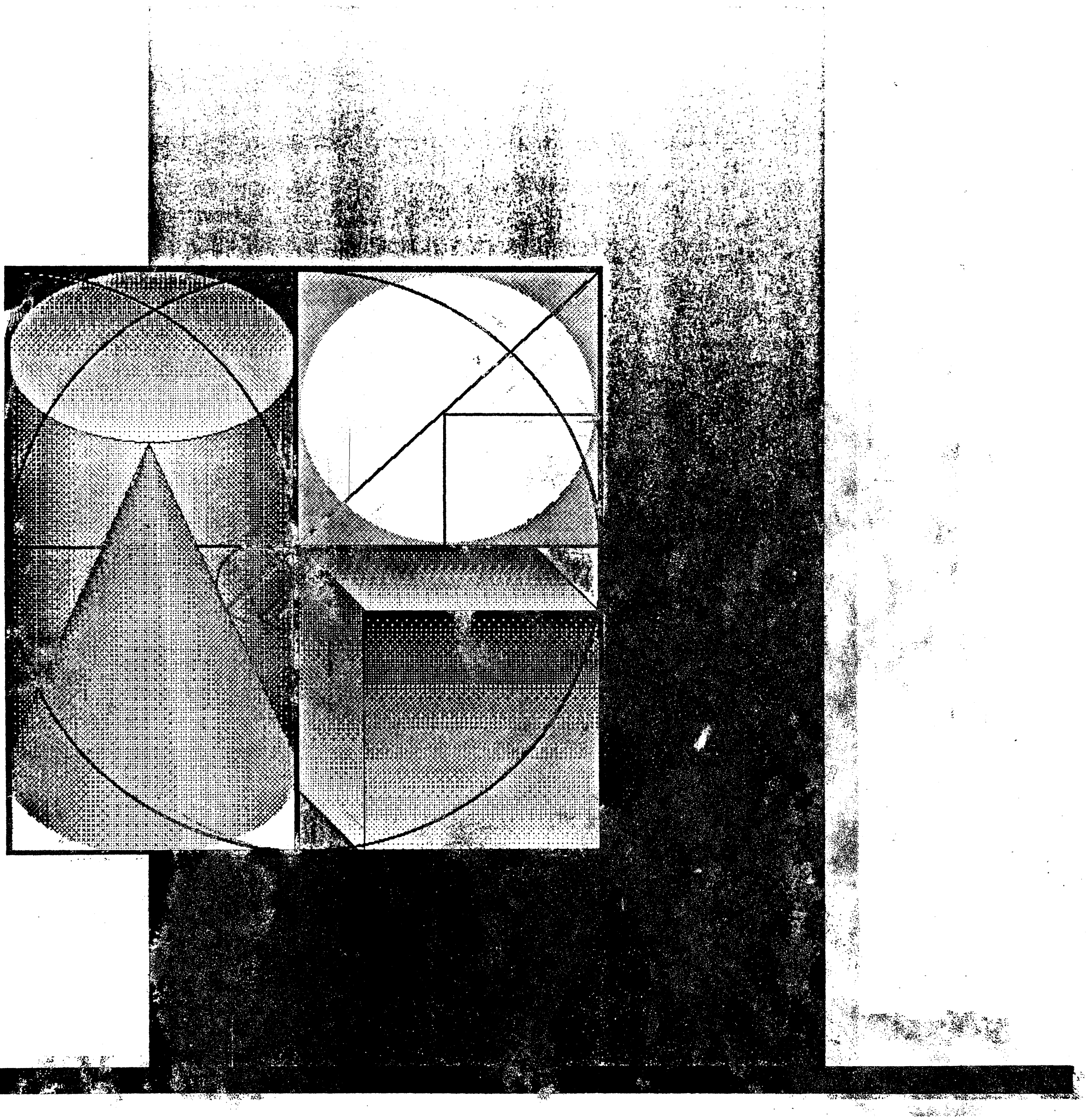
GS/OS™ Reference

Volume 2 (Beta Draft)

APDA #A0008LL/A



Apple® II GS/OS™ Reference

Includes System Loader

Volume 2

Devices and GS/OS

APDA Draft

William H. Harris
Developer Technical Publications
January 26, 1989

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

Simultaneously published in the United States and Canada.

2/21/88

Contents

Figures and Tables

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

One of the principal goals of GS/OS is to provide application writers with access to a wide variety of hardware devices, while insulating them (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 Volume 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 Volume 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. It also describes how to write and install GS/OS interrupt handlers and signal handlers, code segments that execute automatically in response to hardware or software requests.

