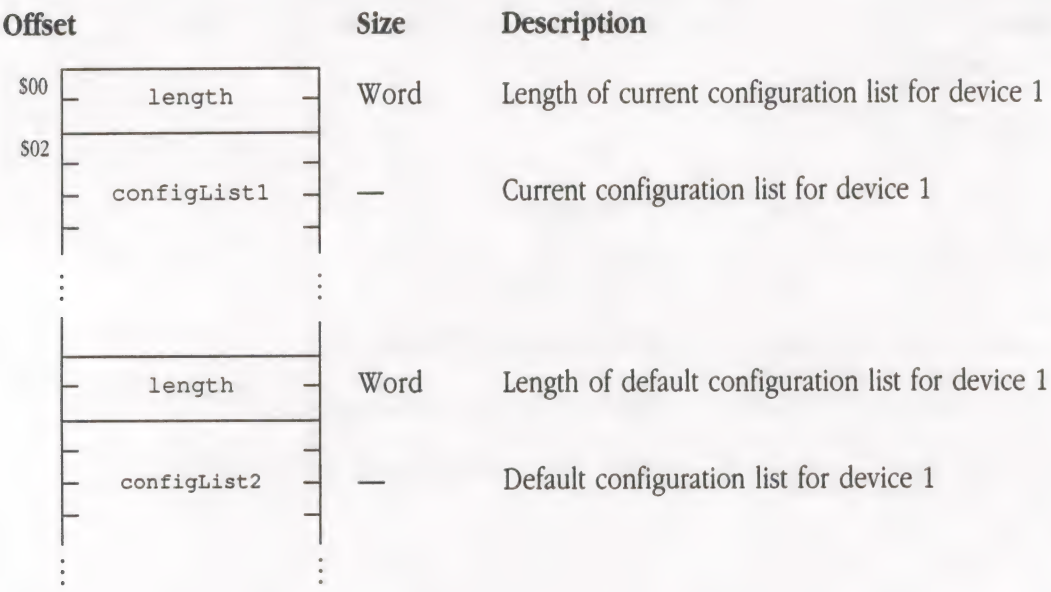


The header fields following `deviceCount` constitute the configuration list offset table; it is a word list of offsets from the beginning of the load segment (the beginning of the driver header) to the first byte of the current configuration list for each device supported by the driver. If there is no configuration list for a device, the entry for that device in the configuration list offset table must be 0.

Configuration lists

A **configuration list** is a table of device-dependent information used to configure a specific device. Each device supported by a driver needs *two* such lists: the first one shows the device's current configuration settings, and the second one holds default values.

A configuration list has a very simple structure as far as GS/OS is concerned: It consists of a length word (containing the number of bytes in the list) followed by the device's configuration parameters. For a driver that supports a single device, the configuration lists would look like this:



Configuration lists are driver specific in content, but they must follow these rules:

- The first word of the list, the length word, must be a byte count; the length of the rest of the list must be in bytes. A length word of 0 indicates an empty list.
- Each parameter in the list must begin on a word boundary (no parameters should be an odd number of bytes in length).
- Each **current configuration list** must have an accompanying **default configuration list,** identical in size and format. The default configuration list contains the default driver configuration values and is never altered.
- The default configuration list must immediately follow the current configuration list in the driver.

An application (through the Device Manager) or an FST obtains a copy of a driver's current configuration parameters by making the call Driver_Status; the driver passes a copy of the current list to the caller in the status list. A caller modifies a driver's configuration parameters by making the call Driver_Control; the caller passes the desired configuration list to the driver in the control list; the driver copies that information into its current list. See Chapter 10 for more information about the Driver_Status and Driver_Control calls.

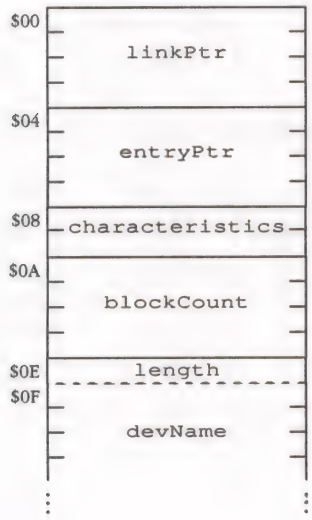
Any time that an application or FST requests that a device revert to its default parameters, the driver should respond by copying the contents of the default configuration list into the current configuration list.

Device information block

Every device accessed by a driver needs a device information block (DIB). In a driver, the DIB is a table of information that describes the device's characteristics; when the driver is loaded into memory, GS/OS uses that information to identify and keep track of the device.

Each DIB has the format shown in Figure 8-4. Descriptions of the individual parameters follow the figure.

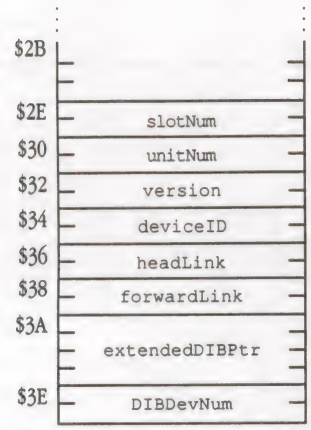
■ **Figure 8-4** Device information block (DIB)



Size	Description
Longword	Pointer to next DIB
Longword	Pointer to driver entry point
Word	Characteristics of device
Longword	Number of blocks on device
String	Name of device (Pascal string; ASCII, high bit clear)

[continued]

■ **Figure 8-4** Device information block (DIB) [continued]



Offset	Size	Description
$2E slotNum	Word	Slot number of device installed
$30 unitNum	Word	Unit number of device in slot
$32 version	Word	Version number of device driver
$34 deviceID	Word	General type of device
$36 headLink	Word	Device number of first linked device
$38 forwardLink	Word	Device number of next linked device
$3A extendedDIBPtr	Longword	Pointer to additional device information
$3E DIBDevNum	Word	Initial device number (assigned at startup)

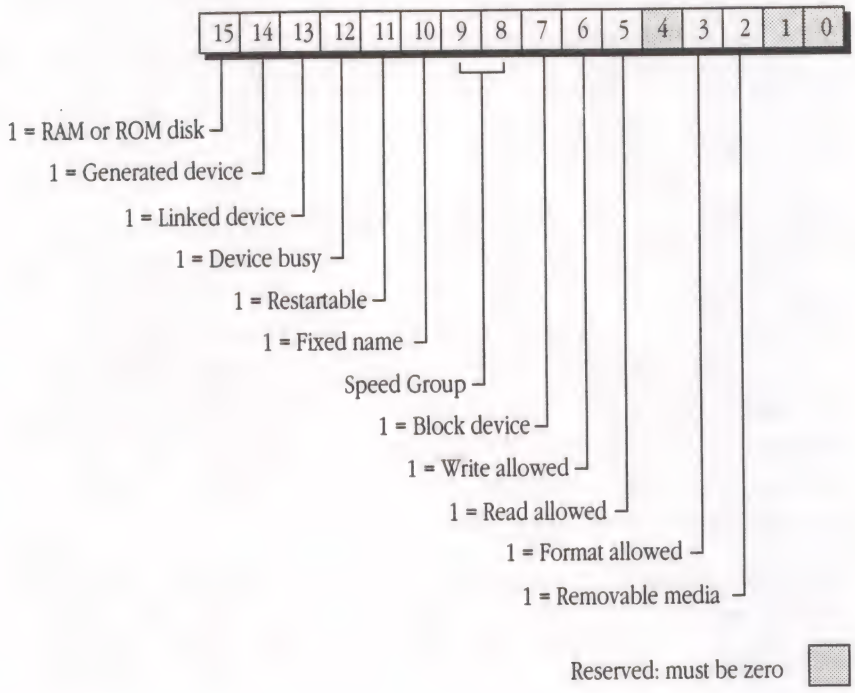
Here is what each parameter in the DIB means:

`linkPtr` Link pointer: longword pointer to the next DIB for device drivers supporting multiple DIBs. If the device driver supports only a single DIB, the link pointer should be set to NIL. The device dispatcher uses the link pointer only to install the device drivers into the device list.

`entryPtr` Entry pointer: longword pointer to the device driver's entry point.

`characteristics` Device characteristics: word value that defines whether or not the device supports certain features. The current definition for this word is shown in Figure 8-5. Shaded bits are reserved and should be set to 0.

■ **Figure 8-5** Device characteristics word



In the device characteristics word, *linked device* means that the device is one of several partitions on a single, removable medium. *Device busy* is maintained by the device dispatcher to prevent reentrant calls to a device.

Speed group defines the speed at which the device requires the processor to be running. Speed group has these binary values and meanings:

Binary value	Speed
00	Apple IIGS normal speed
01	Apple IIGS fast speed
10	Accelerated speed
11	Not speed dependent

See the description of the system service call SET_SYS_SPEED in Chapter 11 for more information.

Restartable defines whether or not the device driver is to be purged or is to remain in memory when switching between a ProDOS 8 application and a GS/OS application. If this bit is a 1, the driver is restartable and will not be purged when quitting from a GS/OS application program to a ProDOS 8 application program. If this bit is a 0, the driver is not restartable and will be purged.

Before the introduction of GS/OS system software 5.0, device drivers were always loaded from disk and thus may have contained preinitialized data. This data may have been modified during the normal execution of the device driver. In order to make these device drivers restartable, the device driver Shutdown call must be modified to reset the variables that have been modified during device driver execution, so that subsequent Startup calls to the driver will function properly. This is an additional task for the device driver Shutdown call and does not in any way diminish previous requirements on the driver Shutdown call.

A flag is available that indicates whether a restart is due to a warm start (restarting from ProDOS 8) or a cold start (power switch being turned on). This flag is at $E1/01D0. A warm start sets bit 0, while a cold start clears it.

Fixed name defines whether or not the device driver name can be changed in the memory-resident DIB. If this bit is a 1, the DRename call will not alter the device driver name.

`blockCount` Block count: longword value that is the total number of blocks accessible on the device. It applies to block devices only; for character devices, it should be set to 0. The value of `blockCount` may be dynamic (changing) if the device supports multiple types of removable media or partitioned removable media. In this case, any status call that detects on-line and disk-switched conditions should update this parameter after media insertion.

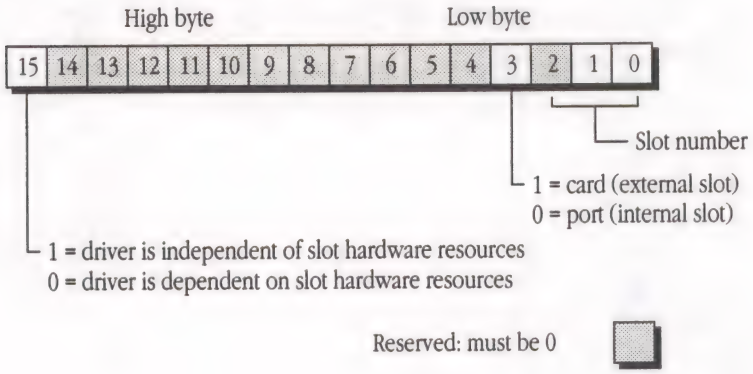
`devName` Device name: 32-byte field that contains the device's name as a Pascal string. It consists of a length byte followed by up to 31 bytes of ASCII characters—uppercase only, high bit clear (= 0). Note that the initial period (.), which defines a device name to the system, is not part of the name in this field.

`slotNum` Slot number: word value indicating the slot in which the device hardware resides. Bits 0 through 2 define the slot, and bit 3 indicates whether it is an internal port (controlled by firmware within the Apple IIGS computer) or an external slot containing a card with its own firmware.

For a given slot number, either the external slot or its equivalent internal port is active (switched in) at any one time; bit 15 indicates whether or not the device driver must access the peripheral card's I/O addresses. For more information on those addresses, see the *Apple IIGS Hardware Reference*. Figure 8-6 shows the format of the slot-number word.

If you are designing a loaded driver to replace a generated driver, you must use the same slot number that would have been generated for the driver. To determine whether an internal or external slot has been used, examine the soft switch SLTROMSEL for slots 1, 2, 4, 5, 6, and 7 or examine the soft switch RDC3ROM for slot 3. See the *Apple IIGS Firmware Reference* and the *Apple IIGS Hardware Reference* for more information on soft switches.

■ **Figure 8-6** Slot-number word



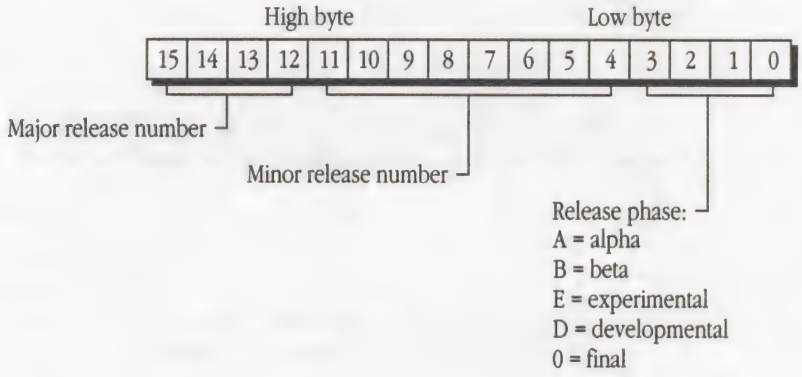
△ **Important** The driver must set bits 14–4 to 0 in the slot-number word. △

`unitNum` Unit number: a word value indicating the number of the device within the slot. Multiple devices within a slot are numbered consecutively. This is not a global unit number relating to the device list.

If you are designing a loaded driver to replace a generated driver after booting, you must use the same unit number that would have been generated for the driver. For ProDOS, the drive number is equal to the unit number; for a SmartPort device, the SmartPort unit number is equal to the unit number.

`version` Driver version: word value indicating the version number of the driver that controls this device. Loaded drivers have their own version numbers; generated drivers may use the version number obtained from the slot-resident firmware interface. Figure 8-7 shows the fields of the driver version word.

■ **Figure 8-7** Driver version word



◆ *Note:* This parameter has a different format from the version parameter returned from the GS/OS call GetVersion.

`deviceID` Device ID: word value specifying the general type of device associated with this DIB. Table 8-1 shows the presently defined devices and their device IDs. It is a guide to assigning device IDs and does not in any way imply that Apple Computer, Inc., intends to provide any of the listed devices or drivers for them.

- *Note:* Device IDs are assigned by Apple Computer, Inc. Contact Apple Developer Technical Support if you have a specific need for a device ID.

■ **Table 8-1** Device IDs

ID	Description	ID	Description
$0000	Apple 5.25 drive (includes Unidisk, Duodisk, DiskIIc, and Disk II drives)	$000F	ROM disk
$0001	ProFile (5 MB)	$0010	File server
$0002	ProFile (10 MB)	$0011	(Reserved)
$0003	Apple 3.5 drive (includes UniDisk 3.5 drive)	$0012	Apple Desktop Bus
$0004	SCSI device (generic)	$0013	Hard disk drive (generic)
$0005	SCSI hard disk drive	$0014	Floppy disk drive (generic)
$0006	SCSI tape drive	$0015	Tape drive (generic)
$0007	SCSI CD-ROM drive	$0016	Character device (generic)
$0008	SCSI printer	$0017	MFM-encoded disk drive
$0009	Modem	$0018	AppleTalk network (generic)
$000A	Console	$0019	Sequential-access device
$000B	Printer	$001A	SCSI scanner
$000C	Serial LaserWriter	$001B	Other scanner
$000D	AppleTalk LaserWriter	$001C	LaserWriter SC
$000E	RAM disk	$001D	AppleTalk main driver
		$001E	AppleTalk file server driver
		$001F	AppleTalk RPM driver

`headLink` Head device link: word value that is the device number of the first device in a chain of linked devices (separate partitions on a single removable medium). Using the head link and forward link as "pointers," GS/OS or an application can find all DIBs associated with a partitioned disk and mark them all on line or off line as needed.

A value of 0 indicates that there are no devices linked to this device.

`forwardLink` Forward device link: word value that is the device number of the next device in a chain of linked devices (separate partitions on a single removable medium). Using the head link and forward link as "pointers," GS/OS or an application can find all DIBs associated with a partitioned disk and mark them all on line or off line as needed.

A value of 0 indicates that there are no devices linked to this device.

`extendedDIBPtr`
: Extended DIB pointer: a longword pointer to a second, device-specific structure containing more information about the device associated with this DIB. This field allows a driver to maintain additional device information for its own purposes.

`DIBDevNum`
: DIB device number: a word value that is the device number initially assigned (during startup) to the device associated with this DIB. This parameter is used to maintain the head link and the forward link between devices within a loaded driver supporting multiple volumes on a single removable medium.

  Note that if a loaded device replaces a generated boot device driver, then this parameter in its DIB will not be valid until the next access of the device.

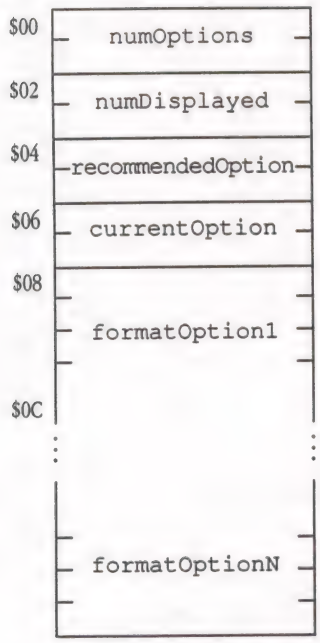
- *Note:* A driver may extend the DIB for its own internal use. The device call DInfo returns the value in the DIB field `extendedDIBPtr`, so any driver-specific extensions that use the extended DIB are available through DInfo. The driver can also expand the current data structure, but the information in those fields will not be returned by DInfo.

Format options table

Some block devices can be formatted in more than one way. Formatting parameters can include such variables as file system group, number of blocks, block size, and interleave. Each driver that supports media variables (multiple formatting options) contains one or more **format options tables,** the formatting options for a particular type of device controlled by the driver.

When a block driver receives the GetFormatOptions subcall of the driver call Driver_Status, it returns a copy of its format options table for the particular device requested. One of the options can then be selected and applied (by an FST, for example) with the Driver_Control subcalls SetFormatOptions followed by FormatDevice. Device drivers that do not support media variables return a transfer count of 0 and generate no error. Character drivers do nothing and return no error from this call.
Figure 8-8 shows the overall structure of the format options table; Figure 8-9 shows the structure of each format options entry within the list.

■ **Figure 8-8** Format options table



The value specified in the `currentOption` parameter is the format option of the current on-line media. If a driver can report this parameter, it should do so. If the driver cannot detect the current option, it should indicate *unknown* by returning $0000.

Of all the options in the format options table, one or more may be displayed in the initialization dialog presented to the user when initializing a disk (see the calls Format and EraseDisk in Chapter 10 of *GS/OS Reference*). The options that are to be displayed must come first in the table. (Undisplayed options are available so that drivers can provide FSTs with logically different options that are physically identical and therefore needn't be duplicated in the dialog.)

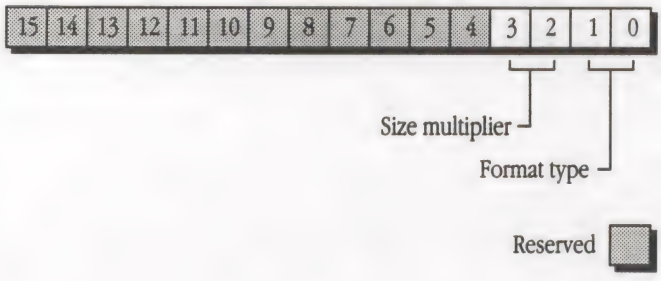
■ **Figure 8-9** Format options entry

Offset		Size	Description
$00	formatOptionNum	Word	Number of this option
$02	linkRefNum	Word	Number of linked option
$04	flags	Word	(See Figure 8-10 below)
$06	blockCount	Longword	Number of blocks supported by device
$0A	blockSize	Word	Block size in bytes
$0C	interleaveFactor	Word	Interleave factor (in ratio to 1)
$0E	mediaSize	Word	Media size (see Figure 8-10 below)

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

Bits within the flags word are defined as shown in Figure 8-10.

■ **Figure 8-10** Format option flags word



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

00 Universal format

01 Apple format

10 Non-Apple format

11 (Not valid)

Size multiplier is used, in conjunction with the format options parameter `mediaSize`, to calculate the total number of bytes of storage available on the device. Size multiplier can have these binary values and meanings:

00 `mediaSize` is in bytes

01 `mediaSize` is in kilobytes

10 `mediaSize` is in megabytes

11 `mediaSize` is in gigabytes

For examples, see the description of the GS/OS call DStatus in Chapter 1 and of the driver call Driver_Status in Chapter 10.

Driver code section

The driver code section must accept all calls and return appropriately. Beyond that, the implementation of the driver is up to the programmer.

Some points to consider when designing a device driver are the following:

- If you are writing a character driver, be sure to include an internal driver-open flag that notes the current state of the driver. Inspect and set the flag properly on Driver_Open and Driver_Close calls, using the calls to return an error if appropriate. See Chapter 10 for details on Driver_Open and Driver_Close.
- If your block driver is capable of detecting disk-switched or off-line conditions, it reports that information as an error from I/O calls but as device status information (*not* as an error) from a status call. Errors should be reserved for conditions that cause a call such as a Read, Write, or Format to fail.
- Because device driver routines typically execute during GS/OS calls, and because GS/OS is not reentrant and therefore cannot accept a call while another is in progress, device drivers normally cannot make GS/OS calls. This includes System Loader calls that must make GS/OS calls such as when loading a dynamic segment.

If some of your device driver routines need to make GS/OS calls, you can use the Scheduler in the Apple IIGS Toolbox to schedule a task for completion after the operating system finishes the current call. See the *Apple IIGS Toolbox Reference* for more information. As an alternative, consider making some routines into signal handlers instead. See Chapter 9 of *GS/OS Reference* for more information.

△ **Important** No work space is available on GS/OS direct page for use by either device drivers or supervisory drivers. Do not use GS/OS direct page as a work space *under any circumstances.* Using GS/OS direct page as work space could result in damaged disk media. This warning applies to all versions of GS/OS system software. △

A device driver or supervisory driver requiring direct-page space can acquire its own direct page at startup; the driver must then release this memory at shutdown.

How GS/OS calls device drivers

Drivers receive calls from GS/OS through the **device dispatcher.** This section describes the device dispatcher, defines the device driver execution environment, and lists the calls (driver calls) that a device driver must accept from the device dispatcher. Driver calls are fully documented in Chapter 10.

The device dispatcher and the device list

The device dispatcher is the main GS/OS interface to drivers. At startup, the device dispatcher installs all drivers; during execution, it is the channel through which all calls to drivers pass. The device dispatcher accepts I/O calls from file system translators or the Device Manager, adds any necessary parameters, and sends the calls on to individual device drivers. Device information requests through the Device Manager are handled by the device dispatcher itself, usually with driver access. The device dispatcher also generates the startup and shutdown calls that are sent to drivers.

The device dispatcher constructs and maintains the device list, a list of all installed device drivers in the system, including both loaded and generated drivers. Devices under GS/OS are specified by **device number,** which is the current position of the device in the device list. Device calls, for example, use the device number as an input parameter; the device dispatcher uses it as an index to the device list when setting up the DIB pointer (an input parameter to the equivalent driver call) prior to calling a device driver.

At system startup, the device dispatcher loads and installs all supervisory drivers first. It then loads and installs all loaded device drivers. Finally, it creates and installs any needed generated drivers. During execution, the device dispatcher can add more devices to the device list, as explained next. A device is considered *installed* when its driver has successfully completed a startup call and its DIB has been placed in the device list.

Dynamic driver installation

The device list under GS/OS is not always static. Because GS/OS supports removable partitionable media on block devices, it must also provide a mechanism for dynamically installing devices in the device list as new partitions come on line. The system service call INSTALL_DRIVER has been provided for this purpose; it is described in Chapter 11. Because of this call, the GS/OS device list can grow during program execution. (However, the device list cannot shrink; there is no mechanism for removing devices from the device list.)

To dynamically install and start up a driver, take the following steps:

1. Make the INSTALL_DRIVER call.
2. Check for out-of-memory or busy errors. If either of these errors occur, no drivers are installed. Postpone installation until later.

   If neither of these errors occur, the drivers will be installed in the system the next time a driver returns to the device dispatcher.

When a new device comes on line, the application receives no notification that the device list has changed size. An application that scans block devices should always begin by issuing a DInfo call to device $0001 and should continue up the device list until error $11 (invalid device number) occurs. The DInfo call should have a parameter count of $0003 or more to give the application each device's device-characteristics word. If the newly installed device is a block device with removable media, the application should make a status call to it.

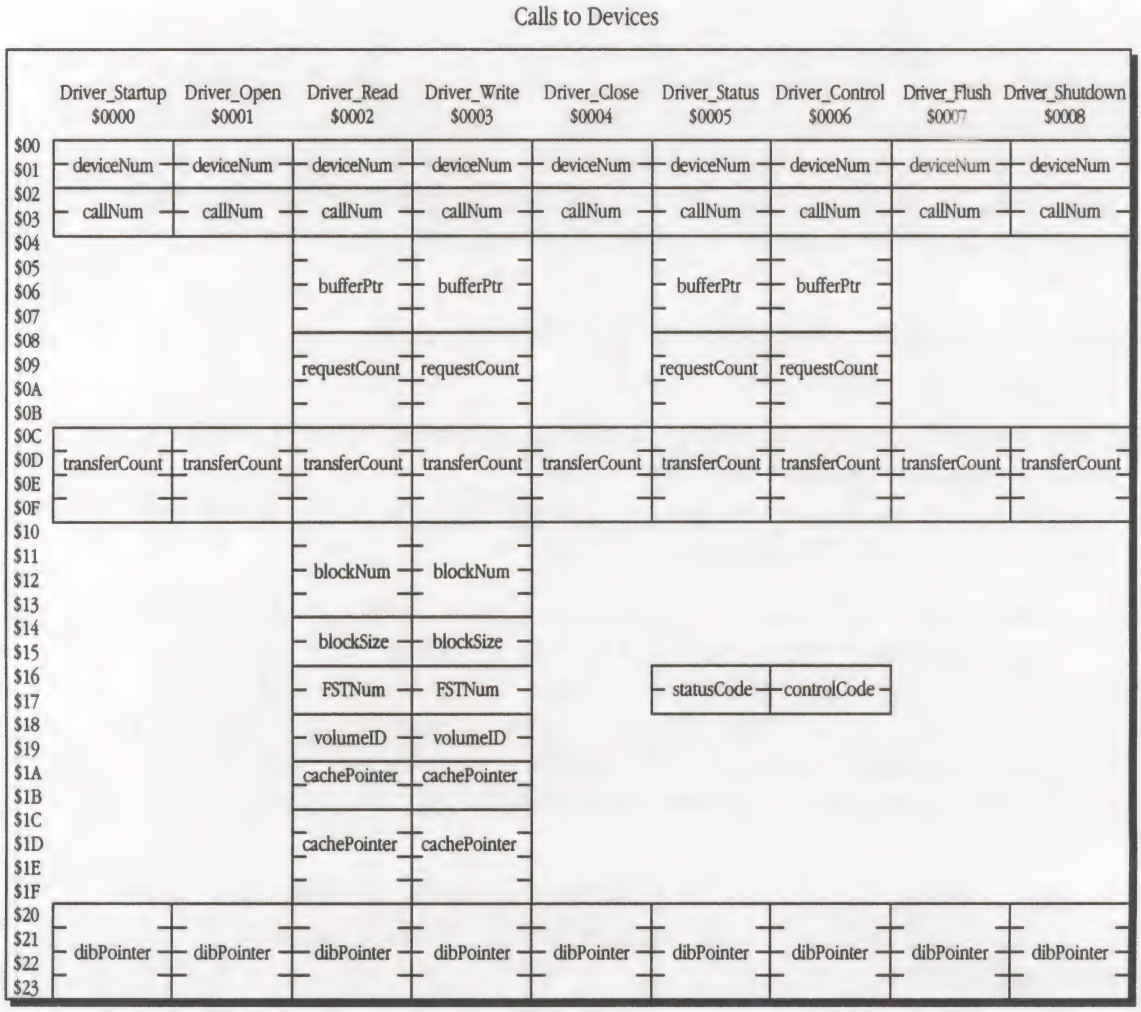
If the device dispatcher gets an error during the Driver_Startup call, the new driver will not be included in the device list, nor will the memory manager dispose of the memory allocated to the driver.

Direct-page parameter space

Below the application level in GS/OS, many calls pass parameters by using a single parameter block on the Apple IIGS direct page. This same direct-page parameter block is shared among all FSTs, the Device Manager, the device dispatcher, all device drivers, system service calls, and the GS/OS Call Manager. All driver calls share those locations (addresses $00–$23), although not all locations have the same meaning for all calls or are even used by all calls.

Figure 8-11 shows the format of the GS/OS direct-page parameter space.

■ **Figure 8-11** GS/OS direct-page parameter space



For most calls to drivers, the device dispatcher sets up any needed input parameters on the GS/OS direct page. Exceptions are those parameters already supplied by the application or FST making the call. A driver can therefore count on all its direct-page parameters being properly set each time it receives a driver call.

Dispatching to device drivers

For every driver call, the device manager sets up the device driver execution environment shown in Table 8-2, completes the GS/OS direct-page parameter block for the call, sets the transfer count parameter on direct page to 0, and calls the device driver's entry point with a JSL instruction. Boldface entries in the table indicate the components of the environment that the driver routine must restore before returning.

■ **Table 8-2** Device driver execution environment

Component	State
Registers	
A	Call number*
X	Unspecified
Y	Unspecified
D	**Base of GS/OS direct page**
S	**Top of GS/OS stack**
Data Bank	Current value
P register flags	
e	**0 (native mode)**
m	**0 (16-bit)**
x	**0 (16-bit)**
i	0 (enabled)
c	Unspecified†
decimal	0
Speed	Fast

*The accumulator contains the call number on entry; on exit, it should contain the error code (if an error occurred) or 0 (if no error).

†On exit, the carry flag should be set (= 1) if an error occurred or clear (= 0) if no error.

The current value in the Data Bank register (DBR) is preserved by the device dispatcher.

Device drivers should not permanently modify any GS/OS direct-page location except `transferCount`, which indicates the number of bytes processed by the driver.

△ **Important** Drivers should never access GS/OS direct page using absolute or absolute long addressing modes. The location of GS/OS direct page is not specified and may not be preserved in any future versions of the operating system. △

Device drivers must return from calls with an RTL instruction, in full native mode, with the portions of the environment preserved as shown in boldface in Table 8-2. The carry flag and accumulator should reflect the error status for the call, as indicated in the footnotes to Table 8-2.

◆ *Note:* When a driver call returns to the device dispatcher, the device dispatcher postprocesses any error codes from the device. If either a disk-switched or an off-line error is returned by the device, the device dispatcher sets an internal error flag for the device to indicate that a disk-switched condition has occurred. GS/OS, for example, uses this status to discard cached blocks and mark volume control records as swapped out.

This fact also means that drivers, which should not return disk-switched or off-line conditions as errors from status calls, must explicitly notify GS/OS when a status call detects a disk-switched or an off-line condition. See descriptions of the driver call Driver_Status (Chapter 10) and the system service call SET_DISKSW (Chapter 11) for more information.

List of driver calls

When an application makes a device call through the Device Manager or a file I/O call through an FST, the call is translated into a driver call and passed on through the device dispatcher to the device driver. In addition, FSTs and the device dispatcher itself make certain driver calls that are not translations of application-level calls. A device driver needs to accept and act on all those driver calls. The calls are

Call no.	Name	Description
$0000	Driver_Startup	Prepares a device for all other device-related calls
$0001	Driver_Open	Prepares a character device for conducting I/O transactions
$0002	Driver_Read	Reads data from a character device or a block device
$0003	Driver_Write	Writes data to a character device or a block device
$0004	Driver_Close	Resets the driver to its nonopen state
$0005	Driver_Status	Gets information about the status of a specific device
$0006	Driver_Control	Sends control information or requests to a specific device
$0007	Driver_Flush	Writes out any characters in a character driver's buffer
$0008	Driver_Shutdown	Prepares a device driver to be purged

For a more detailed explanation of driver calls, see Chapter 10, "GS/OS Driver Call Reference."

How device drivers call GS/OS

GS/OS calls device drivers through driver calls. Device drivers call GS/OS through system service calls. System service calls constitute a standardized mechanism for passing information and providing services among the low-level components of GS/OS, such as FSTs and device drivers.

System service calls exist for various purposes: to perform disk caching, to manipulate buffers in memory, to set system parameters such as execution speed, to send a signal to GS/OS, to call a supervisory driver, and to perform other tasks.

Several of the system service routines are available to device drivers. Access to these routines is through a system service dispatch table located in bank $01. These are some of the available routines:

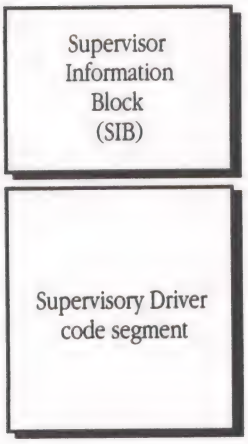
Name	Description
CACHE_FIND_BLK	Searches for a disk block in the cache
CACHE_ADD_BLK	Adds a block of memory to the cache
CACHE_DEL_BLK	Deletes a specified block of memory from the cache
ALLOC_SEG	Allocates a memory segment
RELEASE_SEG	Releases a previously requested memory segment
DEREF	Dereferences a virtual pointer
SET_SYS_SPEED	Controls processor execution speed
LOCK_MEM	Locks a memory segment, keeping it from being relocated
UNLOCK_MEM	Unlocks a memory segment, allowing it to be relocated
MOVE_INFO	Moves data into or out of the cache block
SIGNAL	Informs GS/OS of the occurrence of a signal
SET_DISKSW	Notifies GS/OS of a disk-switched or off-line condition
SUP_DRVR_DISP	Makes a supervisory-driver call
INSTALL_DRIVER	Dynamically installs a device into the device list
DYN_SLOT_ARBITER	Returns status of a slot
UNBIND_INT_VEC	Deletes a previously created link between an interrupt vector and its handler

Supervisory-driver structure

Supervisory drivers accept calls from device drivers and in turn access hardware devices. Supervisory drivers are used where several different (but related) device drivers access several different (but related) types of hardware devices through a single hardware controller, all under the coordination of the supervisory driver.

Supervisory drivers are simpler in overall structure than device drivers. As shown in Figure 8-12, a supervisory driver consists of a supervisor information block (SIB) and the supervisory-driver code section.

■ **Figure 8-12** Supervisory-driver structure

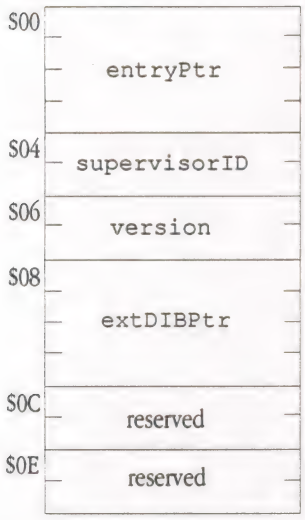


The supervisor information block (SIB)

The supervisor information block (SIB) is a supervisory driver's equivalent to a DIB; it identifies the supervisory driver to the system. At startup, GS/OS constructs a **supervisor list,** equivalent to the device list; it lists pointers to the SIBs of all installed supervisory drivers.

A supervisor information block has the format shown in Figure 8-13.

■ **Figure 8-13** Supervisor information block (SIB)



The defined parameters in the SIB have these meanings:

`entryPtr` Entry pointer: a longword pointer that indicates the main entry point for the supervisory driver.

`supervisorID` Supervisor ID: a word value that specifies the type of supervisory driver. Table 8-3 shows the currently defined values for supervisor ID.

■ **Table 8-3** Supervisory IDs

ID	Description
$0001	AppleTalk supervisory driver
$0002	SCSI supervisory driver
$0003–$FFFF	(Reserved)

◆ *Note:* Supervisor IDs are assigned by Apple Computer, Inc. Contact Apple Developer Technical Support if you have a specific need for a supervisor ID.

`version`	Version: a word value that specifies the version number of the supervisory driver. This parameter has the same format as the driver version word in a device driver DIB (the SmartPort version format). See Figure 8-7.
`extDIBPtr`	Extended DIB pointer: a longword pointer to the name of the extended DIB.
Reserved	Two words have been reserved in the SIB for future expansion. They should contain a value of $0000.

Supervisory-driver code section

The content of the code section of a supervisory driver is strongly device dependent and device driver dependent. A supervisory driver must have a single entry point and must include code routines to accept the standard supervisory-driver calls listed later in this chapter (and under "About Supervisory-Driver Calls" in Chapter 10). It can also contain routines to handle any supervisor-specific calls defined among it and its device drivers; it is the supervisor-specific calls that implement all driver I/O.

All driver calls to its dependent device driver(s) are translated into supervisor-specific calls to the supervisory driver. The supervisory driver in turn accesses the appropriate hardware device.

How device drivers (and GS/OS) call supervisory drivers

All supervisory-driver calls pass through the supervisor dispatcher. Comparable to the device dispatcher, the supervisor dispatcher handles informational calls (from device drivers), passes on I/O calls (from device drivers) to supervisory drivers, and generates the startup/shutdown calls that are sent to supervisory drivers.

At startup, the supervisor dispatcher creates a supervisor list, a list of pointers to all SIBs. Each installed supervisory driver is identified by **supervisor number,** its position in the supervisor list.

For each supervisory-driver call, the supervisor dispatcher sets up the **supervisor execution environment,** as shown in Table 8-4, and calls the supervisory driver's entry point with a JSL instruction. Boldface entries in the table indicate the components of the environment that the supervisory-driver routine must restore before returning.

■ **Table 8-4** Supervisor execution environment

Component	State
Registers	
A	Call number/supervisor ID*
X	Unspecified
Y	Unspecified
D	**Base of GS/OS direct page**
S	**Top of GS/OS stack**
Data Bank	Current value
P register flags	
e	**0 (native mode)**
m	**0 (16-bit)**
x	**0 (16-bit)**
i	0 (enabled)
c	Unspecified†
Speed	Fast

*The accumulator contains the call number or supervisor ID on entry; on exit, it should contain the supervisor number or error code (nonzero if an error occurred, zero if no error). See individual call descriptions.

†On exit, the carry flag should be set (= 1) if an error occurred or clear (= 0) if no error.

The value of the Data Bank register is preserved by the supervisor dispatcher. If appropriate, a pointer to a parameter block is set up on GS/OS direct page *by the device driver* prior to calling the supervisory driver. See Figure 10-5, under "About Supervisory-Driver Calls," in Chapter 10.

A supervisory driver requiring direct-page space could acquire its own direct page at startup; the supervisory driver must then be sure to release this memory at shutdown.

△ **Important** Drivers should never access GS/OS direct page using absolute or absolute long addressing modes. The location of GS/OS direct page is not specified and may not be preserved in any future versions of the operating system. △

Supervisory drivers must return from calls with an RTL instruction, in full native mode, with the appropriate portions of the supervisor execution environment preserved, as shown in boldface in Table 8-4. The carry flag and accumulator should reflect the error status for the call or results, as indicated in the footnotes to Table 8-4.

Here are a list and brief description of the supervisory-driver calls that device drivers can make or that supervisory drivers must respond to:

Call no.	Supervisor no.	Call name	Explanation
$0000	(Nonzero)	Supervisor_Startup	Prepares the supervisory driver to receive calls from device drivers
$0001	(Nonzero)	Supervisor_Shutdown	Releases any system resources allocated at startup
$0000	$0000	Get_Supervisor_Number	Returns the supervisor number for the supervisory driver with a given supervisor ID
$0001	$0000	Set_SIB_Ptr	Sets the direct-page SIB pointer for a specified supervisory driver
$0002–$FFFF	(Nonzero)	(Driver-specific calls)	For use by device drivers

See Chapter 10 for more information.

Chapter 9 **Cache Control**

GS/OS provides for **disk caching,** whereby frequently read disk blocks are kept in memory for faster access. Individual block drivers may or may not implement caching; this chapter shows you how to write your driver to support caching if you want it to.

Drivers and caching

Under GS/OS, caching is the process in which frequently accessed disk blocks are kept in memory to speed subsequent accesses to those blocks. The user (through the Control Panel program) can control whether caching is enabled and what the maximum cache size can be. The driver, however, is responsible for making caching work. This section discusses the design of the GS/OS cache and shows what calls are needed to implement it.

Except for one special case, the GS/OS cache is a write-through cache. When an FST issues a Write call to a device driver, the driver writes the same data to the block in the cache and the equivalent block on the disk. Never does the block in the cache contain information more recent than that in the disk block.

The one special case where the GS/OS cache is not write-through is when a write-deferred session is in effect. In that case, data written to the cache is kept there until the application makes an EndSession call that terminates the session and flushes the cache to the disk.

Like most caching implementations, the GS/OS cache uses a least recently used (LRU) algorithm: Once the cache is full, the least recently used (= read) block in the cache is sacrificed for the next new block that is written.

Cache memory is obtained and released on an as-needed basis. For example, if the user or an application selects 32 KB as the cache size, this amount is not directly allocated for specific use by GS/OS. Only as individual blocks are cached is the necessary amount of memory (up to 32 KB in this case) assigned to the cache.

The size of a block in the cache is essentially unrestricted, limited only by the maximum size of the cache itself. GS/OS makes no assumptions about the size of the block to be cached; it uses whatever block size is requested.

- *Note:* These features differ from caching on the Macintosh computer, in which the Cache Manager holds exclusive control over the entire amount of cache memory and deals in 512-byte blocks only.

Cache calls

The following brief descriptions show what the available cache calls are and what they do. Cache calls are system service calls; they are described in more detail in Chapter 11.

CACHE_FIND_BLK	Searches the cache to find the specified cached block and, if it finds it, sets a pointer on the direct page to the cache.
CACHE_ADD_BLK	Attempts to add the specified block to the cache and sets a pointer to the cache. If there is not enough room left in the cache for the specified block, it makes space available by deleting cached blocks.
MOVE_INFO	Copies the block into or out of the cache.
SET_DISKSW	Deletes from the cache any blocks belonging to a device whose disk has been switched.

How drivers cache

If you are writing a driver that will support caching, it should perform the following tasks on reading from or writing to its device.

On a Read call

When the driver receives control, its direct-page parameters have already been set up by the caller (Device Manager or FST); see the description of Driver_Read in Chapter 10. If the cache priority is nonzero, the driver should support caching by doing the following:

1. Check the FST ID number on the GS/OS direct page. If it is negative (bit 15 = 1; unsigned value = $8000 or greater), then the block is always to be read from the device and not cached. This case is used by FSTs to verify the identity of an on-line volume for which deferred blocks have been written to the cache.
2. Search for the block in the cache by calling CACHE_FIND_BLK.
3. If the block is not in the cache:
   a. Call CACHE_ADD_BLK to add a block of the proper size to the cache.
   b. If the block is granted, read the data from disk and then write it to both the caller's buffer and the cached block. If the block is not granted, just read the data from disk and write it into the caller's buffer.
   c. Skip to step 5.

4. If the block is already in the cache, call MOVE_INFO to transfer the cached block to the caller's buffer.
5. Check for a disk-switched condition; if it is true, then call SET_DISKSW to delete the blocks from the cache and return a disk-switched error from this call. If it is false, the read has been completed successfully.

If the driver must perform multiblock reads to satisfy the request count for the call, it can repeat this loop as many times as needed, or it may be faster to disable caching until all the blocks have been read from the device and then to transfer those blocks to the cache.

On a Write call

When the driver receives control, its direct-page parameters have already been set up by the caller (Device Manager or FST); see the description of Driver_Write in Chapter 10. If the cache priority is nonzero, the driver should support caching by doing the following:

1. Search for the block in the cache by calling CACHE_FIND_BLK.
2. If the block is not in the cache:
   a. Call CACHE_ADD_BLK to add a block of the proper size to the cache.
   b. If the block is granted, continue; otherwise, skip to step 4.
3. Call MOVE_INFO to move data from the caller's buffer to the cached block.
4. Check for a disk-switched condition; if it is true, then call SET_DISKSW to delete the blocks from the cache and return an error from this call.
5. Check the cache priority on the GS/OS direct page. If it is negative (that is, if bit 15 is equal to 1, indicating that the value is $8000 or greater), a deferred-write session is in progress. Your driver should write the block to the cache (if a cached block is available) but not write the data to the device, since the EndSession call that terminates the deferred-write session flushes the cache to disk. This completes the driver's write task.

   If the cache priority is positive, write the block to disk. This completes the driver's write task.

If the driver must perform multiblock writes to satisfy the request count for the call, it can repeat this loop as many times as needed, or it may be faster to disable caching until all the blocks have been written to the device and then to transfer those blocks to the cache. There are, however, several things to be aware of when performing multiblock caching, covered in the folllowing paragraphs.

Multiblock caching

Caching data blocks from a block driver that accepts multiblock calls can be tricky. A great deal of attention must be paid to what is in the cache, what gets updated, and when. Another driver responsibility is to verify that each block is the proper size, for example, that an Apple 3.5 drive is using 512-byte blocks. Some drives that use formatting contain tag bytes in each block, making the block 532 bytes long. When a multiblock request is received, great attention must be paid to where the data goes. The 513th byte of the request must go to block 2—not to the tag bytes—while 532 bytes (the 20 tag bytes must still be accounted for) are written to each block.

All Apple block drivers use the following method of cache-related reads and writes.

When a read call is issued to the driver, the driver checks the cache to see if the requested blocks are there. If the call is requesting more than one block, it starts the check with the first block of the request. If that block is in the cache, the driver copies the data in the block to the read buffer at the location specified in the request. The driver then checks each consecutive block and looks for a match in the cache. If it finds a block that is not cached, the driver then, starting with the first block not in the cache, reads the remaining data from the drive. Once the data is read, the driver picks up where it left off checking the cache. Every time a block is found that is in the cache, the data read from the drive is updated with the data that is in the cache.