Appendixes to this Volume describe how the System Loader works, what the file format for Apple IIGS executable files is, how GS/OS generated drivers interact with slot-based firmware I/O drivers, and what errors GS/OS can return. Also included are assembly-language code examples of four different types of GS/OS drivers.

What is the device level?

As described in the Introduction to Volume 1, 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.

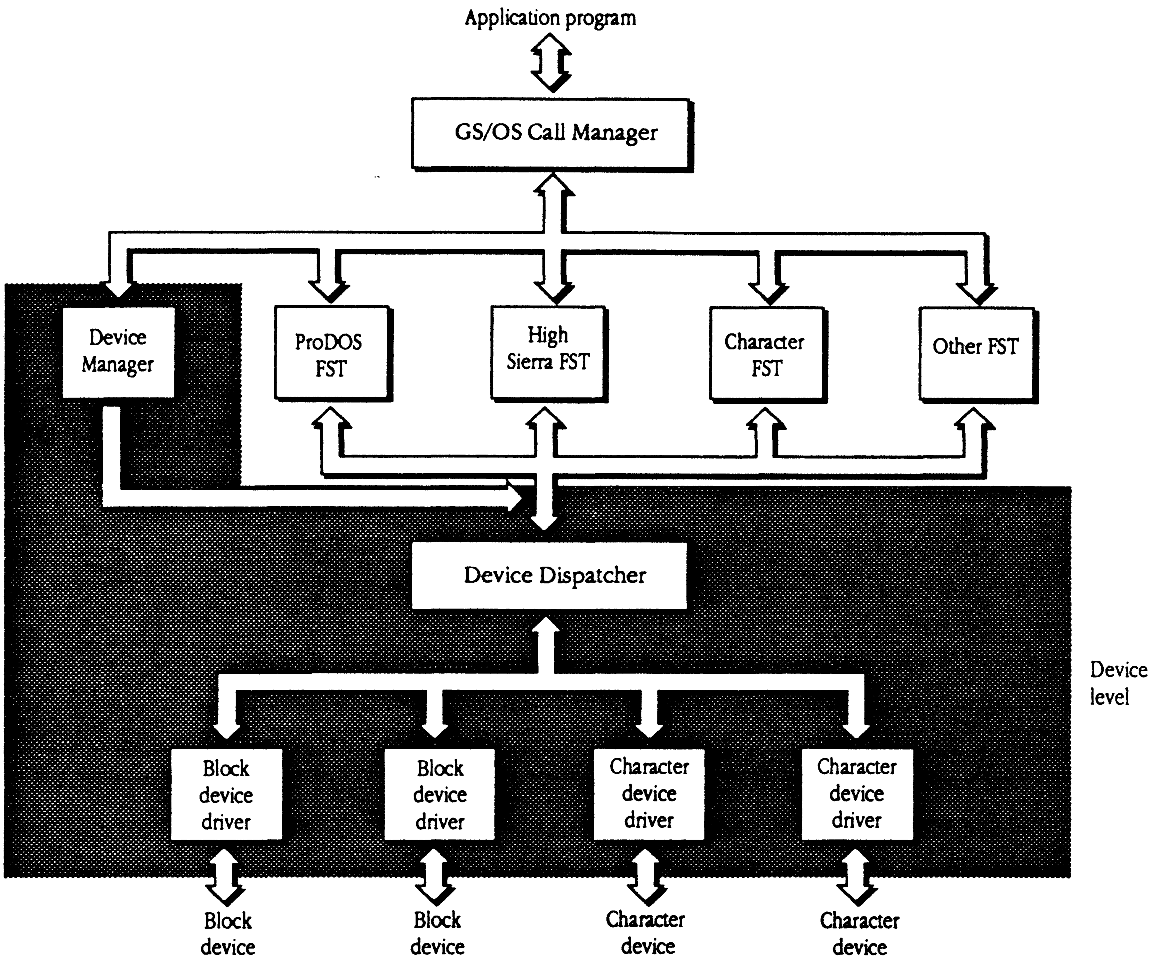
In general, the device level sits between the file system level and hardware devices, translating the file I/O calls made by an application into the calls that access data on peripheral devices. Note also that part of the device level (The Device Manager) extends upward into the level occupied by file 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 11.
- Applications that wish to access devices directly make **device calls,** which 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 **configuration, caching, interrupt-handling,** and **signal-handling** 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 12 of this Volume.

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

■ **Figure I-1** The 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 sections 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 only to 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 6) 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 d**rivers 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 which 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 Volume.

Device drivers and supervisory drivers

It is simplest to assume that each hardware device is associated with only one driver and 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.