Write calls are handled slightly differently, but they are simpler. When a write call that is not a deferred write is received by the driver, the entire transaction is written to the disk. Following that, the driver checks the cache to see if any of the blocks that were just written to the device are also contained in the cache. Every time it finds a block that is cached, the driver updates the cached block with the data that was just written to the device.

Write-deferred calls are also fairly simple. The driver, starting with the first block of the request, checks to see if the block is in the cache. If it is there, the data in the cache is updated. If the block is not in the cache, it is added.

Caching notes

Here are a few other points to keep in mind when designing a driver to support caching.

- *Device calls:* The GS/OS device calls DRead and DWrite do not invoke caching, whether or not the accessed device driver supports it. The Device Manager always sets the cache priority to 0 for those calls.
- *AppleDisk 5.25 driver:* Because it cannot detect disk-switched errors with complete reliability, the AppleDisk 5.25 driver does not support caching. Any block driver with similar limitations should not support caching.

Chapter 10 **GS/OS Driver Call Reference**

This chapter documents the GS/OS driver calls: low-level calls, through the device dispatcher, by which file system translators, the Device Manager, and other parts of GS/OS communicate with device drivers and devices.

The chapter also documents supervisory-driver calls: calls that GS/OS and certain types of device drivers make to supervisory drivers to access supervisor-controlled devices.

About driver calls

All GS/OS device drivers must accept a standard set of calls. These driver calls are of two basic types: internal calls, made by GS/OS to drivers for housekeeping purposes; and device-access calls, low-level translations of application-level calls. The application-level calls that are translated to driver calls include device calls (made through the Device Manager) and all application-level calls that access files (made through an FST).

Both types of calls are described in this chapter. The driver calls that are internal are not like GS/OS calls described elsewhere; the driver calls that access devices, however, are very similar in content and purpose (if not form) to the device calls documented in Chapter 1 of this reference.

Table 10-1 lists the driver calls every GS/OS device driver must accept.

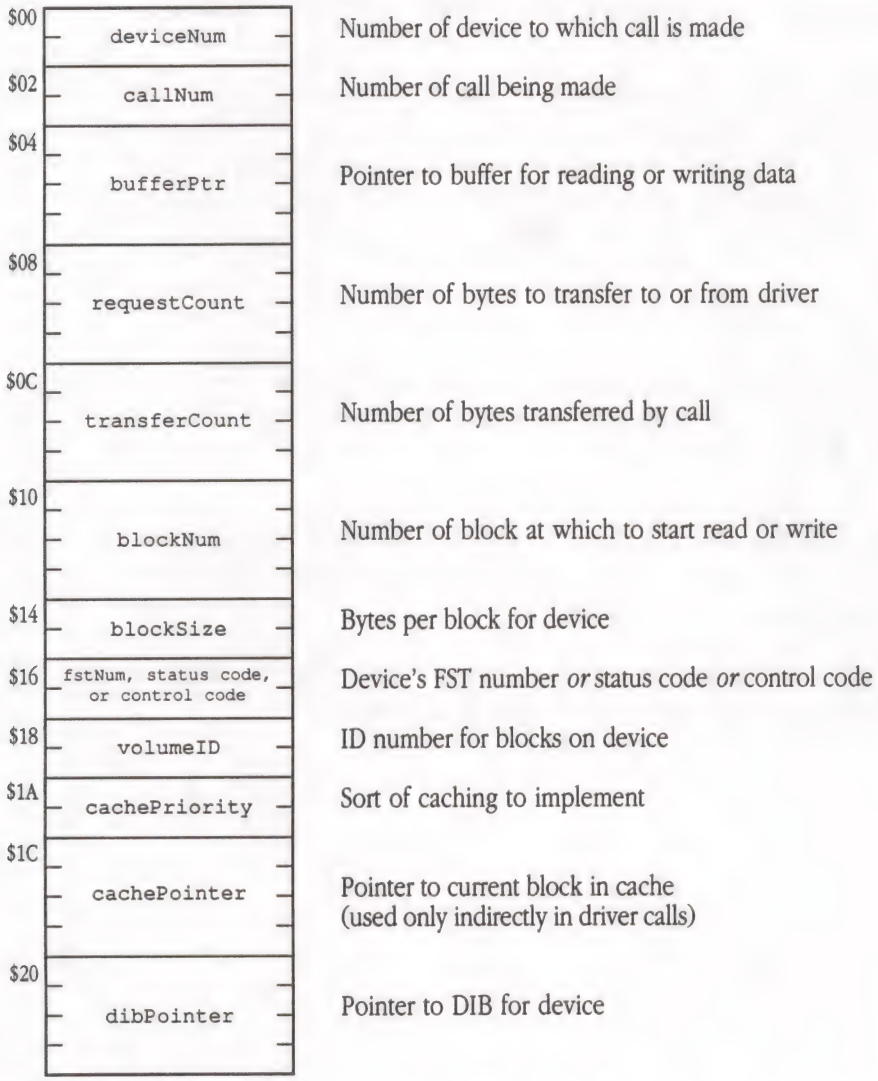
■ **Table 10-1** GS/OS driver calls

Number	Name	Description
$0000	Driver_Startup	Prepares a device for all other device-related calls. This call is issued by the device dispatcher as drivers are loaded or generated.
$0001	Driver_Open	Prepares a character device for conducting I/O transactions.
$0002	Driver_Read	Reads data from a character device or a block device.
$0003	Driver_Write	Writes data to a character device or a block device.
$0004	Driver_Close	Resets a character device driver to its nonopen state.
$0005	Driver_Status	Gets information about the status of a specific device.
$0006	Driver_Control	Sends control information or requests to a specific device.
$0007	Driver_Flush	Writes out any characters in a character device driver's buffer in preparation for purging a driver.
$0008	Driver_Shutdown	Prepares a device driver to be purged (removed from memory).

Recall from Chapter 8 of this reference that GS/OS recognizes both device drivers and supervisory drivers. Supervisory drivers handle a different set of calls from those listed in Table 10-1; see "About Supervisory-Driver Calls," later in this chapter.

All driver calls take their parameters from a parameter block on the GS/OS direct page. Figure 10-1 is a diagram of that parameter block. All driver calls use the same memory locations.

■ **Figure 10-1** Direct-page parameter space for driver calls



Drivers receive calls through a JSL to the driver's main entry point (defined by the driver in its DIB), with the call number in the accumulator and other registers as specified under "Dispatching to Device Drivers" in Chapter 8.

The following sections describe the individual calls. Each call description repeats the direct-page diagram, showing the following features:

- **Offset (direct page):** The width of the direct-page parameter block diagram represents 1 byte; successive tick marks down the side of the block represent successive bytes in memory. Hexadecimal numbers down the left side of the parameter block represent byte offsets from the base address of the GS/OS direct page.
- **Name:** The name of each parameter appears at the parameter's location within the parameter block.
- **Size and type:** Each parameter that is used in a particular call is also identified by size (word or longword) and type (input or result, and value or pointer). A word is 2 bytes; a longword is 4 bytes. An input is a parameter passed from GS/OS to the driver; a result is a parameter returned to GS/OS by the driver. 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.

◆ *Note:* The only result that can be returned from any driver call is `transferCount`. That is, drivers are not permitted to permanently alter any value other than `transferCount` on the GS/OS direct page.

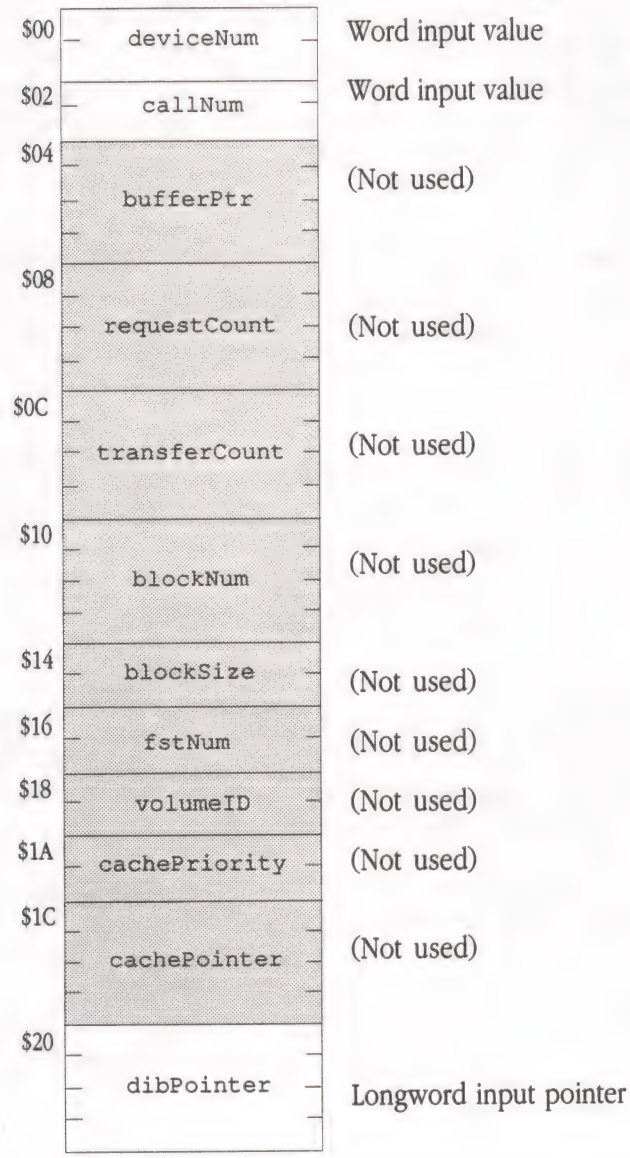
- **Unused parameters:** Although all calls use the same direct-page parameter space, not all parameters are used for every call. For each call description, parameters that are not used are shaded in the parameter-block diagram.

Each parameter used by a call is described in detail following the call's diagram. Additional important notes and call requirements follow the parameter descriptions.

Driver_Startup ($0000)

Description This call performs any tasks necessary to prepare the driver to operate. It is executed by GS/OS during initialization or after loading a driver.

Parameters The Driver_Startup call uses these parts of the direct-page parameter space:



`deviceNum` Word input value: specifies which device is to be accessed by the call. Must be nonzero to be valid.

`callNum` Word input value: specifies the call to be issued. For Driver_Startup, `callNum` = $0000.

`dibPointer` Longword input pointer: points to the device information block for the device being accessed.

Call requirements

Both character device and block device drivers must support this call. For GS/OS, there are 14 slots ($0000–$000F) in the system, only seven of which can be switched in at any one time. To find the slot that your peripheral device is in, start the search at one end of the range and search toward the other end, asking the slot arbiter if the current slot is available. If the slot is not available, the slot arbiter will return an error, and you can continue the search at the next slot number.

▲ **Warning** In GS/OS, you must use the slot arbiter, or you might not find your peripheral if the slot in which the peripheral resides is not currently switched in. ▲

▲ **Warning** Do not use the slot register at $C02D to determine whether the slot is internal or external. Use the bit-encoded slot configuration (BESC) to determine slot assignment. (For more information see the *Apple IIGS Technical Note* #69). ▲

Drivers may use this routine for memory allocation and/or installing an interrupt handler with the GS/OS call BindInterrupt. Character device drivers should maintain an internal flag indicating whether the device is open; that flag should be set to *not open* by this call.

Prior to issuing a startup call to a device, the device dispatcher sets the DIB pointer on the GS/OS direct page.

▲ **Warning** The Driver_Startup call must not be issued by an application. It is for system or device driver use only! ▲

Partitioned devices

Before issuing a startup call, the device dispatcher sets the parameter `dibDevNum` in the device's DIB. This parameter is used by devices that support removable partitioned media. Each partition is accessed as a separate device through its own device driver. Because multiple devices can share a common medium (such as a single CD-ROM disk), it is neccessary to maintain the head links and forward links between devices to reflect disk-switched and off-line conditions among them.

The device driver is responsible for maintaining these device links; it uses the DIB device number (`dibDevNum`) to initialize the head link and forward link in the DIB.

Device numbers can change during the startup process. The boot device driver—always device 1—is replaced by a loaded driver if the slot and unit number of the loaded driver's DIB match those of the boot device. If that happens, the loaded driver's device number (in its DIB) is changed to 1, *but only after startup has been completed.* Therefore, a driver cannot rely on the device number in its DIB to be correct during the startup call. On the second device access (that is, the first call after startup), the driver has another chance to inspect its DIB and note the correct device number.

The driver should examine the head and forward links on the first non-startup call. If the device number does not match the `dibDevNum`, the driver should reestablish the links.

Notes

A driver's DIB is not considered to contain valid information until the successful completion of this call. If a driver returns an error as the result of the startup call, it is not installed in the device list. If the driver returns no error during startup, it then becomes available for an application to access without further initialization (except that a character device requires an open call before use).

There are two possible ways to build a DIB, as follows:

1. Preconstruct the device links, so that each pointer points to the next DIB, and the last pointer is NIL.
2. Allow the device links to be constructed at startup time by taking the following steps:
    a. Set the auxiliary type of the driver file to 3F.
    b. Determine the number of devices.
    c. Allocate the memory for the DIBs.
    d. Establish the links between the DIBs by the link pointer.

Remember that if your driver is active (see "Driver File Types and Auxiliary Types" in Chapter 8) and in the subdirectory *:SYSTEM:DRIVERS on the boot disk, GS/OS always loads it and starts it.

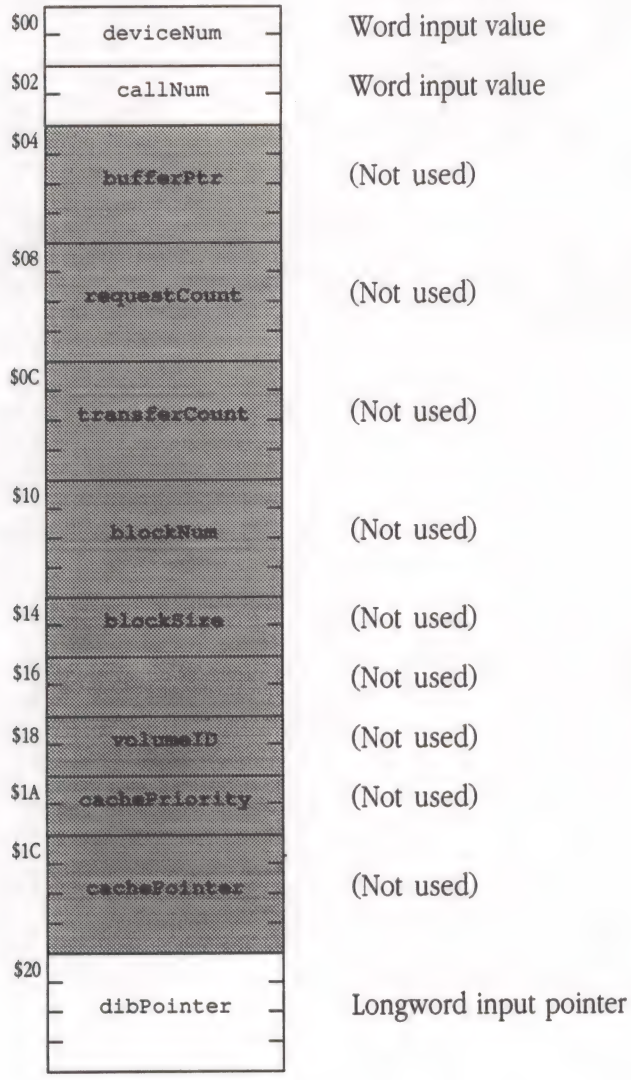
Multiple startup calls to a driver are not permitted. Your driver needn't worry about guarding against them.

If you wish to have separate code for booting and restarting, you can examine the flag at $E1/10D0 to determine which start has occurred. (See "Device Information Block" in Chapter 8, "GS/OS Device Driver Design," for a description of this bit.)

Driver_Open ($0001)

Description This call prepares a character device driver for Read and Write calls. This call is supported by character device drivers only.

Parameters The Driver_Open call uses these parts of the direct-page parameter space:

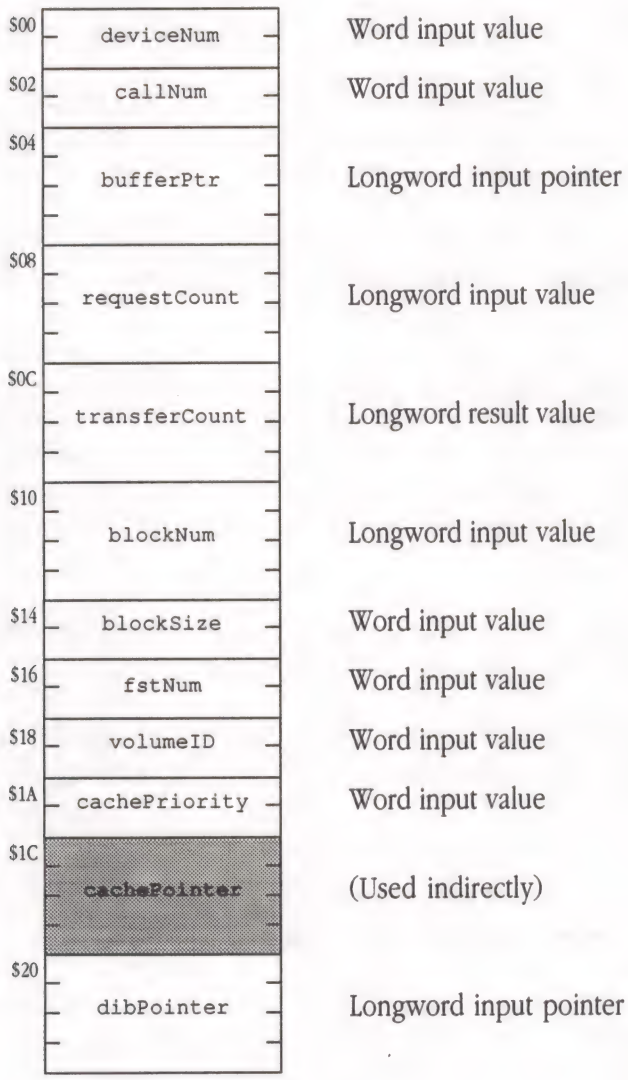


`deviceNum`	Word input value: specifies which device is to be accessed by the call. Must be nonzero.
`callNum`	Word input value: specifies the call to be issued. For Driver_Open, `callNum` = $0001.
`dibPointer`	Longword input pointer: points to the device information block for the device being accessed.
Character device requirements	The driver should maintain a flag indicating whether or not the device is open. This flag should be set to *open* by this call. If the call is issued to a device that is already open, the driver should return a DRVR_PRIOR_OPEN error.
Block device requirements	Block device drivers should take no action on this call and return with no error.
Notes	A driver can use this call to perform whatever tasks are necessary to prepare it for conducting I/O, including allocation of buffers from the Memory Manager.

Driver_Read ($0002)

Description This call transfers data from the device to the buffer specified in the parameter block on direct page. It is supported by both character and block device drivers.

Parameters The Driver_Read call uses these parts of the direct-page parameter space:



`deviceNum` Word input value: specifies which device is to be accessed by the call. Must be a nonzero value.

`callNum` Word input value: specifies the call to be issued. For Driver_Read, `callNum` = $0002.

`bufferPtr` Longword input pointer: points to memory to which the data is to be written after being read from the device.

`requestCount` Longword input value: specifies the number of bytes that the driver is to transfer from the device to the buffer specified by `bufferPtr`.

`transferCount` Longword result value: indicates the number of bytes actually transferred.

`blockNum` Longword input value: specifies the logical address within the block device from which data is to be transferred. This parameter has no application in character device drivers.

`blockSize` Word input value: specifies the size, in bytes, of the block addressed by the block number. This parameter must be nonzero for block devices, zero for character devices.

`fstNum` Word input value: specifies the file system translator that owns the volume from which the block is being transferred. When set, the most significant bit of the FST number forces device access during the read even if the block being accessed is in the cache. In this case no cache access occurs. This parameter has no application in character device drivers.

`volumeID` Word input value: a volume reference number used to identify deferred cached blocks belonging to a specific volume.

`cachePriority` Word input value: specifies whether caching is to be invoked for the block specified in the current I/O transaction, according to this formula:

Priority	Action
$0000	Do not read the block from the cache
$0001–$7FFF	Read the block from the cache

Read operations do not invoke deferred caching; cache priorities are therefore limited to the range from $0000 to $7FFF for this call. Caching is described in more detail in Chapter 9, "Cache Control."

This parameter has no application in character device drivers.

`(cachePointer)`
: Longword pointer: points to the cached equivalent of the disk block requested. Block device drivers that support caching fill in and use this parameter when reading blocks. However, it is neither an input to nor a result from the call; it is set by the system service calls CACHE_FIND_BLK and CACHE_ADD_BLK. See Chapter 11, "System Service Calls," for details.

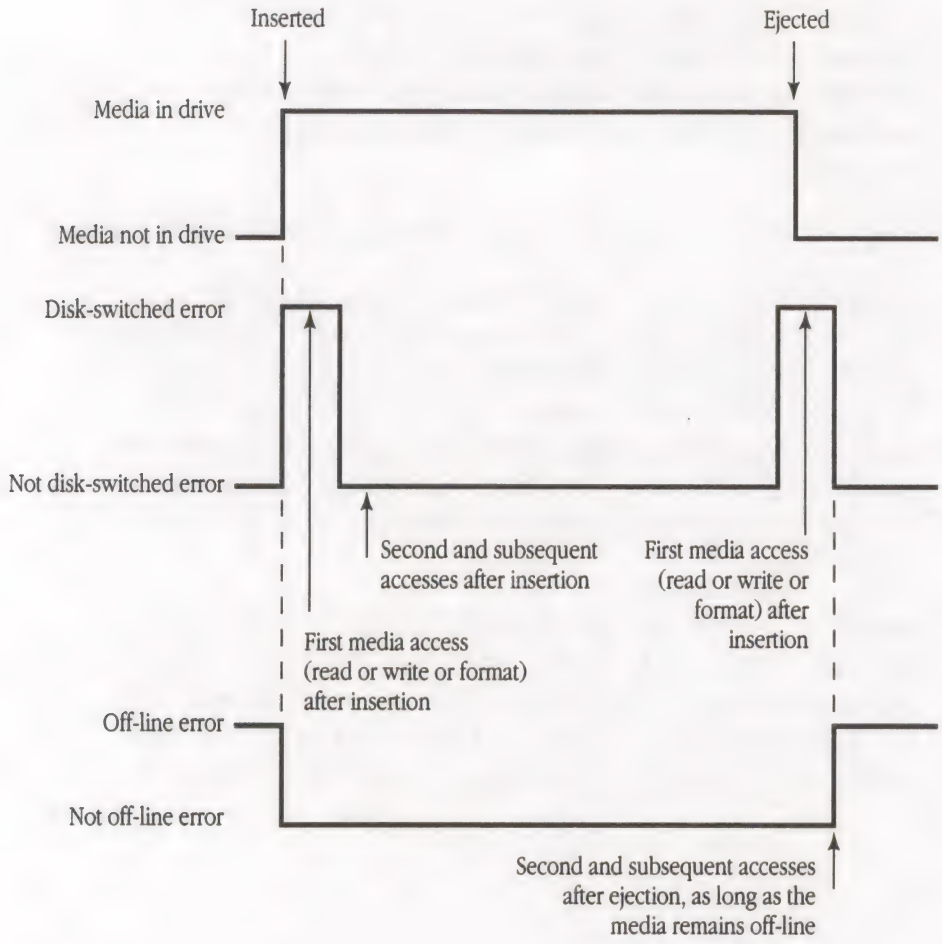
`dibPointer`
: Longword input pointer: points to the DIB for the device being accessed.

Character device requirements

A character device must be open before accepting any I/O transaction requests. If a Driver_Read or DRead is attempted with a device that has not been opened, the driver should return error $23 (device not open). A driver must set the transfer count to 0 before dispatching to a device driver. The driver should then increment the transfer count to reflect the number of bytes received from the device. Typically, a device driver does this by incrementing the transfer count by the block size as each block is read.

The driver should return a disk-switched error on both disk ejection and disk insertion, but only for the first read, write, or format call following the ejection or insertion. The driver should return an off-line error on the second and subsequent read, write, or format calls as long as the media remains off line. Both of these conditions are illustrated in Figure 10-2.

■ **Figure 10-2** Disk-switched and off-line errors



Block device drivers should support caching. How drivers make the calls needed to implement caching is described in Chapter 9, "Cache Control." The calls themselves are described in Chapter 11, "System Service Calls."

Notes

If the request count is greater than the size of a single block, the driver should continue to read contiguous blocks until the request count is satisfied. The driver should validate each block number prior to accessing the device. If at any time during a multiple-block read a bad block number is encountered, the driver should exit with error $2D (invalid block number) and with the transfer count indicating the total number of bytes that were successfully read from the device.

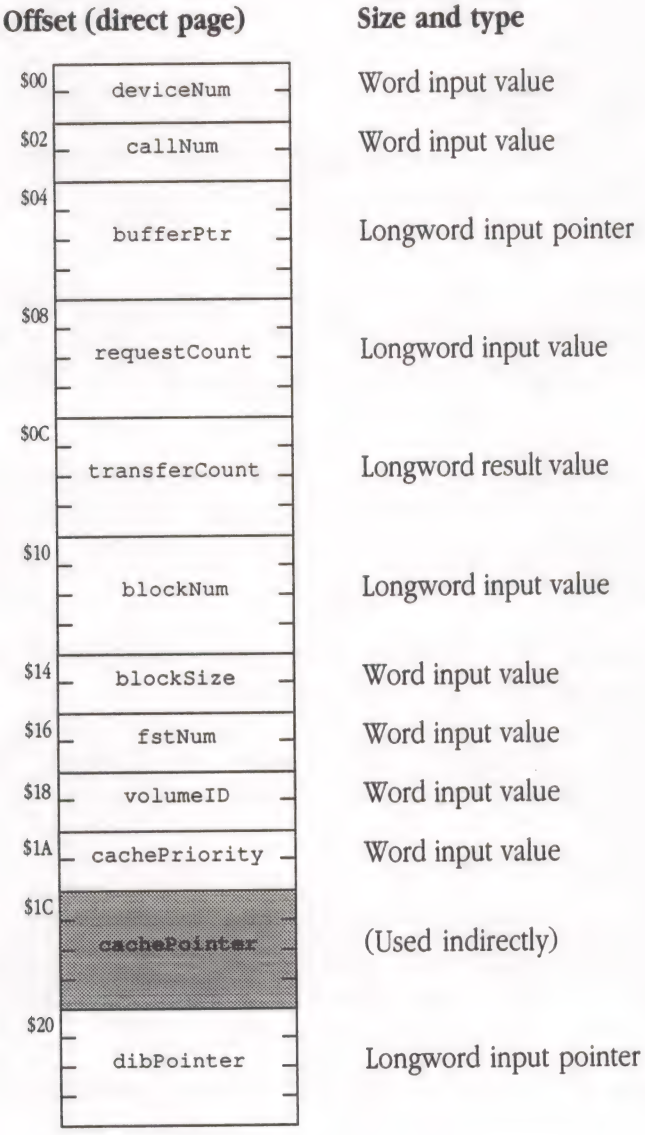
Driver_Write ($0003)

Description

This call transfers data to the device from the buffer specified in the parameter block on direct page. It is supported by both character and block device drivers.

Parameters

The Driver_Write call uses these parts of the direct-page parameter space:



`deviceNum` Word input value: specifies which device is to be accessed by the call. This parameter must be a nonzero value.

`callNum` Word input value: specifies the call to be issued. For Driver_Write, `callNum` = $0003.

`bufferPtr` Longword input pointer: points to memory to which the data is to be written after being read from the device.

`requestCount` Longword input value: specifies the number of bytes that the driver is being requested to transfer from the device to the buffer specified by buffer pointer.

`transferCount` Longword result value: indicates the number of bytes actually transferred.

`blockNum` Longword input value: specifies the logical address within the block device from which data is to be transferred. This parameter has no application in character device drivers.

`blockSize` Word input value: specifies the size in bytes of the block addressed by the block number. This parameter must be a nonzero value for block devices. For character devices, this parameter must be set to a value of zero.

`fstNum` Word input value: specifies the file system translator that owns the volume from which the block is being transferred. The most significant bit of the FST number has no effect on a write call. This parameter has no application in character device drivers.

`volumeID` Word input value: a volume reference number used to identify deferred cached blocks belonging to a specific volume.

`cachePriority` Word input value: specifies whether caching is to be invoked for the block specified in the current I/O transaction, according to this formula:

Priority	Action
$0000	Do not place the block in the cache.
$0001–$7FFF	Place the block in the cache. If no space is available in the cache, purge the least recently used purgeable block to make room for this one.
$8000–$FFFF	Cache the block as a deferred unpurgeable block.

Nondeferred blocks are cached by device number, whereas deferred blocks are cached by volume ID. Caching is described in more detail in Chapter 9, "Cache Control."

This parameter has no application in character device drivers.

`(cachePointer)`
: Longword pointer: points to the cached equivalent of the disk block requested. Block device drivers that support caching fill in and use this parameter when writing blocks. However, it is neither an input to nor a result from the call but is set by the system service calls CACHE_FIND_BLK and CACHE_ADD_BLK. See Chapter 11, "System Service Calls," for details.

`dibPointer`
: Longword input pointer: points to the DIB for the device being accessed.

Character device requirements
: A character device must be open before accepting any I/O transaction requests. If a Driver_Write or DWrite is attempted with a device that has not been opened, the driver should return error $23 (device not open). A driver must increment the transfer count as each byte is written to the device. The driver terminates the I/O transaction when the transfer count equals the request count.

Block device requirements
: A block device does not have to be opened to accept I/O transaction requests.

Prior to accessing any device, the driver should validate that the request count is an integral multiple of the block size; if it is not, the driver should return error $2C (invalid byte count). If the block number is not a valid block number, the driver should exit and return error $2D (invalid block number).

The device dispatcher sets the transfer count to 0 before dispatching to the device driver. The driver should then increment the transfer count to reflect the number of bytes written to the device. Typically, a device driver does this by incrementing the transfer count by the block size as each block is written.

The driver should return a disk-switched error on both disk ejection and disk insertion, but only for the first read, write, or format call following the ejection or insertion. The driver should return an off-line error on the second and subsequent read, write, or format calls as long as the media remains off line. Both of these conditions are illustrated in Figure 10-2 in the Driver_Read call earlier in this chapter.

Block device drivers should support caching. How drivers make the calls needed to implement caching is described in Chapter 9, "Cache Control." The calls themselves are described in Chapter 11, "System Service Calls."

Notes

If the request count is greater than the size of a single block, the driver should write contiguous blocks until the request count is satisfied. The driver should validate each block number prior to accessing the device. If at any time during a multiple-block write a bad block number is encountered, the driver should exit with error $2D (invalid block number), and with the transfer count indicating the total number of bytes that were successfully written to the device.

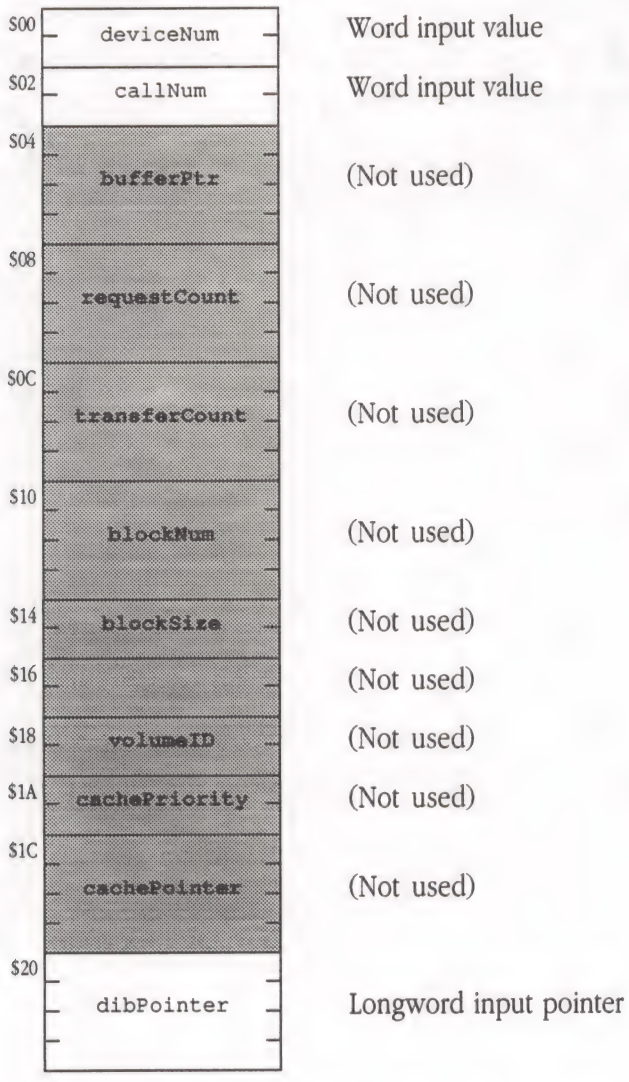
Driver_Close ($0004)

Description

This call sets a character device driver to its closed state, making it unavailable for further I/O requests and releasing any resources acquired as a result of the Open call.

Parameters

The Driver_Close call uses these parts of the direct-page parameter space:

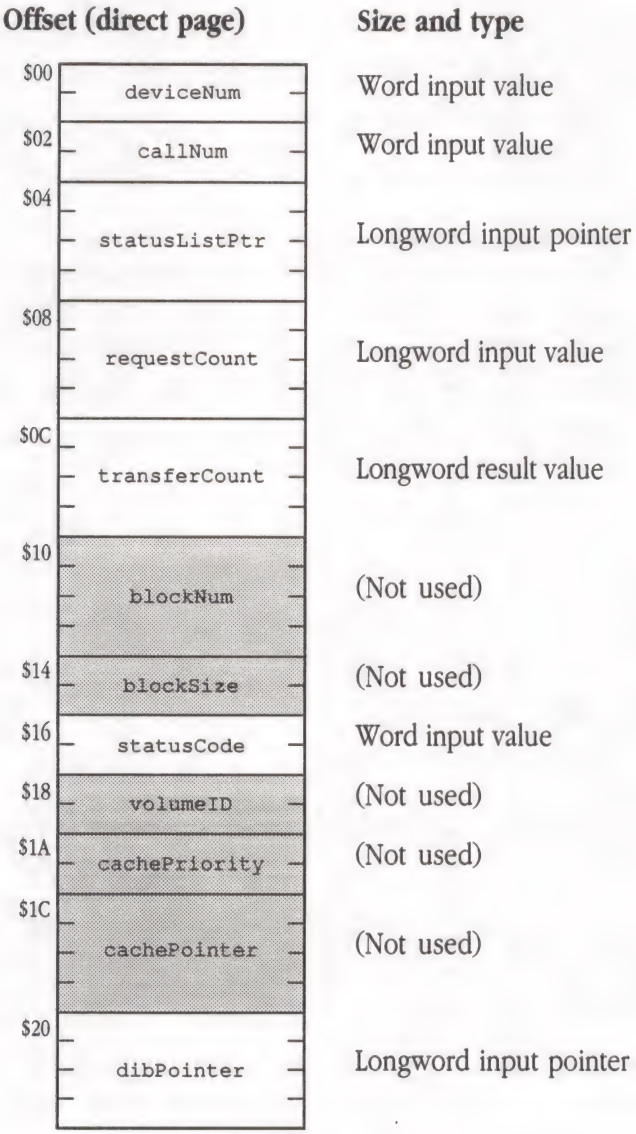


`deviceNum`	Word input value: specifies which device is to be accessed by the call. This parameter must be a nonzero value.
`callNum`	Word input value: specifies the call to be issued. For Driver_Close, `callNum` = $0004.
`dibPointer`	Longword input pointer: points to the DIB for the device being accessed.
Character device requirements	The driver should maintain a flag indicating whether the device is open. This flag should be set to *closed* by this call. If this call is issued to a device that is not open, the driver should return error $23 (device not open). If the driver's open call allocated any memory for buffers, this call should release it back to the Memory Manager.
Block device requirements	This call is supported by character device drivers only; block device drivers should take no action on this call and return with no error.

Driver_Status ($0005)

Description This call obtains current status information from the device or driver. Both standard and device-specific status calls are available.

Parameters The Driver_Status call uses these parts of the direct-page parameter space:



`deviceNum` Word input value: specifies which device is to be accessed by the call. This parameter must be a nonzero value.

`callNum` Word input value: specifies the call to be issued. For Driver_Status, `callNum` = $0005.

`bufferPtr` Longword input pointer: points to a memory buffer into which the status list is to be written. The required minimum size of the buffer is different for different subcalls.

`requestCount` Longword input value: indicates the number of bytes to be transferred. If the request count is smaller than the minimum buffer size required by the call, an error will be returned.

`transferCount` Longword result value: indicates the number of bytes actually transferred.

`statusCode` Word input value: specifies the type of status request. Status codes of $0000 through $7FFF invoke standard status subcalls that must be supported (if not acted upon) by every device driver. Device-specific status subcalls, which may be defined for individual devices, use status codes in the range from $8000 through $FFFF. These are the currently defined status codes and subcalls:

$0000	GetDeviceStatus
$0001	GetConfigParameters
$0002	GetWaitStatus
$0003	GetFormatOptions
$0004	GetPartitionMap
$0005–$7FFF	(Reserved)
$8000–$FFFF	(Device specific)

`dibPointer` Longword input pointer: points to the DIB for the device being accessed.

Notes The device driver is responsible for validating the status code prior to executing the requested status call. If an invalid status code is passed to the driver, the driver should return error $21 (invalid status code).

The device dispatcher sets the transfer count to 0 before calling the device driver. If the call is successful, the device driver should set the transfer count to the number of bytes returned.

- *Note:* Both standard and device-specific status subcalls may detect an off-line or disk-switched status. If either of these conditions occurs, the driver should make the system service call SET_DISKSW to notify the device dispatcher, which maintains the system disk-switched error state. A disk-switched or off-line status should not be returned as an error from a status call; drivers should return errors only when a call fails.

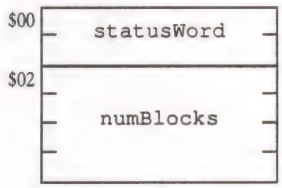
  Any status call that detects on-line and disk-switched conditions should update the parameter `blockCount` in the DIB after media insertion.

GetDeviceStatus (Driver_Status subcall)

`statusCode` = $0000

Description This subcall returns, in the status list, a general device status word followed by a longword parameter specifying the number of blocks supported by the device.

Parameters The status list is 6 bytes long. This is its format:

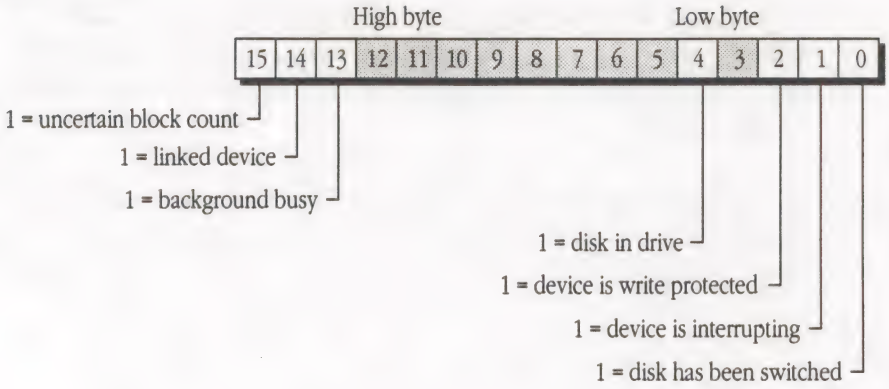


Offset	Size	Description
$00	Word	Status word (see following definition)
$02	Longword	Number of blocks on device

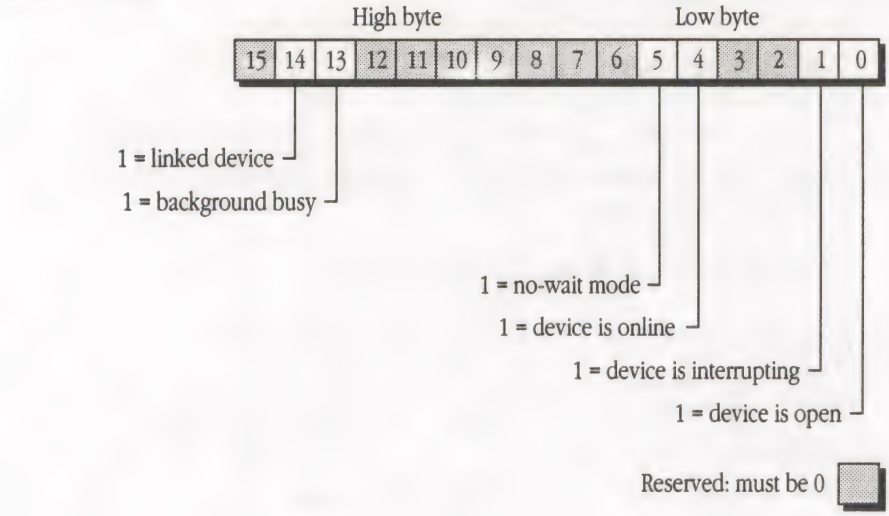
The status word indicates several aspects of the device's status. Character devices and block devices define the status word somewhat differently, as shown in Figure 10-3.

■ **Figure 10-3** Device status word

Block device:



Character device:

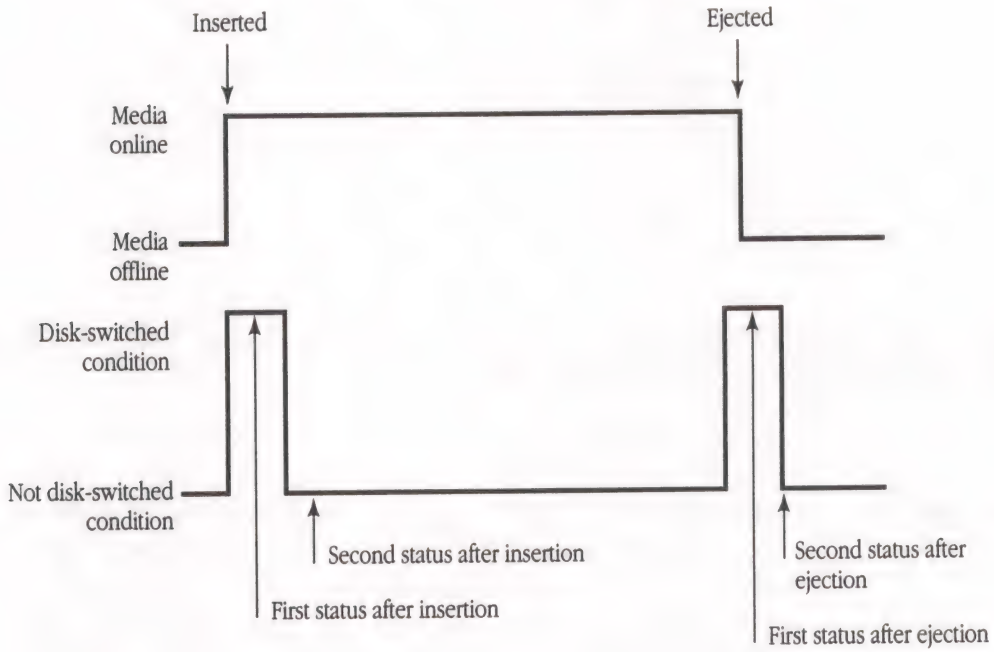


Character device drivers should return a block count of 0.

If the block driver returns either bit 0 as set (= 1) or bit 4 as cleared (= 0), it should also contact the system service call SET_DISKSW. This is because older ProDOS devices supported by the generated drivers do not support disk switch but do support on line; thus, GS/OS treats not on line and disk switch as the same condition.

The status word should show a disk-switched condition (bit 0 = 1) on both disk ejection and disk insertion, but only for the first device access or the first status call following the ejection or insertion. The driver should maintain the status word to show an off-line condition (bit 4 = 0) as long as there is no disk in the drive. Figure 10-4 illustrates the disk-switched condition.

■ **Figure 10-4** Disk-switched condition



◆ *Note:* Error codes should not be returned for conditions indicated with the general status word. A status call should return an error code only if the call fails.

GetConfigParameters (Driver_Status subcall)

`statusCode` = $0001

Description

This subcall returns, in the status list, a length word and a list of configuration parameters. The structure of the configuration list is device dependent. The size of the status list is 2 + `listLength` bytes:

Parameters

Offset	Size	Description
$00 Length word	Word	Length of list (in bytes)
$02 Configuration parameters list	—	Configuration list

GetWaitStatus (Driver_Status subcall)

`statusCode` = $0002

Description

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

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

Parameters	Offset	Size	Description
	$00 waitMode	Word	Wait/no-wait status of device

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

GetFormatOptions (Driver_Status subcall)

`statusCode` = $0003

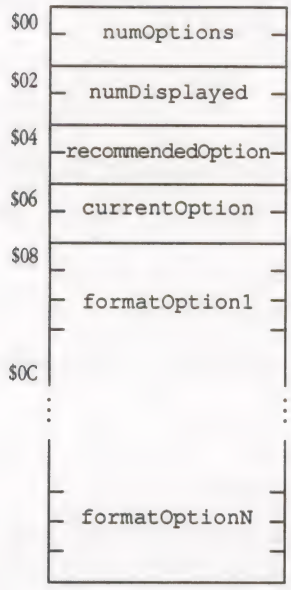
Description

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

This subcall returns the list of formatting options for a particular device. One of the options can then be selected and applied (by an FST, for example) with the Driver_Control subcalls SetFormatOptions followed by FormatDevice. Devices that do not support media variables should return a transfer count of 0 and generate no error. Character devices should do nothing and return no error from this call.