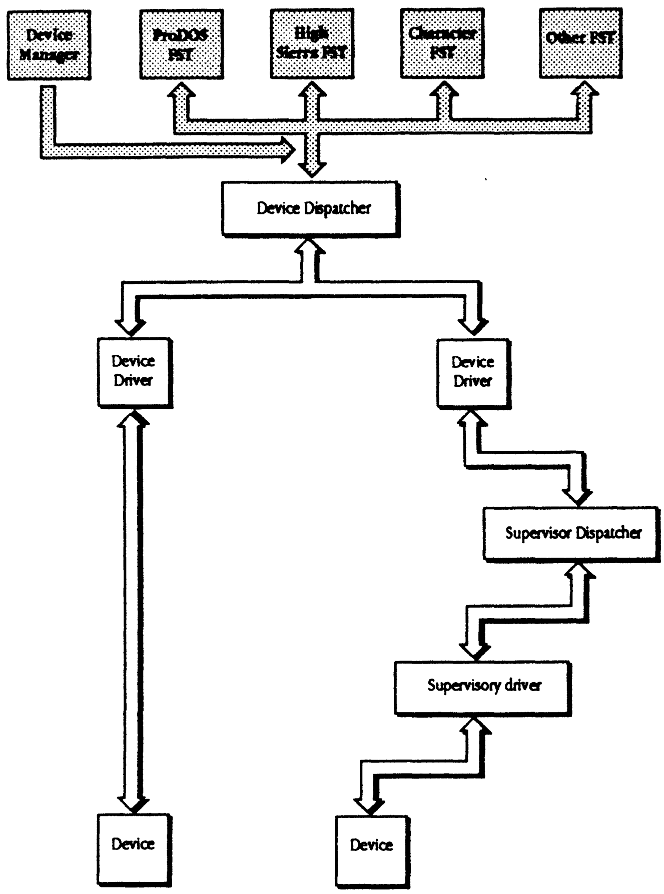
More complexity is possible, however. In some cases there are logical "devices" (hardware controllers such as a SCSI port) that must handle I/O requests from more than one driver (for example, a SCSI hard disk driver and a SCSI CD-ROM driver) and access more than one type of device. To handle those situations, 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 hierarachy is diagrammed in Figure I-2.

■ **Figure I-2** Driver hierarchy within the 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 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 Chapters 8 and 11. The rest of the discussion in this chapter concerns device drivers only.

How applications access devices

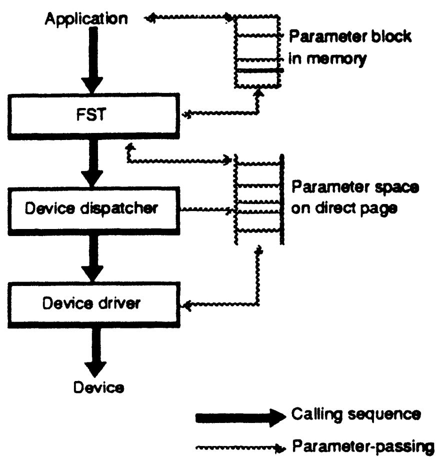
When an application makes a call that results in any kind of I/O, device access occurs. That device access is either indirect, through a file system translator (FST), or direct, through the Device Manager.

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 7 of Volume 1) 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 accesssed. It might not even know which file system is being used. Figure I3 shows the schematic progress of a typical GS/OS call from application to device, including how parameters are passed.

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



High-level calls pass parameters differently than low-level calls. When an FST receives a call from an application, it converts the parameter block information into data on the GS/OS **direct page**; that 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 that 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.

On the other hand, there are times at which 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.

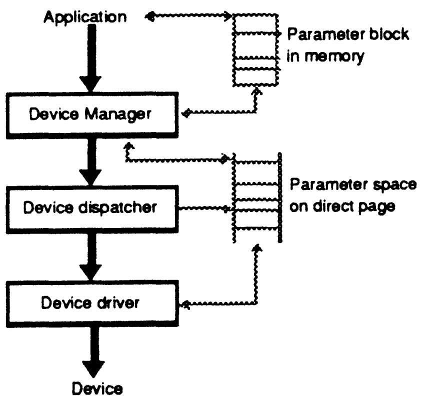
Device calls are application-level GS/OS calls, just like all the calls discussed in Chapter 7 of Volume 1. Your application sets up a parameter block in memory and makes the call as described in Chapter 3 of Volume 1. The only difference from a normal file-access call is that the device calls are routed 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 five GS/OS calls: DInfo, DStatus, DControl, DRead, and DWrite. 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 Volume.