If a device does support media variables, it should return a status list consisting of a four-word header followed by a set of entries, each of which describes a formatting option.

Parameters The status list looks like this:



Size	Description
Word	Number of format option entries in list
Word	Number of options to be displayed
Word	Recommended default formatting option
Word	Option with which current on-line media was formatted
(16 bytes)	First format options entry
(16 bytes)	Last format options entry

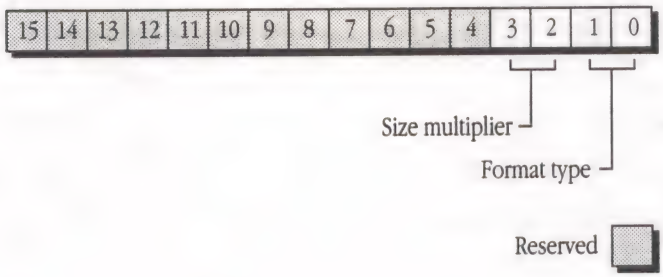
Of the total number of options in the list, one or more may be displayed on the initialization dialog presented to the user when initializing a disk (see the calls Format and EraseDisk in Chapter 10 of *GS/OS Reference*). The options to be displayed are always the first ones in the list. (Undisplayed options are available so that drivers can provide FSTs with logically different options that are actually physically identical and therefore needn't be duplicated in the dialog.)

The value specified in the `currentOption` parameter is the format option of the current on-line media. If a driver can report this parameter, it should do so. If the driver cannot detect the current option, it should indicate ***unknown*** by returning $0000.

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

Offset		Size	Description
$00	formatOptionNum	Word	Number of option
$02	linkRefNum	Word	Number of linked option
$04	flags	Word	(See definition below)
$06	blockCount	Longword	Number of blocks supported by device
$0A	blockSize	Word	Block size in bytes
$0C	interleaveFactor	Word	Interleave factor (in ratio to 1)
$0E	mediaSize	Word	Media size (see flags description)

Bits within the flags word are defined as follows:



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

00 Universal format

01 Apple format

10 Non-Apple format

11 (Not valid)

Size multiplier is used, in conjunction with the parameter `mediaSize`, to calculate the total number of bytes of storage available on the device. Size multiplier can have these binary values and meanings:

00 `mediaSize` is in bytes

01 `mediaSize` is in kilobytes

10 `mediaSize` is in megabytes

11 `mediaSize` is in gigabytes

Character devices should return no error from this call.

Example

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

The entire format options list looks like this:

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

Format option 2:

$0002	Option 2
$0003	LinkRef = option 3
$0005	Apple format/size in kilobytes
$0000 0640	Block count = 1600
$0100	Block size = 256 bytes
$0004	Interleave factor = 4:1
$0190	Media size = 400 KB

Format option 3:

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

GetPartitionMap (Driver_Status subcall)

`statusCode` = $0004

Description This call returns, in the status list, the partition map for a partitioned disk or other medium. See the GetPartitionMap DStatus subcall in Chapter 1 for more information on this subcall.

Device-specific Driver_Status subcalls

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

The content and format of information returned from these subcalls can be defined individually for each type of device. The device dispatcher puts the regular driver-call parameters on the GS/OS direct page, and the device dispatcher and the Device Manager convert the application parameter list from a DStatus call into a GS/OS driver call. The status code must be in the range from $8000 to $FFFF.

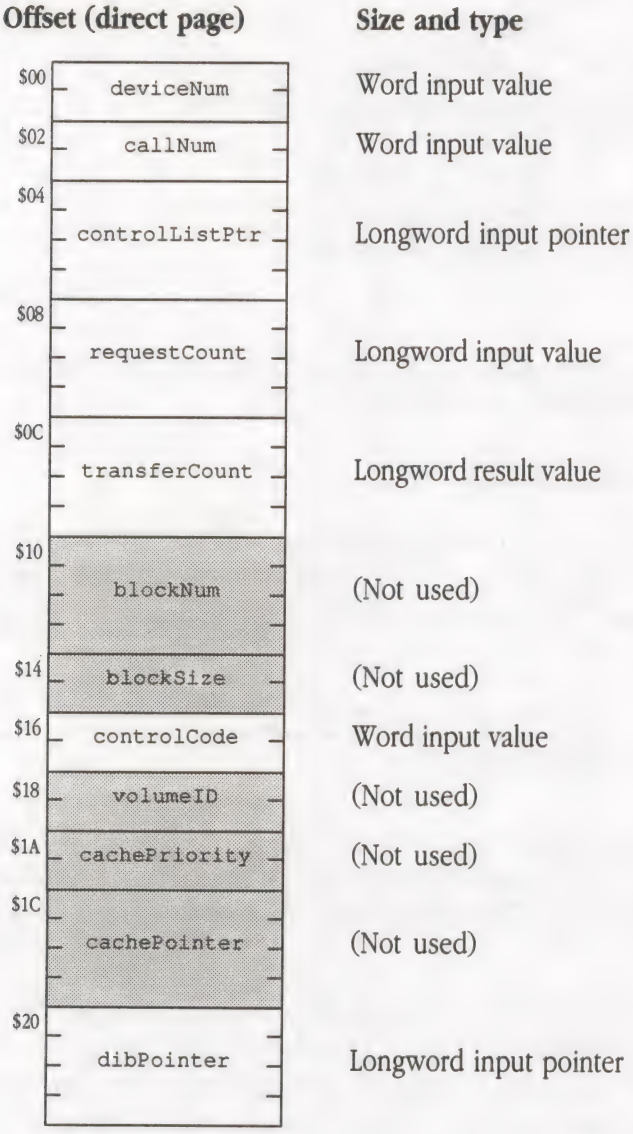
Driver_Control ($0006)

Description

This call sends control information or data to the device or the device driver. Extensions to the standard set of calls are supported through the use of device-specific control codes.

Parameters

The Driver_Control call uses these parts of the direct-page parameter space:



`deviceNum` Word input value: specifies which device is to be accessed by the call. This parameter must be a nonzero value.

`callNum` Word input value: specifies the call to be issued. For Driver_Control, `callNum` = $0006.

`controlListPtr`
Longword input pointer: points to a memory buffer from which the driver reads the control list. The format of the data and the required minimum size of the buffer are different for different subcalls.

`requestCount` Longword input value: indicates the number of bytes to be transferred. If the request count is smaller than the minimum buffer size required by the call, the driver should return an error. For control subcalls that do not use the control list, this parameter is not used.

`transferCount` Longword result value: indicates the number of bytes of information taken from the control list by the device driver.

`controlCode` Word input value: specifies the type of control request. Control codes of $0000 through $7FFF invoke standard control subcalls that must be supported (if not acted upon) by every device driver. Device-specific control subcalls, which may be defined for individual devices, use control codes in the range $8000 through $FFFF. These are the currently defined control codes and subcalls:

$0000	ResetDevice
$0001	FormatDevice
$0002	EjectMedium
$0003	SetConfigParameters
$0004	SetWaitStatus
$0005	SetFormatOptions
$0006	AssignPartitionOwner
$0007	ArmSignal
$0008	DisarmSignal
$0009	SetPartitionMap
$000A–$7FFF	(Reserved)
$8000–$FFFF	(Device specific)

`dibPrinter` Longword input pointer: points to the DIB for the device being accessed.

Notes

The device driver is responsible for validating the control code and control list length prior to executing the requested control call. If an invalid control code is passed to the driver, the driver should return error $21 (invalid control code). If an invalid control list length is passed to the driver, the driver should return error $22 (invalid parameter).

If the call is successful, and if a control list was used, the device driver should set the transfer count to the number of bytes processed. For those subcalls that pass no information in the control list, the driver need not access the control list and verify that its length word is 0; the driver should ignore the control list and request count entirely and pass a transfer count of 0.

ResetDevice (Driver_Control subcall)

`controlCode` = $0000

Description

The ResetDevice subcall sets a device's configuration parameters back to their default values. Every GS/OS device driver contains default configuration settings for each device it controls; see Chapter 8, "GS/OS Device Driver Design," for more information.

ResetDevice also sets a device's format options back to their default values if the device supports media variables. See the description of the SetFormatOptions subcall later in this section for more information.

If successful, this call has a transfer count of 0 and no error is returned. Request count should be ignored; the control list is not used.

FormatDevice (Driver_Control subcall)

`controlCode` = $0001

Description

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

After formatting, FormatDevice resets the device's format options back to their default values if the device supports media variables. See the description of the Driver_Control subcall SetFormatOptions later in this section for more information.

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

If successful, this call has a transfer count of 0. Request count should be ignored; the control list is not used.

EjectMedium (Driver_Control subcall)

`controlCode` = $0002

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

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

If successful, this call has a transfer count of 0. Request count should be ignored; the control list is not used.

SetConfigParameters (Driver_Control subcall)

`controlCode` = $0003

Description The SetConfigParameters subcall sends device-specific configuration parameters to a device. The configuration parameters are contained in the control list. The first word in the control list indicates the length of the configuration list, in bytes. The configuration parameters follow the length word.

Parameters

Offset		Size	Description
$00	length	Word	Length of list (in bytes)
$02	configParamList	—	Configuration list

The structure of the configuration list is device dependent.

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

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

SetWaitStatus (Driver_Control subcall)

`controlCode` = $0004

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

Description

When a device is in wait mode, it does not terminate a read call until it has read the number of characters specified in the request count. In no-wait mode, a read call returns immediately after reading the available characters, up to the maximum specified by `requestCount`, with a transfer count indicating the number of characters returned. If one or more characters is available, the transfer count is nonzero; if no character is available, the transfer count is zero.

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

Parameters

Offset		Size	Description
$00	waitMode	Word	Wait/no-wait status of device

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

SetFormatOptions (Driver_Control subcall)

`controlCode` = $0005

Description

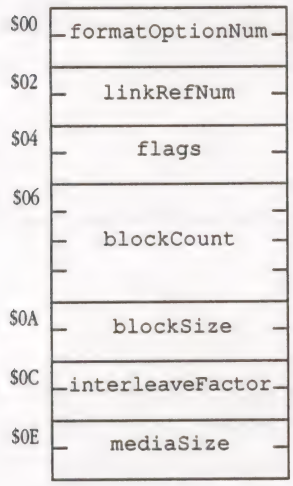
Some block devices can be formatted in more than one way. Formatting parameters can include such variables as file system group, number of blocks, block size, and interleave. Each driver that supports media variables (multiple formatting options) contains a list of the formatting options for its devices.

The SetFormatOptions subcall sets these media-specific formatting parameters prior to the execution of a FormatDevice subcall. SetFormatOptions does not itself cause or require a formatting operation. The control list for SetFormatOptions consists of two word-length parameters:

Parameters

Offset		No.	Size and type
$00	formatOptionNum	Word	Number of format option
$02	interleaveFactor	Word	Override interleave factor (if nonzero)

The format option number (`formatOptionNum`) specifies a particular format option entry from the driver's format options list (returned from the Driver_Status subcall GetFormatOptions). The format option entry has this format:



Offset	Size	Description
$00	Word	Number of option
$02	Word	Number of linked option
$04	Word	File system information
$06	Longword	Number of blocks supported by device
$0A	Word	Block size, in bytes
$0C	Word	Interleave factor (in ratio to 1)
$0E	Word	Media size

See the description of the Driver_Status subcall GetFormatOptions, earlier in this chapter, for a more detailed description of the format options entry.

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

If you want to carry out a formatting process with this subcall and not use the GS/OS format call, your application can take the following steps (if you use the format call, the Initialization Manager takes these steps for you):

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

△ **Important** SetFormatOptions is meant to set the parameters for *one* subsequent formatting operation only. Drivers should expect SetFormatOptions to be called each time a disk is to be formatted with anything other than the recommended (default) option. This implies that, after each successful formatting operation, the driver should revert to the default option. △

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

AssignPartitionOwner (Driver_Control subcall)

`controlCode` = $0006

Description

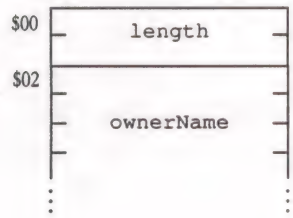
The AssignPartitionOwner subcall provides support for partitioned media on block devices. Each partition on a disk has an owner, identified by a string stored on disk. The owner name identifies the file system to which the partition belongs.

This subcall is executed by an FST after making one of the Driver_Control subcalls EraseDisk and FormatDevice to allow the driver to reassign the partition to the new owner.

Partition owner names can be up to 32 bytes in length. Uppercase and lowercase characters are considered equivalent.

Parameters

The control list for this call consists of a GS/OS string, generated by the FST or other caller, naming the partition owner.



Offset	Size	Description
$00	Word	Length of name (in bytes)
$02	—	Partition owner name

This call does not reassign physical block allocation within a device partition but merely changes the ownership of that partition. The names of the partition owners can be found in "The SCSI Manager" in Chapter 2, "The SCSI Driver."

Block devices with nonpartitioned media and character devices should do nothing with this call and return no error.

ArmSignal (Driver_Control subcall)

`controlCode` = $0007

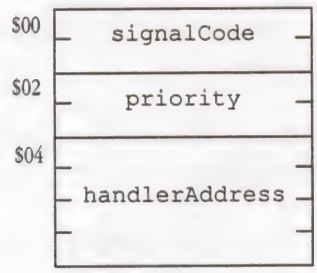
Description

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

An ArmSignal subcall is issued by application programs to inform the driver to call an application-supplied interrupt handler routine at the location specified in `handlerAddress`. The SIGNAL system service call provides the `signalCode` and `priority` values to GS/OS.

Parameters

This is the control list for the subcall:



`signalCode` Contains the ID of the condition that the driver will pass to GS/OS when the condition occurs. The `signalCode` ID is assigned by the caller and must match a unique number defined by the device driver. The only subsequent use of the `signalCode` number is as an input to the DControl subcall DisarmSignal. A device driver should bind only one signal handler to each of its defined `signalCode` numbers.

`priority` The signal priority the caller wishes to assign to the signal condition; $0000 is the lowest priority, and $FFFF is the highest priority.

`handlerAddress`

Entry address of the caller's signal handler. Control is passed to this address when GS/OS dispatches a queued signal associated with an occurrence of the signal condition. See Chapter 9 of the *GS/OS Reference* for a description of the signal handler execution environment.

See the ArmSignal description in Chapter 1, "GS/OS Device Call Reference," for an example of ArmSignal.

DisarmSignal (Driver_Control subcall)

`controlCode` = $0008

Description

The DisarmSignal subcall provides a means for a device driver to remove its signal handler from the GS/OS signal handler list. The `signalCode` parameter is the identification number assigned to that handler when the signal was armed.

Parameters

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

SetPartitionMap (Driver_Control subcall)

`statusCode` = $0009

Description

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

Device-specific Driver_Control subcalls

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

The content and format of information returned from these subcalls can be defined individually for each type of device. The device dispatcher puts the regular driver-call parameters on the GS/OS direct page, and the device dispatcher and the Device Manager convert the application parameter list from a DControl call into a GS/OS driver call. The status code must be in the range from $8000 to $FFFF.

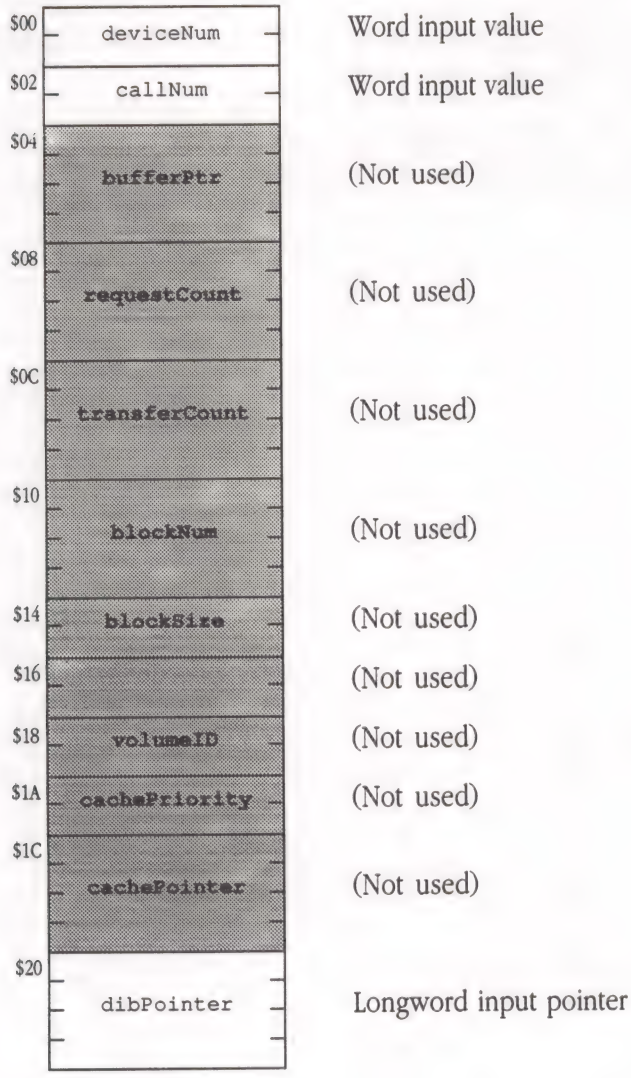
Driver_Flush ($0007)

Description

Driver_Flush is issued only in preparation for a close or shutdown call. A character device that maintains its own buffer should write out any remaining buffer contents.

Parameters

The Driver_Flush call uses these parts of the direct-page parameter space:



`deviceNum` Word input value: specifies which device is to be accessed by the call. This parameter must be a nonzero value.

`callNum` Word input value: specifies the call to be issued. For Driver_Flush, `callNum` = $0007.

`dibPointer` Longword input pointer: points to the DIB for the device being accessed.

Notes

This call is not supported by block device drivers; they should return error $20 (invalid request).

A character device driver that does not maintain its own data buffers need take no action on this call.

Even if the driver is currently set to no-wait mode, the driver must not return until its output buffer is completely flushed.

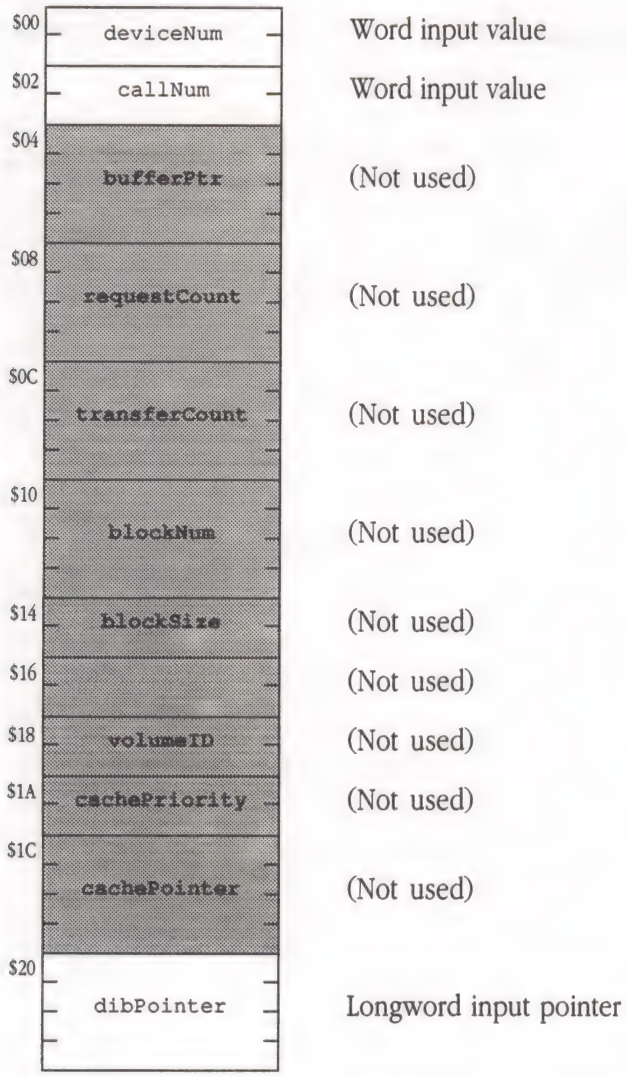
Driver_Shutdown ($0008)

Description

Driver_Shutdown is issued by GS/OS in preparation for removing a driver from memory. The driver executes any necessary operations, such as releasing buffer memory.

Parameters

The Driver_Shutdown call uses these parts of the direct-page parameter space:



`deviceNum` Word input value: specifies which device is to be accessed by the call. This parameter must be a nonzero value.

`callNum` Word input value: specifies the call to be issued. For Driver_Shutdown, `callNum` = $0008.

`dibPointer` Longword input pointer: points to the DIB for the device being accessed.

Notes If Driver_Shutdown is sent to an open character device, the driver should perform the equivalents of a flush and a close call before shutting down.

△ **Important** This call is for system use only. It is not to be issued by an application! △

If more than one device is associated with a single code segment, only the last device to be shut down should return no error. Other devices should return an I/O error to prevent the segment from being purged before the last device is shut down.

About supervisory-driver calls

As explained in Chapter 8, supervisory drivers (or supervisors) are programs that mediate among several types of device drivers, allocating and dispatching their calls and interrupt-handling facilities among several types of hardware devices. Calls to supervisory drivers can be classified according to who makes them and who handles them:

- From a device driver's point of view, there are calls that the device driver can make and calls it cannot (because only other parts of GS/OS can make them).
- From the supervisory driver's point of view, there are calls that the supervisory driver itself must handle and calls that are handled by the supervisor dispatcher and thus never reach the supervisory driver.

If you are writing a device driver that accesses a supervisory driver, you need to know which calls you can make and whether they actually access the supervisory driver. Table 10-2 shows the supervisory-driver calls available to device drivers. If you are writing a supervisory driver, you need to know which calls your driver must accept and whether they come from a device driver. Table 10-3 shows those calls that supervisor drivers must accept.