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



Parameter-passing in device calls is the same as in GS/OS calls that pass through FSTs. 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 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

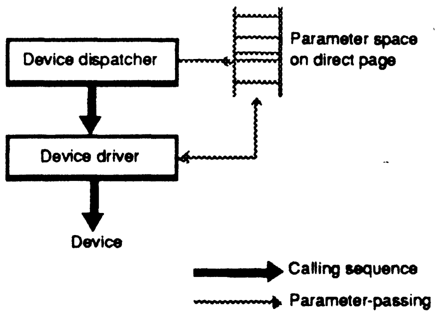
All calls to device drivers pass through the device dispatcher. The device dispatcher maintains a list of information about each driver attached to the system, and thus knows where to transfer control when it receives a driver call from an FST or the Device Manager.

The driver calls that the device dispatcher receives from FSTs or the Device Manager and passes on to drivers are these: Driver_Status, Driver_Control, Driver_Read, and Driver_Write. They are documented in Chapter 11. 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 call DInfo; DInfo is handled entirely by the device dispatcher, by consulting its list of device information. No access of the driver or device is necessary for DInfo.

The device dispatcher and other parts of GS/OS also make driver calls that are not translations of device calls. These other driver calls are concerned with setting up drivers to perform I/O and shutting them down afterward. They are Driver_Startup, Driver_Open, Driver_Close, Driver_Flush, and Driver_Shutdown, and are documented in Chapter 11. Figure I-5 shows the progress of such a driver call; note that Figure I-5 also is identical to the lower part of Figures I-3 and I-4.

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



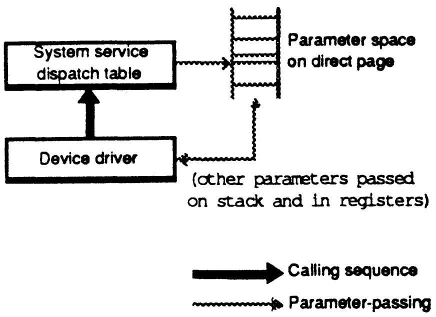
System service calls

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 12 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 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 a 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 in the first place are specific to the individual driver. However, the overall format in which the configuration parameters are to be to be stored in the device driver, and what calls are used to set or modify those parameters, are defined in Chapters 8 and 11.

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, the user can control whether caching is enabled and 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 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 (= read) 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 sevice calls are documented in Chapter 12.

Interrupt handling

An **interrupt** is 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.** The interrupt handler performs the functions required by the occurrence of the interrupt and returns control to the CPU, which restores the state of the interrupted application and resumes execution of the application as if nothing had happened.

In a non-multitasking system such as GS/OS, interrupts are commonly used by device drivers to operate their devices more efficiently and to make possible simple background tasks such as printer spooling.

When installed, a GS/OS interrupt handler can be associated with a particular class of interrupt source, for faster dispatching when an interrupt occurs. GS/OS interrupt handlers are installed and removed with the GS/OS calls BindInt and UnbindInt. How to write interrupt handlers for GS/OS device drivers or applications is discussed in Chapter 11.

Signals and signal handling

A **signal** is a message from one software subsystem to a second that something of interest to the second has occurred. When a signal occurs, GS/OS typically places it in the **signal queue** for eventual handling. As soon as it can, GS/OS suspends execution of the current program, saves the program's state, removes the signal from the queue, calls the signal handler in the receiving subsystem to process the signal, and finally restores the state and returns to the suspended program.

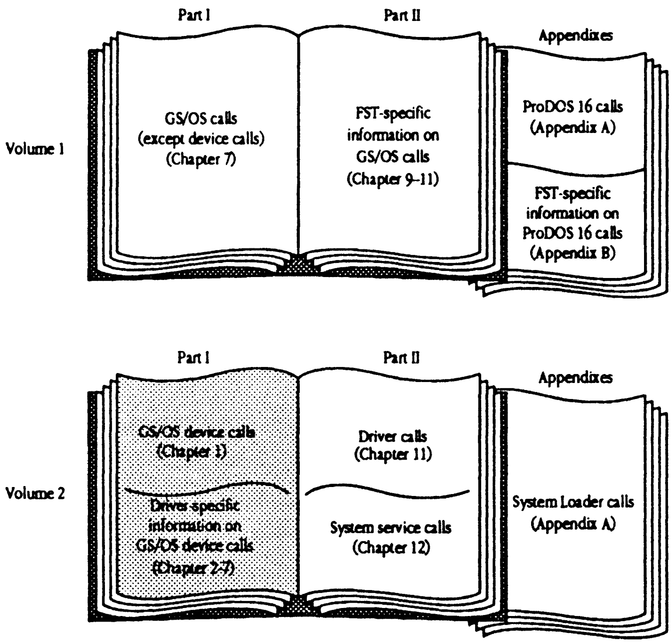
The most important feature of signal handlers is that they are allowed to make GS/OS calls. That is why the signal queue exists; GS/OS removes signals from the queue and executes their signal handlers only when GS/OS is free to accept a call. The most common kind of signal is a software response to a hardware interrupt, but signals need not be triggered by interrupts.