■ **Table 10-2** Supervisory-driver calls available to device drivers

Call no.	Supervisor no.	Call name	Explanation
$0000	$0000	GetSupervisorNumber	Returns the supervisor number for the supervisory driver with a given supervisor ID
$0001	$0000	Set_SIB_Ptr	Sets the direct-page supervisor information block pointer for a specified supervisory driver
$0002–$FFFF	$0000	—	(Reserved)
$0002–$FFFF	(Nonzero)	(Driver-specific calls)	For use by device drivers

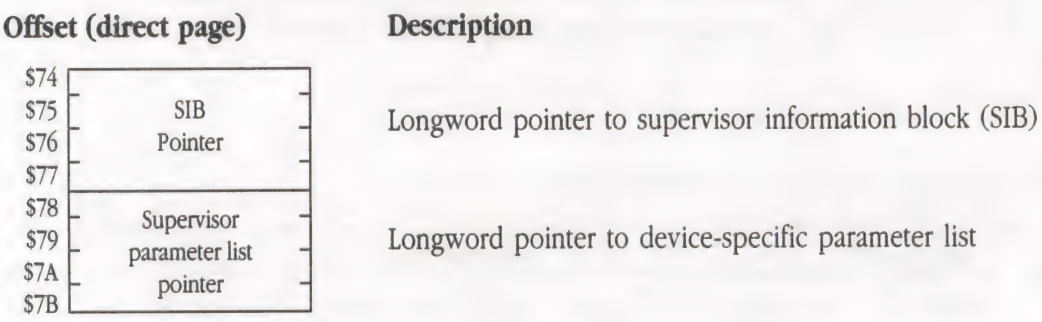
Note that only those calls in Table 10-2 with nonzero supervisor numbers appear also in Table 10-3; they are the only calls in Table 10-2 that are actually handled by supervisory drivers.

■ **Table 10-3** Calls that supervisory drivers must accept

Call no.	Supervisor no.	Call name	Explanation
$0000	(Nonzero)	Supervisor_Startup	Prepares the supervisory driver to receive calls from device drivers
$0001	(Nonzero)	Supervisor_Shutdown	Releases any system resources allocated at startup
$0002–$FFFF	(Nonzero)	(Driver-specific calls)	For use by device drivers

A device driver or other program makes a call to a supervisory driver by making the system service call SUP_DRVR_DISP (see Chapter 11). Parameters for supervisory-driver calls are passed both in registers and in locations $74–$7B on the GS/OS direct page, called the supervisor direct page (Figure 10-5).

■ **Figure 10-5** Supervisor direct page: parameter space



On input to the supervisory driver, the A register (accumulator) contains the supervisor number, which specifies the supervisory driver to which the call is directed; the X register contains the call number. On return from the call, the A register contains the error code (zero if no error). Other registers have call-specific functions.

The supervisor number in the A register is a required input to all supervisory-driver calls. Calls with a supervisor number of zero (see Table 10-2) are handled by the supervisor dispatcher; calls with a nonzero supervisor number (see Table 10-3) are handled by supervisory drivers.

The rest of this chapter documents the currently defined supervisory-driver calls.

GetSupervisorNumber ($0000)

Description

When a device driver is started up, it makes this call to get the supervisor number (the position in the supervisor list) of its supervisory driver. The device driver needs that number for subsequent access to its supervisory driver.

The device driver passes the **supervisor ID** (a numerical indication of general supervisor type, such as "SCC" or "SCSI") of its supervisory driver to this call; the call then returns the supervisor number in the X register.

The call requires an input supervisor number of zero; if the input supervisor number is nonzero, this call becomes the call Supervisor_Startup, described next.

Parameters

Input:

A register = $0000 (on input, `supervisorNum` = 0)
X register = $0000 (`callNum`)
Y register = `supervisorID`

Output:

A register = error code
X register = `supervisorNum`
Supervisor direct page = `sibPtr`

`callNum` Word input value: This X-register input specifies which type of call is to be issued to the supervisory driver. It is 0 for this call.

`supervisorID` Word input value: This Y-register input specifies the general type of supervisor ID whose supervisor number is sought. These are the supervisor IDs currently defined by Apple Developer Technical Support:

$0001	SCC supervisory driver
$0002	SCSI supervisory driver
$0003–$FFFF	(Reserved)

`supervisorNum` This parameter appears twice in this call:

Word input value: This A-register input must be 0 for this call.

Word result value: This X-register result is the supervisor number of the supervisory driver whose supervisor ID was passed as input.

`sibPtr` Longword result pointer: This result on the supervisor direct page points to the supervisor information block (SIB) for the supervisory driver being accessed. It is a side benefit of the call; the supervisor dispatcher places the supervisory driver's SIB on the supervisor direct page before returning to the caller.

Notes This call is handled by the supervisor dispatcher; it does not result in any execution of the supervisory driver itself.

Error handling If the supervisor dispatcher cannot find a supervisory driver with the input supervisor ID, error $28 (device not connected) is returned. In such a case the device driver will not be able to use the supervisory driver and should return an error from its startup call.

Supervisor_Startup ($0000)

Description This call is responsible for preparing the supervisory driver for use by device drivers. Any system resources required by the supervisory driver, such as memory, should be allocated during this call. If the supervisory driver cannot allocate sufficient resources to support device driver calls, then it should return an error; if it returns an error as a result of the startup call, it is removed from the supervisor list.

This call requires that the supervisor number be nonzero.

Parameters

Input:
Contents of supervisor direct page (`sibPtr`) *plus*
A register = `supervisorNum`
X register = `callNum` ($0000)

Output:
A register = error code

`callNum` Word input value: This X-register input specifies which type of call is to be issued to the supervisory driver. It is 0 for this call.

`supervisorNum` Word input value: This A-register input specifies which supervisory driver is to be started. It must be nonzero for this call.

`sibPtr` Longword input pointer: This supervisor direct-page input is the address of the supervisor information block for the supervisory driver being started up. This parameter is set up by the supervisor dispatcher, in case the supervisory driver needs it.

Notes GS/OS starts up supervisory drivers before starting up any device drivers, so that the supervisor is available to the device driver at startup time.

△ **Important** This call is for system use only. It is not to be issued by a device driver. △

Set_SIB_Pointer ($0001)

Description

This call sets the parameter `sibPtr` on the supervisor direct page to the proper value for the specified supervisory driver.

This call requires that the input supervisor number be zero. If the input supervisor number is nonzero, this call becomes the call Supervisor_Shutdown, described next.

Parameters

Input:
A register = `supervisorNum` ($0000)
X register = `callNum` ($0001)
Y register = `supervisorNum`

Output:
Contents of supervisor direct page (`sibPtr`) *plus*
A register = error code

`callNum` Input word value: This X-register input specifies which type of call is to be issued to the supervisory driver. It is $0001 for this call.

`supervisorNum` (A register) Word input value: This A-register input must be 0 for this call, which directs the call to the supervisor dispatcher.

`supervisorNum` (Y register) Word input value: This Y-register input specifies the supervisor number of the supervisory driver whose SIB pointer is to be placed on the supervisor direct page.

`sibPtr` Longword result pointer: This supervisor direct-page result points to the supervisor information block for the supervisory driver specified.

Notes

This call is handled by the supervisor dispatcher; it does not result in any execution of the supervisory driver itself.

Supervisor_Shutdown ($0001)

Description This call is responsible for releasing any system resources acquired during startup of the supervisory driver.

This call requires that the input supervisor number be nonzero.

Parameters *Input:*
Contents of supervisor direct page (`sibPtr`) *plus*
A register = `supervisorNum`
X register = `callNum` ($0001)

Output:
A register = error code

`callNum` Word input value: This X-register input specifies which type of call is to be issued to the supervisory driver. It is $0001 for this call.

`supervisorNum` Word input value: This A-register input specifies which supervisory driver is to be shut down. It must be nonzero for this call.

`sibPtr` Longword input pointer: This supervisor direct-page input points to the supervisor information block for the supervisory driver being accessed. This parameter is set up by the supervisor dispatcher in case the supervisory driver needs it.

Notes GS/OS shuts down supervisory drivers only after shutting down all device drivers.

△ **Important** This call is for system use only. It is not to be issued by a device driver. △

Driver-specific calls ($0002–$FFFF)

Description These calls are used by device drivers to request specific tasks from their supervisory drivers. The nature of those tasks is device specific.

Parameters

Input:
Contents of GS/OS direct page including supervisor direct page, *plus*
A register = `supervisorNum`
X register = `callNum` ($0002–$000F)

Output:
Contents of GS/OS direct page *plus*
A register = error code

`callNum` Word input value: This X-register input specifies which type of call is to be issued to the supervisory driver. It must be in the range $0002 through $000F for this call.

`supervisorNum` Word input value: This A-register input value specifies which supervisory driver is to be called. It must be nonzero for this call.

`sibPtr` Longword input pointer: This supervisor direct-page input points to the SIB for the supervisory driver being accessed. This parameter is set up by the supervisor dispatcher in case the supervisory driver needs it.

Driver error codes

GS/OS can recognize the device driver error codes listed in Table 10-4. Any device driver or supervisor driver you write should be able to return all appropriate errors from this list. Also please note the following requirements:

- All block device drivers must support disk-switched errors without exception. The first media access after a disk is switched must report a disk-switched condition; subsequent accesses under the same conditions should not report it.
- Error codes that a device driver returns must have the high byte cleared. The device dispatcher maintains certain error codes under certain conditions, and device dispatcher error codes are passed in the upper byte of the accumulator.

Table 10-4 Driver error codes and constants

Code	Constant	Description
$0000	`NoError`	No error occurred
$0010	`DevNotFound`	Device not found
$0011	`InvalidDevNum`	Invalid device number
$0020	`DrvrBadReq`	Invalid request
$0021	`DrvrBadCode`	Invalid control or status code
$0022	`DrvrBadParm`	Invalid parameter
$0023	`DrvrNotOpen`	Device not open (character device driver only)
$0024	`DrvrPriorOpen`	Device already open (character device driver only)
$0026	`DrvrNoResrc`	Resource not available
$0027	`DrvrIOError`	I/O error
$0028	`DrvrNoDev`	Device not connected
$0029	`DrvrBusy`	Device is busy
$002B	`DrvrWrProt`	Write protected (block device driver only)
$002C	`DrvrBadCount`	Invalid byte count
$002D	`DrvrBadBlock`	Invalid block number (block device driver only)
$002E	`DrvrDiskSw`	Disk switched (block device driver only)
$002F	`DrvrOffLine`	Device off line or no media present
$004E	`InvalidAccess`	Invalid access or access not allowed
$0058	`NotBlockDev`	Not a block device
$0060	`DataUnavail`	Data unavailable

Chapter 11 **System Service Calls**

GS/OS provides a standardized mechanism for passing information among its low-level components such as FSTs and device drivers. That mechanism is the system service call.

System service calls exist for various purposes: to perform disk caching, to manipulate buffers in memory, to set system parameters such as execution speed, to send a signal to GS/OS, to call a supervisory driver, and to perform other tasks.

This chapter documents the system service calls that a driver can make.

About system service calls

Access to several system service routines has been provided for device drivers by GS/OS. Access to these routines is through a system service dispatch table located in bank $01 from addresses $FC00 through $FCFF. A list of the available system service routines and their entry locations within the system service dispatch table is shown in Table 11-1.

■ **Table 11-1** System service calls

Call name	Location	Function
CACHE_FIND_BLK	$01FC04	Searches for a disk block in the cache
CACHE_ADD_BLK	$01FC08	Adds a block of memory to the cache
CACHE_DEL_BLK	$01FC14	Deletes a block of memory from the cache
ALLOC_SEG	$01FC1C	Returns virtual pointer to specified size segment
RELEASE_SEG	$01FC20	Frees block of memory obtained with ALLOC_SEG
SWAP_OUT	$01FC34	Marks a volume as off line
DEREF	$01FC38	Returns pointer to current location of virtual block
SET_SYS_SPEED	$01FC50	Controls processor execution speed
LOCK_MEM	$01FC68	Locks all GS/OS-managed memory segments
UNLOCK_MEM	$01FC6C	Unlocks all segments acquired with ALLOC_SEG call
MOVE_INFO	$01FC70	Moves data between memory buffers
SIGNAL	$01FC88	Notifies GS/OS of the occurrence of a signal
SET_DISKSW	$01FC90	Notifies GS/OS of a disk-switched or off-line condition
SUP_DRVR_DISP	$01FCA4	Makes a supervisory-driver call
INSTALL_DRIVER	$01FCA8	Dynamically installs a device into the device list
DYN_SLOT_ARBITER	$01FCBC	Returns slot status
UNBIND_INT_VECT	$01FCD8	Unbinds a link between interrupt vector and handler

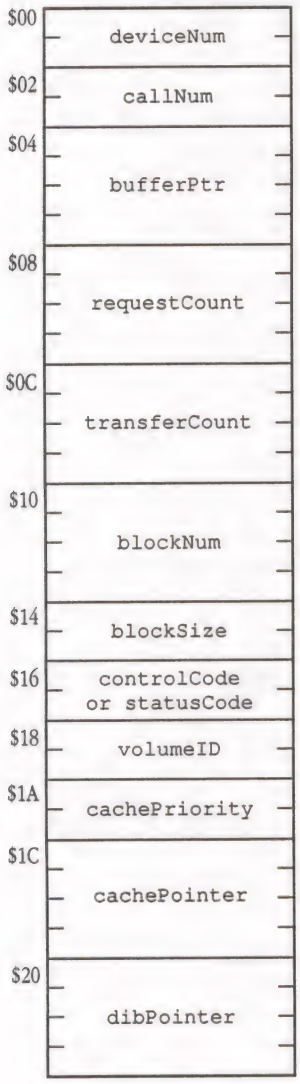
To make a system service call, follow this procedure:

1. Set up the parameters as required by the call (whether on GS/OS direct page, in registers, or on the stack).
2. Execute a JSL instruction to the proper location in the system service dispatch table.
3. When the call completes, take any parameters returned from the direct page or from registers, as indicated.

Descriptions of each of the system service routines follow.

Some system service calls make use of the **GS/OS direct-page** parameter space, the same parameter space used by the GS/OS driver calls described in Chapter 10. Figure 11-1 shows the GS/OS direct-page parameters. These are used by driver calls and some system service calls.

■ **Figure 11-1** GS/OS direct-page parameter space



CACHE_FIND_BLK ($01FC04)

Description

This routine attempts to find the requested block in the cache. If the block is found, it is moved to the start of the LRU chain, and a 4-byte pointer to its start is returned to the caller. One of two possible searches may be specified for this call: a search by device number (used by drivers) or a search by volume ID (used by FSTs when a deferred-write session is in progress). A routine making this system service call must specify the type of search desired by setting the carry flag appropriately.

Parameters

Input:

GS/OS direct page:

```
blockNum
deviceNum
volumeID
```

Carry flag: 0 = search by device number
1 = search by volume ID

Return:

GS/OS direct page:

`cachePointer` Pointer to start of block in cache

Notes

Full native mode is always assumed.

Drivers making this call should request a search by device number (c = 0).

Errors

If c = 0: no error; block is in cache.
If c = 1: error; block is not in cache.

CACHE_ADD_BLK ($01FC08)

Description

This routine attempts to add the requested block to the cache. The block is added at the start of the LRU chain (that is, at those most recently used). If there is not enough room in the cache, the block(s) at the end of the chain (that is, at those least recently used) are purged until there is enough room for the requested block.

Parameters

Input:
GS/OS direct page:

```
blockSize
blockNum
deviceNum
volumeID
cachePriority
```

Return:
GS/OS direct page:

```
cachePointer
```

Notes

Full native mode is always assumed.

When drivers make this call, the block is cached by device number.

Errors

If c = 0: no error; block was added to cache.
If c = 1: error; block was not added to cache.

CACHE_DEL_BLK ($01FC14)

Description This routine attempts to delete the specified block from cache memory.

Parameters *Input:*
GS/OS direct page:

```
blockSize
blockNum
deviceNum
volumeID
cachePriority
```

Return:
None

Notes Input and output are always passed by GS/OS direct-page locations in this routine. Full native mode is used.

Errors If c = 0: no error; block was deleted from cache.
If c = 1: error; block was not deleted from cache.

ALLOC_SEG ($01FC1C)

Description This routine returns a virtual pointer to a segment of the requested size.

Parameters

Input:
A register: requested memory block size (number of bytes)

Return:
X register: virtual pointer (low byte) to newly allocated block
Y register: virtual pointer (high byte) to newly allocated block

Notes None

Errors

If c = 0: no error; memory was allocated.
If c = 1: error; memory could not be allocated.

RELEASE_SEG ($01FC20)

Description Releases a memory segment that was allocated with the ALLOC_SEG call.

Parameters

Input:
X register: virtual pointer (low byte) to target block
Y register: virtual pointer (high byte) to target block

Return:
None

Notes None

Errors

If c = 0: no error; memory was freed.
If c = 1: error; memory was not freed.

SWAP_OUT ($01FC34)

Description This routine moves offline any volume in the device specified (a volume is offline if its media is not currently in a device). (Actually, all volumes with the passed device number are marked offline; there should never be more than one volume corresponding to a device number.) A volume associated with the specific device that has no open files is deleted from the system.

Parameters

Input:
A register: device number

Return:
None

Notes None

Errors None

DEREF ($01FC38)

Description This routine dereferences a virtual pointer and returns a pointer corresponding to the current location of the block referenced by the virtual pointer. This is the *only* way you should dereference virtual pointers.

Parameters

Input:
X register: virtual pointer (low byte)
Y register: virtual pointer (high byte)

Return:
X register: pointer (low byte) to dereferenced block
Y register: pointer (high byte) to dereferenced block

Notes The 32-bit pointer return in the X and Y registers points to the first byte in the block.

Errors None

SET_SYS_SPEED ($01FC50)

Description

This call allows hardware accelerators to stay compatible with device drivers that may have speed-dependent software implementations.

Whenever it dispatches to a driver, the device dispatcher obtains the device driver's speed class from the DIB and issues this system service call to set the system speed. When the driver completes the call, the device dispatcher restores the system speed to what it was before the call.

An accelerator card may intercept this vector and replace the system service call with its own routine, thus maintaining compatibility with GS/OS device drivers.

Parameters

Input:
The A register contains one of these speed settings:

Setting	Speed
$0000	Apple IIGS normal speed
$0001	Apple IIGS fast speed
$0002	Accelerated speed
$0003	Not speed dependent

Settings from $0004 through $FFFF are not valid.

Return:
The accumulator contains the speed setting that was in effect prior to issuing this system service call.

Notes

None

Errors

None

LOCK_MEM ($01FC68)

Description This routine locks all memory segments that were allocated with the ALLOC_SEG call. Use UNLOCK_MEM when you no longer need these segments; otherwise, the system could run out of available memory.

Parameters

Input:
None

Return:
None

Notes None

Errors None

UNLOCK_MEM ($01FC6C)

Description This routine releases all locked segments that were created with the ALLOC_SEG call.

Parameters *Input:*
None

Return:
None

Notes None

Errors None

MOVE_INFO ($01FC70)

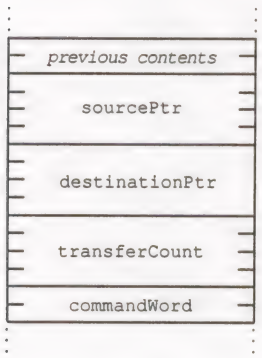
Description

This call transfers a block of data from a source buffer to a destination buffer. MOVE_INFO can be used by device drivers to transfer data from a single I/O location to a buffer or from a buffer to a single I/O location.

Parameters

The source buffer pointer, destination buffer pointer, and number of bytes to transfer are passed as input parameters to this routine via the stack. Source and destination buffers may be in the same or different memory banks, and either may straddle a bank boundary.

Input:
This is how the stack looks on entry to the call (before execution of the JSL instruction):

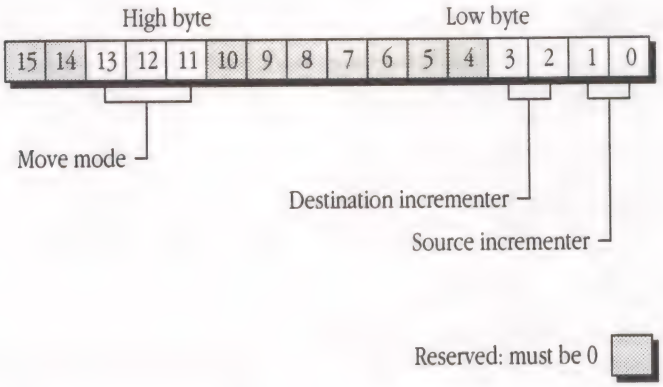


Parameters on stack	Size and type	Description
<— stack pointer		
sourcePtr	Longword pointer	Pointer to source buffer
destinationPtr	Longword pointer	Pointer to destination buffer
transferCount	Longword value	Number of bytes to transfer
commandWord	Word value	Flags (see description below)

The high bytes of `sourcePtr`, `destinationPtr`, and `transferCount` must be 0.

Return:
Data Bank register: unchanged
Direct register: unchanged
Accumulator: error code
X register: undefined
Y register: undefined

Command word The command word tells MOVE_INFO what kind of transfer to make and how to increment the destination and source addresses (useful, for example, for inverting the order of data as it is copied or for filling memory with a single value). The command word format is this:



where *move mode* can have these values and meanings:

000	(Reserved)
001	Block move
010–111	(Reserved)

and *destination incrementer* can have these values and meanings:

00	Constant destination
01	Increment destination by 1
10	Decrement destination by 1
11	(Reserved)

and *source incrementer* can have these values and meanings:

00	Constant source
01	Increment source by 1
10	Decrement source by 1
11	(Reserved)

Presently, only block moves are defined.

Source incrementer and destination incrementer define in what order successive bytes are transferred from the source buffer and in what order they are placed in the destination buffer. The following recommended predefined constant values for the MOVE_INFO command word cover most typical situations:

Move mode:

```
moveblkcmd      equ   $0800
                (a block move)
```

Most common command:

```
move_sinc_dinc  equ   $05+moveblkcmd
                (source and destination both increment)
```

Less common commands:

```
move_sinc_ddec  equ   $09+moveblkcmd
                (source increments, destination decrements)

move_sdec_dinc  equ   $06+moveblkcmd
                (source decrements, destination increments)

move_sdec_ddec  equ   $0a+moveblkcmd
                (source decrements, destination decrements)

move_scon_dcon  equ   $00+moveblkcmd
                (source constant, destination constant)

move_sinc_dcon  equ   $01+moveblkcmd
                (source increments, destination constant)

move_sdec_dcon  equ   $02+moveblkcmd
                (source decrements, destination constant)

move_scon_dinc  equ   $04+moveblkcmd
                (source constant, destination increments)

move_scon_ddec  equ   $08+moveblkcmd
                (source constant, destination decrements)
```

With these various combinations, buffers can be emptied or filled from the bottom up or from the top down, and single values can be placed in a buffer from the bottom up or from the top down. Some of the values are particularly helpful for moving data from one buffer into another buffer that overlaps the first.

Calling sequence From assembly language, you set up and invoke MOVE_INFO like this:

1. Place machine in full native mode (e = 0, m = 0, x = 0).
2. Push parameters onto stack as shown under "Parameters," earlier in this section.
3. Execute this instruction:

```
jsl Move_Info
```

Sample code Here is an assembly-language example of a call to MOVE_INFO:

```
rep  #$30
pea  source_pointer|-16    ;source pointer
pea  source_pointer
pea  dest_pointer|-16      ;destination pointer
pea  dest_pointer
pea  count_length|-16      ;count length
pea  count_length
pea  move_sinc_dinc        ;command word
jsl  move_info
```

Errors If c = 0: no error
If c = 1: error

SIGNAL ($01FC88)

Description

This call announces the occurrence of a specific signal to GS/OS and provides GS/OS with the information needed to execute the proper signal handler (previously installed with the ArmSignal subcall of the Driver_Control call). GS/OS queues this information and uses it when it dispatches to the signal handler.

For more information on GS/OS signals and signal handlers, see Chapter 9, "Handling Interrupts and Signals," of *GS/OS Reference.*

Parameters

Input:
A register: signal priority
X register: low word of signal-handler address
Y register: high word of signal-handler address

Return:
A register: undefined
X register: undefined
Y register: undefined

Signal priority: priority ranking of the signal, with $0000 being the lowest priority and $FFFF being the highest.

Signal-handler address: address of the signal-handler entry point.

Notes

A signal source that makes this call as the result of an interrupt should announce no more than one signal per interrupt to avoid the possibility of overflowing the signal queue.

Errors

None

SET_DISKSW ($01FC90)

Description

Some device drivers detect volume-off-line or disk-switched conditions through device-specific status calls rather than through returned errors. Such a condition would then not be detected by the device dispatcher on exit from the driver call. In fact, by GS/OS convention, off-line and disk-switched conditions should never be returned as errors from a status call; errors are reserved for conditions in which a call fails, not for passing status information.

With the call SET_DISKSW, drivers can specifically request that the disk-switched status (maintained internally by the device dispatcher) be set in this situation. SET_DISKSW, if necessary, removes the device's blocks from the cache and places its volumes off line (if the device dispatcher–maintained disk-switched flag has not already been set). All GS/OS drivers are expected to call SET_DISKSW if they detect a disk-switched or off-line condition as a result of a status call.

Parameters

Input:
GS/OS direct page:
`deviceNum` Device number of disk-switched device

Return:
None

Full native mode is assumed. Register contents are unspecified on entry and return, except that the Data Bank register and Direct register are unchanged by the call.

Errors

None

SUP_DRVR_DISP ($01FCA4)

Description

This call is the main entry point to the supervisor dispatcher. It dispatches calls among supervisory drivers. Supervisory drivers provide an interface that gives higher-level device drivers access to hardware.

Supervisory-driver calls can be classified into two groups. Calls with a supervisor number of zero are handled by the supervisor dispatcher; calls with a nonzero supervisor number are passed on to a supervisory driver.

The following calls are handled by the supervisor dispatcher and are not passed on to a supervisory driver:

Call no.	Sup. no.	Function
$0000	$0000	GetSupervisorNumber
$0001	$0000	Set_SIB_Pointer
$0002–$FFFF	$0000	(Reserved)

The following calls are dispatched by the supervisor dispatcher to a supervisory driver:

Call no.	Sup. no.	Function
$0000	(Nonzero)	Supervisor_Startup
$0001	(Nonzero)	Supervisor_Shutdown
$0002–$FFF	(Nonzero)	(Driver-specific calls)

These subcalls and other supervisory-driver calls are described in detail in Chapter 10, "GS/OS Driver Call Reference."

Notes

None

Errors

$28 Device not connected

INSTALL_DRIVER ($01FCA8)

Description

Because GS/OS supports removable, partitionable media on block devices, it must be able to install devices dynamically in its device list as new partitions come on line. INSTALL_DRIVER has been provided for that purpose.

△ **Important** The existence of this call implies that the GS/OS device list can grow during program execution. Drivers and applications cannot count on a fixed device list. See "Scanning the Device List," later in this section. △

Parameters

Input:
X register: DIB list address (low word)
Y register: DIB list address (high word)

Return:
A register: error code

DIB list address: longword input pointer: specifies the address of a list of device information blocks to be installed into the device list. The first field in the list is a longword that specifies the number of device information blocks to be installed; it is followed by a series of longword pointers, one to each DIB to be installed.

Notes

This call informs the device dispatcher that a driver or set of drivers is to be dynamically installed into the device list at the end of the next time the driver returns to the device dispatcher. When installing the driver, the device dispatcher inserts the device into the device list and then issues a startup call to the device. If space cannot be allocated in the device list for the new device, or if the device returns an error as a result of the startup call, then the device will not be installed into the device list.

Scanning the device list

There is no indication to an application that the device list has changed size as a result of this call. An application (such as the Finder) that scans block devices should always begin by issuing a DInfo call to device $0001 and should continue up the device list until error $11 (invalid device number) occurs. The DInfo call should have a parameter count of $0003 to give the application each device's device-characteristics word. If the new device is a block device with removable media, the application should make a status call to the device. If applications scan devices in this manner, dynamically installed devices will always be included in the scan operation.

Errors

Error checking is critical when using this call. Two possible errors may be returned. If error $54 (out of memory) occurs, it is not possible to install any drivers; if error $29 (device busy) occurs, it means that an INSTALL_DRIVER is already pending. In case the latter current driver installation cannot be accepted, the device driver must wait until it is accessed once more before it can install additional devices.

DYN_SLOT_ARBITER ($01FCBC)

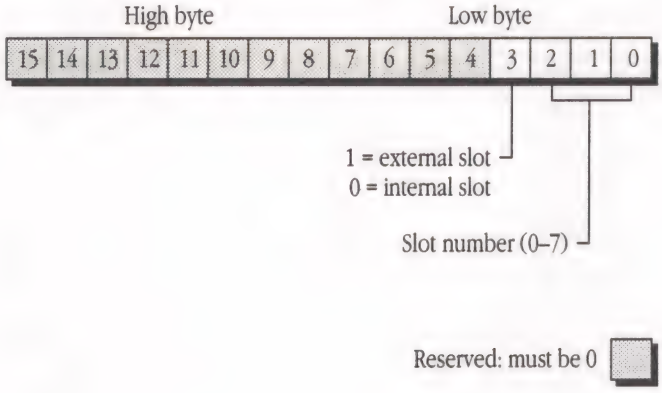
Description This call might provide support for dynamic switching between devices on internal and external slots in the future. At the time of publication, the call indicates only whether the slot is available.

Parameters

Input:
A register: requested slot
X register: undefined
Y register: undefined

Return:
A register: error
X register: byte-encoded slot configuration
Y register: undefined
Carry flag: cleared if requested slot was granted;
set if requested slot was denied

Requested slot: word input value: specifies the slot to be requested. The requested-slot parameter has this format:



Notes None

Errors Carry flag set if request denied

UNBIND_INT_VECT ($01FCD8)

Description This call allows the caller to perform an UnbindInt call when GS/OS is busy (typically during shutdown). There is no system service call to bind an interrupt source. To bind an interrupt source, use the BindInt GS/OS call.

Parameters

Input:
A register: `intNum` (from the BindInt GS/OS call)

Return:
None

Notes None

Errors None

Apple® IIGS GS/OS® Device Driver Reference

Appendixes

Appendix A Generated Drivers and Firmware Drivers

This appendix provides information of use to designers of BASIC, Pascal 1.1, ProDOS, SmartPort, and extended SmartPort peripheral cards; it explains how GS/OS constructs generated drivers for these devices and how it dispatches to them.

If you are writing a firmware driver for an Apple IIGS peripheral card, read this appendix. It explains how GS/OS recognizes your driver, dispatches to it, and manages I/O and caching for it, depending on what kind of driver it is.

Also see Chapter 7 of this reference for more information on generated drivers.

Generated-driver summary

At startup, for each slot that does not have an associated loaded driver, GS/OS looks for a firmware I/O driver. For slot *n*, GS/OS examines the appropriate firmware ID bytes in the $C*n*00 page of bank 0 and generates a GS/OS driver for any firmware driver it finds that uses BASIC, Pascal 1.1, ProDOS, SmartPort, or extended SmartPort protocols.

Each generated driver has an associated device information block (DIB), just like a loaded driver. The DIB contains device-specific information that can be used by the driver and by other parts of GS/OS.

GS/OS generates drivers for three broad types of slot-resident, firmware-based I/O drivers:

- **BASIC and Pascal 1.1 drivers**: For BASIC firmware drivers, a BASIC generated driver is created. For Pascal 1.1 firmware drivers, a Pascal 1.1 generated driver is created. For firmware drivers that support both BASIC and Pascal 1.1 protocols, a Pascal 1.1 generated driver is created.
- **ProDOS drivers:** Either one or two DIBs are created for each generated ProDOS block device driver, depending on the value of $C*n*FE.
- **SmartPort drivers:** All SmartPort block devices are supported by a single generated block device driver, and all SmartPort character devices are supported by a single generated character device driver. Each device's DIB is associated with either the character driver or the block driver.

All GS/OS generated drivers support these standard device calls:

- DInfo
- DStatus
- DControl
- DRead
- DWrite
- DRename

All generated drivers support the standard set of DStatus and DControl subcalls, although not all perform meaningful actions with all of them. No generated drivers support driver-specific DStatus or DControl calls.

◆ *Note:* For convenience and tradition, all addresses listed in this section are bank $00 addresses. Thus, the full Apple IIGS address corresponding to a listed address such as $C*n*05 would be $00 C*n*05.

Generating and dispatching to BASIC drivers

Generating

Because there are no conventional firmware ID bytes for BASIC drivers in the $C*n*00 space, GS/OS cannot always be sure that a BASIC card is *not* in a given slot. Therefore, to be safe, it creates a BASIC generated driver for every slot that is

- occupied by a peripheral card, or
- has no loaded driver, or
- has no ProDOS, Pascal 1.1, or SmartPort ID bytes

Dispatching

Contrary to the documented standard (see, for example, the *Apple IIGS Firmware Reference*), BASIC devices do not support a fixed entry point for input or output. The only defined entry point for BASIC device drivers is $C*n*00, which is the initialization entry point. The driver's initialization routine is responsible for putting the offsets to the driver output and input entry points into absolute zero-page locations $0036–0039. GS/OS maintains a list of the input and output entry points for BASIC devices as described in the following paragraphs.

This is the only BASIC device driver entry point:

$C*n*00 Initialization entry point

The driver initialization routine puts the proper values into page zero, so that the input and output entry points are as follows:

$C*n*00+($0038): add contents of $0038 to $C*n*00 to get input routine entry point
$C*n*00+($0036): add contents of $0036 to $C*n*00 to get output routine entry point

After initialization for a driver has been completed, GS/OS saves the entry points for the BASIC peripheral card.

This is the processor register state when dispatching to a BASIC driver:

Register	Contents
Accumulator	Character
X register	$C*n* (*n* = slot where driver resides)
Y register	$*n*0 (*n* = slot where driver resides)
P register	Unspecified

On completion of the dispatch to a BASIC driver, the processor register state must be this:

Register	Contents
Accumulator	Character
X register	Unspecified
Y register	Unspecified
P register	Unspecified

BASIC device drivers are not capable of returning errors. BASIC device drivers do not support a device status call.

Generated-driver interface

BASIC firmware drivers support single-character I/O only called through bank $00 of Apple IIGS memory. When a BASIC generated driver receives a multicharacter read or write request, it issues a separate call to the firmware driver for each character to be transferred. The generated driver also copies the character from the accumulator to the destination or from the source to the accumulator, if necessary.

Generating and dispatching to Pascal 1.1 drivers

Generating

At startup, GS/OS assumes that it has found a driver conforming to the Pascal 1.1 firmware protocol if all of the following conditions are true for slot *n*:

$C*n*05 = $38

$C*n*07 = $18

$C*n*0B = $01

In these circumstances, GS/OS creates a Pascal 1.1 generated driver to interface with that firmware driver and assigns a device ID to the generated driver.

Dispatching

Pascal 1.1 slot-resident firmware drivers support a standard set of entry points (not requiring a hook table like that needed for BASIC cards). Dispatches to Pascal 1.1 drivers occur by obtaining an offset and dispatching to $C*n*00+*offset*. The offset values are bytes stored at these addresses:

Address	Contents
$C*n*0D	Offset to initialization routine
$C*n*0E	Offset to read routine
$C*n*0F	Offset to write routine
$C*n*10	Offset to status routine

This is the processor register state when dispatching to a Pascal 1.1 driver:

Register	Contents
Accumulator	Character or request code (for status call)
X register	$C*n* (*n* = slot where driver resides)
Y register	$*n*0 (*n* = slot where driver resides)

The processor register state on completion of the dispatch to a Pascal 1.1 driver must be this:

Register	Contents
Accumulator	Character
X register	Error code on status; otherwise unspecified
Y register	Unspecified
P register	Unspecified

The Pascal 1.1 firmware I/O protocol is documented in the *Apple IIGS Firmware Reference.*

Generated-driver interface

Pascal 1.1 firmware drivers support single-character I/O only called through bank $00 of Apple IIGS memory. When a Pascal 1.1 generated driver receives a multicharacter read or write request, it issues a separate call to the firmware driver for each character to be transferred. The generated driver also copies the character from the accumulator to the destination or from the source to the accumulator, if necessary.

Generating and dispatching to ProDOS drivers

Generating

At startup, GS/OS assumes that it has found a driver conforming to the ProDOS protocol if all of the following conditions are true for slot *n:*

$C*n*01 = $20

$C*n*03 = $00

$C*n*05 = $03

$C*n*07 is not equal to $00

$C*n*FF is not equal to $00 or $FF

In these circumstances, GS/OS creates a ProDOS driver to interface with that firmware driver and assigns a device ID to the generated driver.

Dispatching

ProDOS block I/O drivers support a single standard entry point, which requires a parameter block in the absolute zero page to specify the call type. GS/OS supports these devices by generating the appropriate parameter block prior to dispatching to the slot-resident firmware driver. Entry points for ProDOS drivers are calculated as follows:

$C*n*00+($C*n*FF): add value of byte at address $C*n*FF to $C*n*00 to get entry point

This is the processor register state when dispatching to a ProDOS block I/O driver:

Register	Contents
Accumulator	Unspecified
X register	Unspecified
Y register	Unspecified

On completion of the dispatch to a ProDOS block I/O driver, the processor register state must be this:

Register	Contents
Accumulator	Error code
X register	Unspecified, except status returns low byte of block count
Y register	Unspecified, except status returns high byte of block count
P register	Carry set if error occurred; otherwise clear

The input parameters for the ProDOS block device driver are set up by the generated driver on absolute zero page as follows:

Offset	Parameter
$0042	Command byte
$0043	ProDOS unit number
$0044–$0045	Buffer pointer
$0046–$0047	Block number

Functions supported by the ProDOS block I/O driver include

- Status
- Read
- Write
- Format

The Format call is implemented only as a subcall (FormatDevice) of the GS/OS driver call Driver_Control. See Chapter 10 of this reference for more information.

The ProDOS block device protocol is documented in the *ProDOS 8 Technical Reference Manual*.

Generated-driver interface

ProDOS firmware block device drivers support only single-block transfers and can access only bank $00 of Apple IIGS memory. When a ProDOS generated driver receives a multiblock read or write request, the driver first checks that the request count is a multiple of the block size. If it is not, the generated driver returns an error; if it is, the generated driver issues a read or write call to the firmware driver for each block to be transferred. The generated driver also copies the data between the system bank $00 buffer and the caller's buffer (which may be anywhere in memory), if necessary.

The ProDOS generated driver supports caching. Blocks written to the ProDOS device through the firmware driver are also written to the cache (if enabled) by the generated driver; blocks to be read from the device may instead be read from the cache by the generated driver.

Generating and dispatching to SmartPort drivers

Generating

At startup, GS/OS assumes that it has found a driver conforming to the SmartPort protocol if all of the following conditions are true for slot *n*:

$C*n*01 = $20

$C*n*03 = $00

$C*n*05 = $03

$C*n*07 = $00

$C*n*FF is not equal to $00 or $FF

In these circumstances, GS/OS creates a SmartPort driver to interface with that firmware driver and assigns a device ID to the generated driver.

GS/OS then examines the SmartPort ID type byte at $C*n*FB to find out whether the drive supports only the standard SmartPort protocol or both the standard and extended protocols.

Dispatching

SmartPort drivers can support either the standard or the standard and extended SmartPort protocols. The standard **SmartPort protocol** uses 2-byte addresses and therefore cannot access or reside in Apple IIGS memory beyond bank $00. The **extended SmartPort protocol** uses 4-byte addresses and can access all parts of Apple IIGS memory. All SmartPort device drivers must support the standard protocol. GS/OS generated drivers permit use of the extended protocol only in cases where both the device driver and the device itself support it.

The SmartPort driver entry point is determined as follows:
$C*n*00+[($C*n*FF)+$03]: add (3 plus value of byte at address $C*n*FF) to $C*n*00 to get SmartPort entry point

This is the processor register state when dispatching to a SmartPort driver:

Register	Contents
Accumulator	Unspecified
X register	Unspecified
Y register	Unspecified

On completion of the dispatch to a SmartPort driver, the processor register state must be this:

Register	Contents
Accumulator	Error code
X register	Low byte count of bytes transferred to system
Y register	High byte count of bytes transferred to system
P register	Carry set if error occurred; otherwise clear

Calls to the standard SmartPort device driver use the following format:

```
jsr     smartport       ; call to standard smartport
dc      i1'command'     ; command byte
dc      i2'parameterlist'       ; pointer to parameter list
```

Calls to the extended SmartPort device driver use the following format:

```
jsr     smartport                   ; call to standard smartport
dc      ih'command'                 ; command byte
dc      i4'parameterlist'           ; pointer to parameter list
```

The SmartPort protocols, both standard and extended, are described in the *Apple IIGS Firmware Reference*.

Generated-driver interface

SmartPort firmware character device drivers support multiple-character I/O up to 767 bytes per request. Standard and extended calls are handled differently:

- Drivers that support only standard calls can access only bank $00 of Apple IIGS memory, and their data must be copied through the 512-byte system buffer in bank $00. Therefore, the generated driver makes multiple 512-byte requests until the remaining characters to transfer are fewer than 512; it then makes one final request for the remaining characters.
- Drivers that support extended calls can access any memory bank. In that case the generated driver makes multiple 768-byte requests until the remaining characters to transfer are fewer than 768; it then makes one final request for the remaining characters.

SmartPort firmware block device drivers support only single-block transfers. When a SmartPort generated driver receives a multiblock read or write request, the driver first checks that the request count is a multiple of the block size. If it is not, the generated driver returns an error; if it is, the generated driver issues a read or write call to the firmware driver for each block to be transferred. If either the firmware driver or the device it is attached to does not support extended SmartPort calls, the generated driver copies the data between the system bank $00 buffer and the caller's buffer (which may be anywhere in memory), if necessary.

The SmartPort generated block device driver supports caching. Blocks written to the SmartPort device through the firmware driver are also written to the cache (if enabled) by the generated driver; blocks to be read from the device may instead be read from the cache by the generated driver.

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

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

GS/OS error codes

The first column in Table B-1 lists the GS/OS error codes that an application can receive. The second column 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. The third column gives a brief description of what each error means.

■ **Table B-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

[continued]

■ **Table B-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 is full
$49	volDirFull	Volume directory is 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 pathnames for ChangePath
$5C	notSystemFile	Not an executable file

[continued]

■ **Table B-1** GS/OS errors [continued]

Code	Constant	Description
$5D	`osUnsupported`	Operating system not supported
$5F	`stackOverflow`	Too many applications on stack
$60	`dataUnavail`	Data unavailable
$61	`endOfDir`	End of directory has been reached
$62	`invalidClass`	Invalid FST call class
$63	`resNotFound`	File does not contain required resource

Glossary

absolute-bank segment: A load segment that is restricted to a particular memory bank but that can be placed anywhere within that bank. The ORG field in the segment header specifies the bank to which the segment is restricted.

abstract file system: The generic file interface that GS/OS provides to applications. Individual **file system translators** convert file information in abstract format into formats meaningful to specific file systems, and back again.

AppleDisk 3.5 driver: A GS/OS loaded driver that controls Apple 3.5 drives.

AppleDisk 5.25 driver: A GS/OS loaded driver that controls Apple 5.25 drives.

Apple II: Any computer from the Apple II family, including the Apple II Plus, the Apple IIc, the Apple IIe, and the Apple IIGS.

Apple 3.5 drive: A block device that can read 3.5-inch disks in a variety of formats.

Apple 5.25 drive: A disk drive that reads 5.25-inch disks. In this book, the essentially identical UniDisk, DuoDisk, Disk IIc, and Disk II drives are all referred to as *Apple 5.25 drives.*

application level: One of the three **interface levels** of GS/OS. The application level accepts calls from applications and may send them on to the file system level or the device level.

application-level calls: The calls an application makes to GS/OS to gain access to files or devices or to set or get system information. Application-level calls include **standard GS/OS calls** and **ProDOS 16–compatible calls**.

arm: To provide a **signal source** with the information needed to execute its **signal handler.** Signals are armed with a **subcall** of the device call DControl or the driver call Driver_Control.

associated file: In the ISO 9660 file format, a file analogous to the resource fork of a GS/OS **extended file.**

BASIC protocol: An I/O protocol for character devices, used by some firmware-based drivers on Apple II expansion cards.

block: (1) A unit of data storage or transfer, typically but not necessarily 512 bytes. (2) A contiguous region of computer memory of arbitrary size, allocated by the Memory Manager.

block device: A device that reads and writes information in multiples of one block of characters at a time. Disk drives are block devices.

block driver: A driver that controls a block device. Also called *block device driver.*

cache: A portion of the Apple IIGS memory set aside for temporary storage of frequently accessed disk blocks. By reading blocks from the cache instead of from disk, GS/OS can greatly speed I/O in some cases.

cache priority: A number that determines how a block is cached during a write operation. Depending on its priority, a block may be (1) not cached at all, (2) written both to the cache and to disk, or (3) written to the cache only (if a **deferred write** is in progress).

caching: The process of placing disk blocks in the cache and retrieving them. GS/OS uses an **LRU** caching mechanism, with a **write-through** cache.

call: (v.) To execute an operating-system routine. (n.) The routine so executed.

character device: A device that reads or writes a stream of characters in order, one at a time. The keyboard, screen, printer, and communications port are character devices.

character driver: A driver that controls a character device. Also called *character device driver.*

character FST: The part of the GS/OS file system level that makes character devices appear to application programs as if they were sequential files.

class-0 calls: See **ProDOS 16–compatible calls.**

class-1 calls: See **standard GS/OS calls.**

configuration list: A table of device-dependent information in a device driver, used to configure a specific device controlled by the driver. There are two lists for each configurable device: a **current configuration list** and a **default configuration list.**

configuration script: A set of commands, either part of a driver or in a separate module, that are used by a configuration program to display configuration options and allow a user to select among them. The configuration program then modifies the driver's **current configuration list** accordingly.

console: The main terminal of the computer; the keyboard and screen. Through the **console driver,** GS/OS treats the console as a single device.

console driver: A GS/OS **character driver** that allows applications to read data conveniently from the keyboard or write it to the screen.

Console Input routine: The part of the **console driver** that accepts characters from the keyboard. There are two basic input modes: **raw mode** and **user input mode.**

Console Output routine: The part of the **console driver** that writes characters to the screen.

control character: A nonprinting character that controls or modifies the way information is printed or displayed.

control code: (1) A **control character.** (2) A parameter in the device call DControl (and the driver call Driver_Control) whose value determines which control **subcall** is to be made.

controlling program: A program that loads and runs other programs, without itself relinquishing control. A controlling program is responsible for shutting down its subprograms and freeing their memory space when they are finished. A shell, for example, is a controlling program.

control list: A buffer used in some control subcalls to pass data to devices.