Signals are analogous to interrupts, but are handled with less urgency. If immediate response to an interrupt request is needed, and if the routine that handles the interrupt needn't make any operating-system calls, then it should be an interrupt handler. On the other hand, if a certain amount of delay can be tolerated, the full range of operating system calls are available to a handler if it is a signal handler.

A **signal source** is a software routine (perhaps an interrupt handler) that announces a signal to GS/OS; the signal handler associated with that source is then executed as a result of the signal occurrance. GS/OS signal sources and handlers are installed and removed with the device call DControl or the driver call Driver_Control.

Interrupt handlers, signal handlers, and signal sources are commonly written in conjunction with drivers. If you want to write a signal source or a signal handler or both to go with your driver or application, see Chapter 10.

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 Chapter 7 of Volume 1, except that it 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 Volume 1.

How to make a device call

Your application makes GS/OS device calls just like any other application-level GS/OS calls—it sets up a parameter block in memory, and executes either an in-line or stack-based call method (either directly or with a macro). Chapter 3 of Volume 1 describes all the methods for making GS/OS calls.

All device calls are handled by the Device Manager. Table 1-1 lists them. 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

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 one byte; successive tick marks down the side of the block represent successive bytes in memory. Each diagram also includes these features:

- **Offset:** Hexadecimal numbers down the left side of the parameter block represent byte offsets from the base address of the block.
- **Name:** The name of each parameter appears at the parameter's location within the block.
- **No.:** Each parameter in the block has a number, identifying its position within the block. The total number of parameters in the block is called the **parameter count** (`pCount`); `pCount` is the initial (zeroth) parameter in each call. The `pCount` parameter is needed because in some calls parameter count is not fixed; see 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.

$202C DInfo

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

:HD2 RamFast

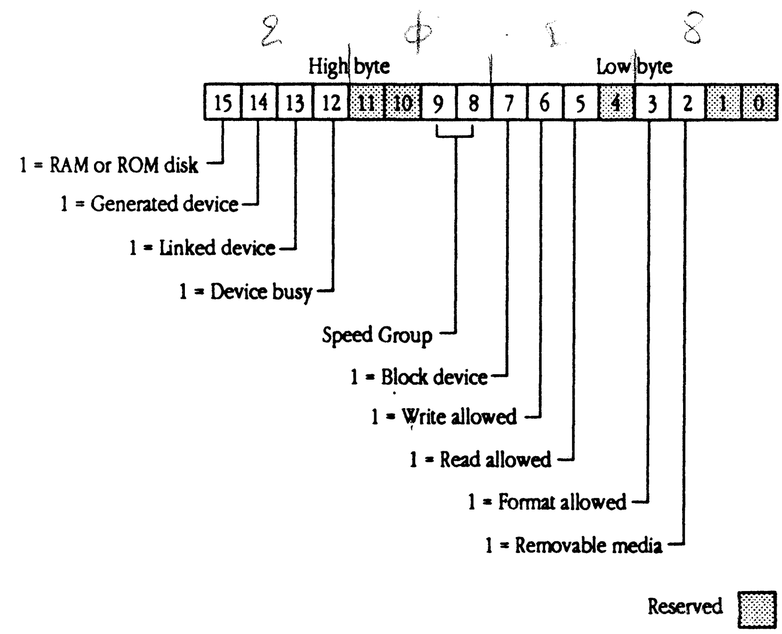
	Offset		No.	Size and type
B	\$00	pCount	—	Word INPUT value (minimum = 2)
4	\$02	devNum	1	Word INPUT value
.RAMFAST.SCSI.2	\$04	devName	2	Longword INPUT pointer
2ØF8	\$08	characteristics	3	Word RESULT value
91132 ← \$528C	\$0A	totalBlocks	4	Longword RESULT value
F	\$0E	slotNum	5	Word RESULT value
2	\$10	unitNum	6	Word RESULT value
1Ø1B	\$12	version	7	Word RESULT value
5	\$14	deviceIDNum	8	Word RESULT value
1	\$16	headLink	9	Word RESULT value
Ø	\$18	forwardLink	10	Word RESULT value
Ø	\$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 31 bytes, so the maximum size of the returned value is 33 bytes. Thus the buffer size should be 35 bytes.