Control Panel program: A text-based Apple IIGS desk accessory that allows the user to make certain system settings, such as changing cache size and selecting external or internal firmware for slots. See also **Disk Cache program.**

current configuration list: One of the two configuration lists for each configurable device contolled by a driver; it contains the present values for all the device's configuration parameters.

data fork: The part of an extended file that contains data created by an application.

default configuration list: One of the two configuration lists for each device contolled by a driver; it contains the default configuration settings for the device.

deferred write: A process in which GS/OS writes blocks to the cache only, deferring writing to disk until all blocks to be written are in the cache. A deferred write session is started with a BeginSession call; it is ended (and all cached blocks are written to disk) with an EndSession call.

desktop interface: The visual interface that a typical Apple IIGS or Macintosh application presents to the user.

device: A physical piece of equipment that transfers information to or from the Apple IIGS computer. Disk drives, printers, mouse devices, and joysticks are external devices. The keyboard and screen are also a device (the **console**).

device call: See **GS/OS device calls.**

device characteristics word: Part of the **device information block,** this word describes some fundamental characteristics of the device, such as whether its driver is loaded or generated and what access permissions it allows.

device dispatcher: The component of GS/OS that controls all access to devices and device drivers. The device dispatcher handles informational calls about devices, passes on I/O calls to the proper driver, starts up and shuts down device drivers, and maintains the **device list.**

device driver: A driver that accepts **driver calls** from GS/OS and either (1) controls a hardware device directly or (2) accesses a **supervisory driver** that in turn controls the hardware.

device ID: A numerical indication of a general type of device, such as *Apple 3.5 drive* or *SCSI CD-ROM drive.*

device information block (DIB): A table of information describing a device. It is stored in the device's driver and used by GS/OS when accessing or referring to the device.

device level: One of the three **interface levels** of GS/OS. The device level mediates between the **file system level** and individual device drivers.

device list: A list of all installed devices; it is actually a linked list of pointers to all devices' DIBs. This list is constructed and maintained by the **device dispatcher.**

Device Manager: The part of GS/OS that provides application-level access to devices and device drivers.

device number: The number by which a device is specified under GS/OS. It is the position of the device in the **device list.**

DIB: See **device information block.**

directory entry: See **file entry.**

directory file: A file that describes and points to other files on disk. Compare **standard file, extended file.**

direct page: An area of memory used for fast access by the microprocessor; it is the 256 contiguous bytes starting at the address specified in the 65816 microprocessor's Direct register. Direct page is the Apple IIGS equivalent of the standard Apple II **zero page;** the difference is that it need not be page zero in memory. See also **GS/OS direct page.**

direct-page/stack segment: A load segment used to preset the location and contents of the direct page and stack for an application.

disarm: To notify a **signal source** that a particular **signal handler** will no longer process occurrences of the signal. Signals are disarmed with a subcall of the device call DControl or the driver call Driver_Control.

disk cache: See **cache.**

Disk Cache program: A graphics-based Apple IIGS desk accessory that allows the user to set the cache size. See also **Control Panel program.**

disk switched: A condition in which a disk or other recording medium has been removed from a device and replaced by another. Subsequent reads or writes to the device will access the wrong volume unless the disk-switched condition is detected.

dormant: Said of a program that is not being executed but whose essential parts are all in the computer's memory. A dormant program may be quickly **restarted** because it need not be loaded from disk.

driver: A program that handles the transfer of data to and from a peripheral device, such as a printer or disk drive. GS/OS recognizes two types of drivers in this regard: **device drivers** and **supervisory drivers.**

driver calls: A class of low-level calls, not accessible to applications, that access GS/OS **device drivers.** Driver calls are made from within GS/OS; all driver calls pass through the device dispatcher.

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare **static segment.**

extended file: A named collection of data consisting of two sequences of bytes, referred to by a single directory entry. The two different byte sequences of an extended file are called the **data fork** and the **resource fork.**

extended SmartPort protocol: See **SmartPort protocol.**

file: An ordered collection of bytes that has several attributes under GS/OS, including a name and a file type.

file entry: A component of a directory file that describes and points to some other file on disk.

filename: The string of characters that identifies a particular file within its directory. Compare **pathname.**

file system level: One of the three **interface levels** of GS/OS. The file system level consists of **file system translators (FSTs),** which take calls from the **application level,** convert them to a specific file system format, and send them on to the **device level.**

file system translator (FST): A component of GS/OS that converts application calls into a specific file system format before sending them on to **device drivers.** FSTs allow applications to use the same calls to read and write files for any number of file systems.

firmware I/O driver: A character or block driver on an expansion card in a slot (or in the slot's equivalent internal-port firmware). GS/OS creates **generated drivers** to provide applications and FSTs with a consistent interface to firmware I/O drivers.

format option entry: A description of a single formatting option for a particular device supported by a device driver. Part of the **format options table,** the format option entry includes such information as the interleave factor, the block size, and the number of blocks supported by the device.

format options table: A table in a **device driver** that contains formatting parameters for a device. The format options table contains a **format option entry** for each supported format.

FSTSpecific: A standard GS/OS call whose function is defined individually for each FST.

generated drivers: Drivers that are constructed by GS/OS itself, to provide a GS/OS interface to pre-existing, usually firmware-based peripheral-card drivers.

GS/OS: A 16-bit operating system developed for the Apple IIGS computer. GS/OS replaces ProDOS 16 as the preferred Apple IIGS operating system.

GS/OS calls: See **standard GS/OS calls.**

GS/OS device calls: A subset of the standard GS/OS calls, they bypass the **file system level** altogether, giving applications direct access to devices and device drivers.

GS/OS direct page: A portion of bank $00 memory used as a **direct page** by GS/OS. Some parts of the GS/OS direct page are used to pass parameters to **device drivers** and **supervisory drivers.**

GS/OS driver calls: See **driver calls.**

header: In **object module format,** the first part of every segment. Following the header, each segment consists of a sequence of **records.**

High Sierra: The High Sierra Group format; a common file format for files on CD-ROM compact discs. Similar to the **ISO 9660** international standard format.

High Sierra FST: The part of the GS/OS **file system level** that gives applications transparent access to files stored on optical compact discs (CD-ROM), in the most commonly used file formats: **High Sierra** and **ISO 9660.**

initialization segment: A segment in a load file that is loaded and executed independently of the rest of the program. It is commonly executed first, to perform any initialization that the program may require.

input port: In the console driver, a data structure that contains all of the information about the current input.

install: For an interrupt handler, to connect it to its interrupt source, with the GS/OS call BindInt (or the ProDOS 16 call ALLOC_INTERRUPT). For a signal handler, to connect it to its signal source, with the control or driver subcall ArmSignal. For a device (or driver), to put its DIB into the **device list,** thereby making it accessible to GS/OS and applications.

interface level: A conceptual division in the organization of GS/OS. GS/OS has three interface levels: the **application level,** the **file system level,** and the **device level.** The application level and the device level are external interfaces, whereas the file system level is internal to GS/OS.

interrupt: A hardware signal sent from an external or internal device to the CPU. When the CPU receives an interrupt, it suspends execution of the current program, saves the program's state, and transfers control to an **interrupt handler.** Compare **signal.**

interrupt dispatching: The process of handing control to the appropriate interrupt handler after an **interrupt** occurs.

interrupt handler: A program that executes in response to a hardware interrupt. Interrupts and interrupt handlers are commonly used by device drivers to operate their devices more efficiently and to make possible simple background tasks such as printer spooling. Compare **signal handler.**

interrupt source: Any hardware device that can generate an interrupt, such as the mouse or serial ports. Compare **signal source.**

inverse text: Text displayed on the screen with foreground and background colors reversed: instead of the usual light characters on a dark background, inverse text is in the form of dark characters on a light background.

ISO 9660: An international standard that specifies volume and file structure for CD-ROM discs. ISO 9660 is similar to the **High Sierra** format.

jump table segment: A segment in a load file that contains all references to dynamic segments that may be called during execution of that load file. The jump table segment is created by the linker. In memory, the loader combines all jump table segments it encounters into the jump table.

library file: An object file containing program segments, each of which can be used in any number of programs. The linker can search through the library file for segments that have been referenced in the program source file.

linker: A program that combines files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

loaded drivers: Drivers that are written to work directly with GS/OS and are usually loaded in from the system disk at boot time.

load file: The output of the linker. Load files contain memory images that the **System Loader** can load into memory, together with relocation dictionaries that the loader uses to relocate references.

long prefix: A GS/OS prefix whose maximum total length is approximately 8000 characters. Prefix designators 8/ through 31/ refer to long prefixes. Compare **short prefix.**

LRU: *Least recently used.* The caching method employed by GS/OS. When the cache is full and another block needs to be written to it, GS/OS purges the least recently used block(s)—the one(s) with the longest time since last access—to make room for the new block.

media variables: The set of multiple formatting options supported by a driver.

medium: (1) A disk, tape, or other object on which a storage device reads or writes data. Some media are removable; others are fixed. (2) A material, such as metal-oxide tape, from which storage objects are constructed.

Memory Manager: An Apple IIGS tool set that controls all allocation and deallocation of memory.

minimum parameter count: The minimum permitted value for the total number of parameters in the parameter block for a standard GS/OS call.

MouseText: Special characters, such as check marks and apples, used in some applications.

newline character: Any character (most typically a return character) that indicates the end of a sequence of bytes.

newline mode: A mode of reading data in which the end of the data (the termination of the Read call) is caused by reading a **newline character** (and not by a specific byte count).

no-wait mode: A mode for reading characters in which a driver accepts whatever characters are immediately available and then terminates a Read call, whether or not the total number of requested characters was read. No-wait mode allows an application to continue running while input is pending. Compare **wait mode.**

object file: The output from an assembler or compiler, and the input to a linker. It contains machine-language instructions.

object module format (OMF): The general format followed by Apple IIGS object files, library files, and load files.

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

parameter count: The total number of parameters in a block. Also called *pCount*. See also **minimum parameter count.**

partition map: A data structure describing the state of a specific partition on a device.

Pascal 1.1 protocol: An I/O protocol for character devices, used by some firmware-based drivers on Apple II expansion cards.

pathname: The complete name by which a file is specified. It is a sequence of filenames separated by **pathname separators,** starting with the filename of the volume directory and proceeding through any subdirectories that a program must follow to locate the file.

pathname segment: The segment in a load file that contains the cross-references between load files referenced by number (in the jump table segment) and their pathnames (listed in the file directory). The pathname segment is created by the linker.

pathname separator: The character slash (/) or colon (:). Pathname separators separate filenames in a pathname.

position independent: Code that is written specifically so that its execution is unaffected by its position in memory. It can be moved without needing to be **relocated.**

prefix: A portion of a **pathname,** starting with a volume name and ending with a subdirectory name. A prefix always starts with a **pathname separator** because a volume directory name always starts with a separator.

prefix designator: A number (0–31) or the asterisk character (*), followed by a **pathname separator.** Prefix designators are a shorthand method for referring to prefixes.

prefix number: See **prefix designator.**

ProDOS: (1) A general term describing the family of operating systems developed for Apple II computers. It includes both ProDOS 8 and ProDOS 16; it does not include DOS 3.3 or SOS. (2) The **ProDOS file system.**

ProDOS 8: The 8-bit ProDOS operating system, originally developed for standard Apple II computers but compatible with the Apple IIGS computer. In some earlier Apple II documentation, ProDOS 8 is called simply ProDOS.

ProDOS file system: The general format of files created and read by applications that run under ProDOS 8 or ProDOS 16 on Apple II computers. Some aspects of the ProDOS file system are similar to the GS/OS **abstract file system.**

ProDOS FST: The part of the GS/OS file system level that implements the ProDOS file system.

ProDOS protocol: An I/O protocol for block devices, used by some firmware-based drivers on Apple II expansion cards.

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

ProDOS 16–compatible calls: Also called *ProDOS 16 calls* or *class 0-calls*, a secondary set of **application-level calls** in GS/OS. They are identical to the ProDOS 16 system calls described in the *Apple IIGS ProDOS 16 Reference*. GS/OS supports these calls so that existing ProDOS 16 applications can run without modification under GS/OS.

purge: To delete the contents of a memory block.

quit return stack: An internal GS/OS stack that contains the user IDs of programs that have quit but wish to be launched again once the programs currently running finish executing.

raw mode: In the console driver, one of two **Console Input routines.** Raw mode allows for simple keyboard input.

record: In object module format, a component of a segment. Records consist of either program code or relocation information used by the linker or System Loader.

reload: To reexecute a program whose user ID has been pulled off the **quit return stack** but is not presently in a **dormant** state in memory. The System Loader can reload a program quickly because it has the program's **pathname** information; however, it is much faster to **restart** a dormant program than to reload it from disk.

reload segment: A load-file segment that is always loaded from the file at startup, regardless of whether the rest of the program is loaded from file or restarted from memory. Reload segments contain initialization information, without which certain types of programs would not be restartable.

relocate: To modify a file or segment at load time so that it will execute correctly at its current memory location. Relocation consists of patching the proper values onto address operands. The loader relocates load segments when it loads them into memory.

resource fork: One of the forks of an extended file. In the Macintosh file systems, the resource fork contains specifically formatted, generally static data used by an application (such as menus, fonts, and icons).

restart: To reexecute a program **dormant** in memory. Restarting is much faster than reloading because disk access is not required (unless the dormant application contains **reload segments**).

restartable: Said of an application that initializes itself and makes no assumptions about machine state when it executes. Only restartable applications can be restarted successfully from a **dormant** state.

restart-from-memory flag: A flag, part of the Quit call, that lets the System Loader know whether the quitting program can be restarted from memory if it is executed again.

return flag: A flag, part of the Quit call, that notifies GS/OS whether control should eventually return to the program making the Quit call.

run-time library file: A load file containing program segments–each of which can be used in any number of programs–that the System Loader loads dynamically when they are needed.

screen bytes: The actual values, as stored in screen memory, of characters displayed on screen (in Apple IIGS text mode).

segment: A component of an OMF file, consisting of a header and a body. In object files, each segment incorporates one or more subroutines. In load files, each segment incorporates one or more object segments.

separator: See **pathname separator.**

session: See **deferred write.**

short prefix: A GS/OS prefix whose maximum total length is 63 characters. Prefix designators */ and 0/ through 7/ refer to short prefixes. Compare **long prefix.**

SIB: See **supervisor information block.**

signal: A message from one software subsystem to a second that something of interest to the second has occurred. Compare **interrupt.**

signal handler: A program that executes in response to the occurrence of a signal. A useful feature of signal handlers is that, unlike interrupt handlers, they can make GS/OS calls. Compare **interrupt handler.**

signal queue: A portion of memory that holds a signal until it is ready to be handled. GS/OS does not allow signals to be handled until GS/OS is free to accept calls.

signal source: A software routine that announces a signal to GS/OS. Compare **interrupt source.**

SmartPort protocol: An I/O protocol for both block devices and character devices, used by the Apple IIGS disk port and by some firmware-based drivers on Apple II expansion cards. The *standard SmartPort protocol* uses two-byte pointers and can directly access only bank $00 of Apple IIGS memory; the *extended SmartPort protocol* uses four-byte pointers, so that data can be accessed anywhere in Apple IIGS memory.

special memory: On an Apple IIGS computer, all of banks $00 and $01 and all display memory in banks $E0 and $E1.

speed class: Part of the **device characteristics word,** it is a two-bit field that specifies what processor speed the device requires.

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's stack register.

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.**

static segment: A segment that is loaded only at program boot time and is not unloaded during execution. Compare **dynamic segment.**

status code: A parameter in the device call DStatus (and the driver call Driver_Status) whose value determines which status **subcall** is to be made.

status list: A buffer used by drivers to return data from some status **subcalls.**

status word: A parameter returned by the status or driver subcall GetDeviceStatus that describes some aspects of a device's current status, such as whether it is busy or whether it is interrupting.

subcall: An instance of a device call or driver call in which one of the call input parameters selects which routine is to be invoked. For example, if the parameter `statusCode` in the device call DStatus (or the driver call Driver_Status) has the value $0003, the status or driver subcall GetFormatOptions is executed.

supervisor: See **supervisory driver.**

supervisor dispatcher: The component of GS/OS that controls all access to supervisory drivers. The supervisor dispatcher handles informational calls about supervisory drivers, passes on I/O calls from device drivers, starts up and shuts down supervisory drivers, and maintains the **supervisor list.** Compare **device dispatcher.**

supervisor execution environment: The execution environment set up by the supervisor dispatcher for each supervisory-driver call.

supervisor ID: A numerical indication of the general type of supervisory driver, such as AppleTalk or SCSI.

supervisor information block (SIB): A table of information describing a supervisory driver. It is stored in the supervisory driver and used by GS/OS when accessing or referring to the driver. Compare **device information block.**

supervisor list: A list of pointers to the SIBs of all installed supervisory drivers. Compare **device list.**

supervisor number: The identifying number for each installed supervisory driver. It is equivalent to the driver's position in the **supervisor list.**

supervisory driver: A driver that arbitrates **supervisory-driver calls** from separate device drivers and dispatches them to the proper devices. Supervisory drivers are used when several individual device drivers must access several different devices through a single hardware controller.

supervisory-driver calls: Calls that a supervisory driver accepts from its individual device drivers. They are different from **driver calls,** although many may be direct translations of driver calls.

system file: Under ProDOS 8, any file of ProDOS file type $FF whose name ends with ".SYSTEM". In GS/OS, several different types of files are defined as system files.

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

system service call: A low-level call in a common format used by internal components of GS/OS—such as FSTs—and used between GS/OS and device drivers.

terminator: A character that terminates a console driver Read call. The **console driver** permits more than one terminator character and also can note the state of modifier keys in considering whether a character is to be interpreted as a terminator. Compare **newline character.**

terminator list: A list of terminator characters kept track of by the **console driver.**

text port: In the **console driver,** a rectangular portion of the screen in which all console output operations occur.

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

UniDisk 3.5 drive: An intelligent block device that can read 3.5-inch disks in a variety of formats.

UniDisk 3.5 driver: A GS/OS loaded driver that controls UniDisk 3.5 drives.

user ID: A number, assigned by the User ID Manager, that identifies the owner of every allocated block of memory in the Apple IIGS computer. Generally, each application has a particular user ID, with which all its allocated memory is identified. The user ID is also used as a general identifier of the program itself.

user input mode: One of the two **Console Input routines,** this mode allows for text-line editing and application-defined terminator keys.

vector reference number (VRN): The unique identifier given to each **interrupt source** that is explicitly identifiable by the firmware. VRNs are used to associate interrupt sources with **interrupt handlers.**

volume: A named collection of files on a logical storage device.

volume ID: A number assigned to every volume on an installed device.

wait mode: A mode for reading characters in which a driver does not terminate a Read call until the total number of requested characters is read. In wait mode, normal program execution is suspended until input is completed. Compare **no-wait mode.**

write-through: The kind of cache implemented by GS/OS. When a driver writes a block of data, it writes the same data to the block in the cache and the equivalent block on the disk. Never does the block in the cache contain information more recent than the disk block (unless a **deferred write** session is in progress).

zero page: Also called *absolute 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 0, only a single byte is needed to specify a zero-page address. Compare **direct page.**

Index

G

H

I, J, K

L

M

N

O

P

Q

R

S

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word software. Proof pages were created on Apple LaserWriter® printers. Final pages were created on the Varityper™ VT600 imagesetter. Line art was created using Adobe Illustrator™. POSTSCRIPT®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Apple's corporate font, a condensed version of ITC Garamond®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writers: Dave Carpenter, Bill Harris, and Dave Bice

Copy Editor: Beverly Zegarski

Illustrators: Sandee Karr and Peggy Kunz

Production Supervisor: Teresa (Tess) Lujan

Formatter: Gerri Gray

Special thanks to Greg Branche, Mark Day, Matt Deatherage, Matt Gulick, Jim Luther, Dave Lyons, and Ray Montagne.

LICENSE AGREEMENT

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING OUR PRODUCT. BY USING OUR PRODUCT, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE PRODUCT TO RESOURCE CENTRAL AND YOUR PURCHASE PRICE WILL BE REFUNDED.

Apple retains certain rights to its software as specified in this license.

1. License. Resource Central, Inc. licenses to you the materials we license from Apple (the "Software"). You own the disk on which the Software is recorded but Apple retains title to the Software and related documentation. This License allows you to use the Software on a single Apple computer and make one copy of the Software in machine-readable form for backup purposes only. You must reproduce on such copy the copyright notice and any other proprietary legends that were on the original copy of the Software. You may also transfer all your license rights in the Software, the backup copy of the Software, the related documentation and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. Restrictions. The Software contains copyrighted material, trade secrets and other proprietary materials. In order to protect our rights in them you may not decompile, reverse engineer, disassemble or otherwise reduce the Software to a human-perceivable form. You may not modify, network, rent, lease, loan, distribute or create derivative works based up the Software in whole or in part. You may not electronically transmit the Software from one computer to another, whether by use of a local network, the public telephone system, or otherwise.

3. Termination. This license is effective until terminated. You may terminate the license at any time by destroying the Software, related documentation, and all copies thereof. This License will terminate immediately without notice from Resource Central, Inc.or Apple Computer, Inc. if you fail to comply with any provision of this License. Upon termination you must destroy the Software and related documentation and all copies thereof.

4. Export Law Assurances. You agree and certify that neither the Software nor any other technical data received from Resource Central Inc., nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States.

5. Government End Users. If you are acquiring the Software on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees:
(i) if the Software is supplied to the Department of Defense, the Software is classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software and its documentation as that term is defined in Clause 252.227- 7013(c)(1) of the DFARS; and (ii) if the Software is supplied to any unit or agency of the United States Government other than the Department of Defense, the Government's rights in the Software and its documentation will be as defined in Clause 52.227-19(c)(2) of the FAR, or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

6. Other Limitations. The Disclaimer of Warranty and Limitation of Liability presented separately within this document are also a part of this License.

7. Controlling Law. This License shall be governed by and construed in accordance with the laws of the United States and the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

8. Complete Agreement. This License constitutes the entire agreement between the parties with respect to the use of the Software, the Publication, and related documentation, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Resource Central, Inc.

NEITHER APPLE COMPUTER, INC. NOR RESOURCE CENTRAL, INC. MAKES ANY WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. APPLE COMPUTER, INC. AND RESOURCE CENTRAL, INC. DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

IN NO EVENT WILL APPLE COMPUTER, INC., RESOURCE CENTRAL, INC. AND THEIR DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF APPLE COMPUTER, INC. AND RESOURCE CENTRAL, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. Apple Computer, Inc. and Resource Central, Inc.'s liability to you for actual damages from any cause whatsoever, and regardless of the form of the actions (whether contract, tort (including negligence), product liability or otherwise), will be limited to $50.