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



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

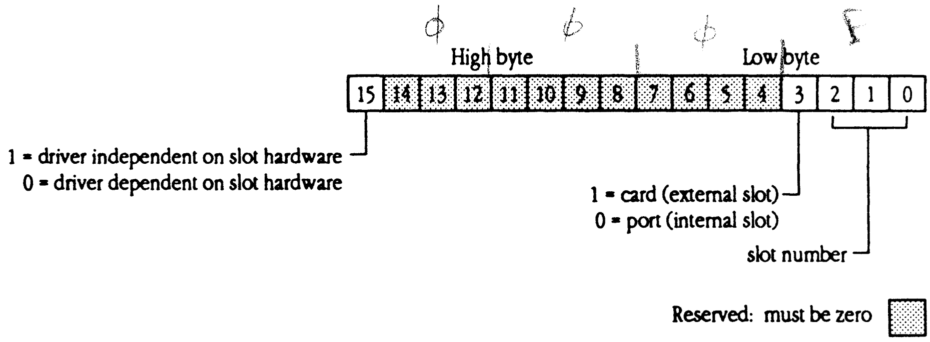
Speed group defines the speed at which the device requires the processor to be running. Speed group has these binary values and meanings:

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

`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 zero.

`slotNum` Word result value: Slot number of the (1) device hardware or (2) 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) or an external slot containing a card with its own firmware.

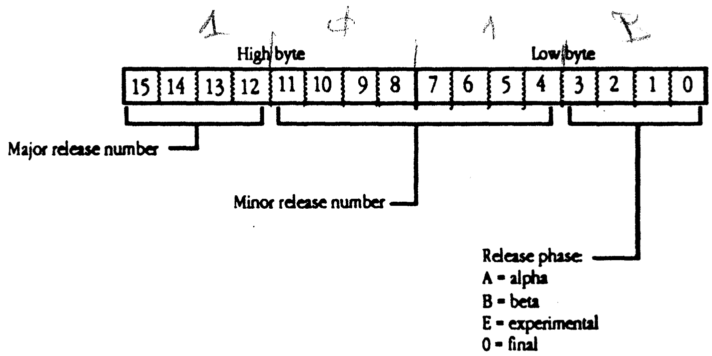
For a given slot number, either the external slot or its equivalent internal port is active (switched-in) at any one time; Bit 15 indicates whether or not the device driver must access the peripheral card's I/O addresses. For more information on those addresses, see the *Apple IIe Technical Reference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



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

`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 zero 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 zero 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). Only certain devices have extended DIBs.

Errors

\$11	invalid device number
\$53	parameter out of range

$202D DStatus

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 correpsonds to a particular DStatus subcall, described under DStatus Subcalls, 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.

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

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

Status code = $0000.

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

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

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

1Ø 4Ø FF FF ØØ ØØ

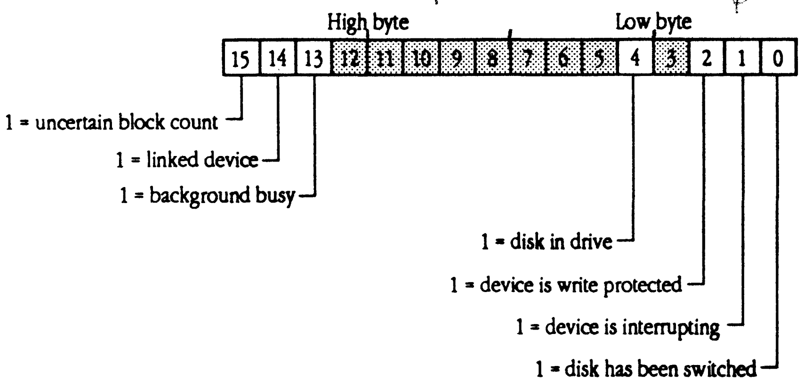
32 Mégas = 64 k blocs

p. count ØØØ5
dev. num ØØØ1
status. code ØØØØ
list_pt ADD1 —— D 1Ø 4Ø FF FF ØØ ØØ
req ØØØØ ØØØ6
tx " " 6

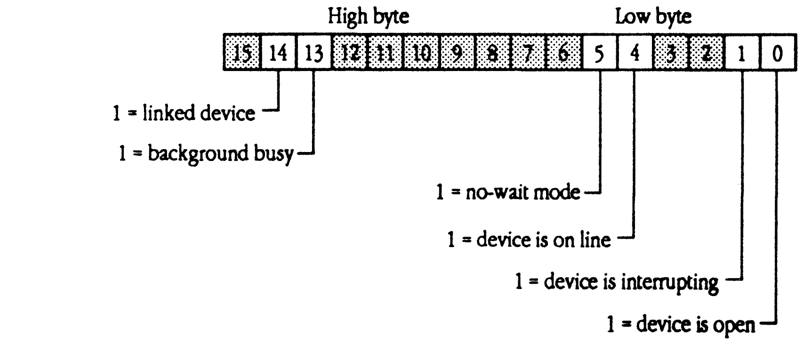
The device status word has two slightly different formats, depending on whether the device is a bock device or a character device. This is its definition:

Block device:

4 ϕ A ϕ



Character device:



Reserved: must be zero

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

GetConfigParameters (DStatus subcall)

Status code = $0001.

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

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

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

GetWaitStatus (DStatus subcall)

Status code = $0002.

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

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

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

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

GetFormatOptions (DStatus subcall)

Status code = $0003.

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

- An application can select one with a SetFormatOptions subcall, prior to formatting a block device. See the description of the DControl call, 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 8 of Volume 1.

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

Offset	Field	Size	Description
$00	`numOptions`	Word	Number of format-option entries in the list
$02	`numDisplayed`	Word	Number of options to be displayed
$04	`recommendedOption`	Word	Recommended default formatting option
$06	`currentOption`	Word	The option with which the currently on-line media was formatted
$08	`formatOption1`	(16 bytes)	The first format option entry
$0C	⋮		
	`formatOptionN`	(16 bytes)	The last format option entry

Of the total number of options in the list, zero or more can be displayed on the initialization dialog presented to the user when initializing a disk (see the calls Format and EraseDisk in Chapter 7 of Volume 1). 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.)

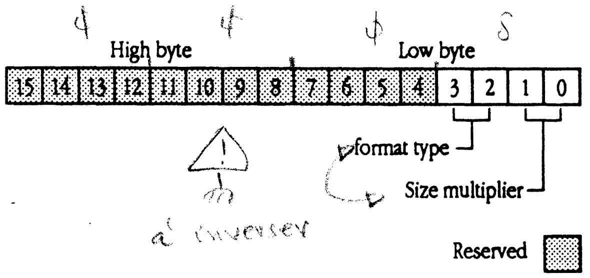
ϕ3 ϕϕ ϕ3 ϕϕ ϕ1 ϕϕ ϕ3 ϕϕ

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

Offset	Field	Size	Description
$00	`formatOptionNum`	Word	The number of this option
$02	`linkRefNum`	Word	Number of linked option
$04	`flags`	Word	(See the 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 which may appear different at the FST level. Linked options are in sets; one of the set is displayed, whereas all others are not, so that the user is not presented with several choices on the initialization dialog. See "Example," later in this section.

Bits within the flags word are defined as follows:



1ØMégas

Ø1 ØØ. Ø2 ØØ. Ø8 ØØ. 5C 52 ØØØØ. ØØ Ø2. Ø1 ØØ. ØA ØØ
Ø2 Ø3 Ø2
Ø3 Ø4 Ø4

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

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

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

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

Example

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

The entire format options list looks like this:

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

GetPartitionMap (DStatus subcall)

Status code = $0004.

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

Device-specific DStatus subcalls

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

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

$202E DControl

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	`trasferCount`	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, as described for each subcall 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 (`bufferPtr`) 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 zero; the driver ignores the control list entirely.

Subcalls

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

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)

Errors

$11 invalid device number
$21 invalid control code
$53 parameter out of range

ResetDevice (DControl subcall)

Control code = $0000.

The Reset Device subcall sets a device's configuration parameters back to their default values. Many GS/OS device drivers contain default configuration settings for each device it controls; see 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 successful, the transfer count for this call is zero. The request count is ignored, and the control list is not used. However, for future compatibility, the `requestCount` parameter should be set to $0.

ok

FormatDevice (DControl subcall)

Control code = $0001.

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

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

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

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

EjectMedium (DControl subcall)

Control code = $0002.

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

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

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

Access not Allowed

SetConfigParameters (DControl subcall)

Control code = $0003.

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

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

The structure of the configuration list is device-dependent. See Chapter 9, "Configuration and Cache Control," for more information.

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

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

SetWaitStatus (DControl subcall)

Control code = $0004.

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

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

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

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

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

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

SetFormatOptions (DControl subcall)

Control code = $0005.

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

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

Offset		Size	Description
$00	formatOptionNum	Word	The number of the format option
$02	interleaveFactor	Word	The override interleave factor (if nonzero)

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

Offset	Size	Description
formatOptionNum	Word	The number of this option
linkRefNum	Word	Number of linked option
flags	Word	File system information
blockCount	Longword	Number of blocks supported by device
blockSize	Word	Block size in bytes
interleaveFactor	Word	Interleave factor (in ratio to 1)
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 (DControl subcall)

Control code = $0006.

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

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

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

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

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

ArmSignal (DControl subcall)

Control code = $0007.

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

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

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

DisarmSignal (DControl subcall)

Control code = $0008.

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

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

SetPartitionMap (DControl subcall)

Status code = $0009.

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

Device-Specific DControl subcalls

Device-specific DControl subcalls are provided to allow device-driver writers to implement control calls specific to individual device drivers' needs. DControl subcalls with `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.

$202F DRead

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 non-block devices, the parameter must be set to zero.

`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 zero 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.

new

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

$2030 DWrite

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	buffer	2	Longword INPUT pointer
$08	requestCount	3	Longword INPUT value
$0C	startingBlock	4	Longword INPUT value
$10	blockSize	5	Word INPUT value
$12	transferCount	6	Longword RESULT value

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

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

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

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

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

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

`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 zero 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

Chapter 2 The SCSI Driver

This chapter describes the GS/OS SCSI driver. The current version of the SCSI driver provides access to both SCSI hard-disk devices and CD-ROM devices. ■

General information

The SCSI Driver is a GS/OS loaded driver that provides direct application access to SCSI devices. It communicates with the firmware on the Apple II SCSI Card and, as such, supports multiple devices. It translates calls from the GS/OS format into the SCSI Card SmartPort format, allowing access to SCSI hard disks and the Apple CD SC drive.

△ **Important** This version of the SCSI driver supports only Revision C of the Apple II SCSI Card. △

The SCSI driver ensures that the Apple CD SC drive stays in 512 byte/block mode.

The SCSI driver provides special handling of CD Audio discs during DRead calls, as follows:

- The Apple CD SC does not allow reading of audio data, and will return an I/O error if attempted. The driver handles this by determining if an I/O error was caused by trying to read audio data and, if so, returns error $28 (no device connected).
- If a read call is issued to the Apple CD SC when it is in play or pause mode, it will stop playing. Because FSTs frequently scan all devices looking for particular volumes, trying to play an audio disc can be frustrating. The driver remedies this problem by checking to see if the Apple CD SC is in play or pause mode and, if so, returns error $28 (no device connected) without issuing the read call to the drive.

Device calls to the SCSI driver

The SCSI driver supports these standard GS/OS device calls:

DInfo
DStatus
DControl
DRead
DWrite

including the standard set of DStatus and DControl subcalls.

The driver also supports additional device-specific DStatus and DControl subcalls. Because the detailed functions and formats of the device-specific DStatus and DControl subcalls are dependent on the device being accessed, and because the SCSI driver accesses CD-ROM devices as well as SCSI hard disk devices, this chapter does not provide all the details on how the device-specific calls work. To fully understand them, you need other documents that describe Apple SCSI commands and Apple CD-ROM commands, such as

- *Apple CD SC Developers Guide*
- ANSI X3.131-1986, *Small Computer System Interface (SCSI)*

You will also need the SCSI Manager chapter in *Inside Macintosh, Volume V*.

The rest of this chapter describes the device-specific DStatus and DControl subcalls. Any device calls or subcalls not discussed here are handled exactly as documented in Chapter 1.

DStatus ($202D)

Please see Chapter 1 of this Volume for a description of the general format of the DStatus call; the SCSI driver supports all standard DStatus subcalls.

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

Offset		Description
$00	`$0000`	Reserved, must be zero
$02	`commandData`	12 bytes of data
$0E	`bufferPtr`	Pointer to a buffer that may contain additional information

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

TestUnitReady (DStatus subcall)

In the DStatus parameter block for this call, `statusCode` = $8000. In the status list, `commandData` contains this information:

byte	**Meaning**
$00	SCSI command: $00
$01	SCSI command flags: $00
$02–0B	(reserved)

The `bufferPtr` parameter is reserved. The call will return an error if the subcall is not successful.

RequestSense (DStatus subcall)

In the DStatus parameter block for this call, `statusCode` = $8003. In the status list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $03
$01	SCSI command flags: $00
$02–0B	(reserved)

Request sense data is returned in the data buffer.

Inquiry (DStatus subcall)

In the DStatus parameter block for this call, `statusCode` = $8012. In the status list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $12
$01	SCSI command flags: $00
$02–0B	(reserved)

Inquiry data is returned in the data buffer.

ModeSense (DStatus subcall)

In the DStatus parameter block for this call, `statusCode` = $801A. In the status list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $1A
$01	SCSI command flags: $00
$02–0B	(reserved)

Mode sense data is returned in the data buffer.

ReadCapacity (DStatus subcall)

Capacité totale du device = Σ partitions

In the DStatus parameter block for this call, `statusCode` = $8025. In the status list, `commandData` contains this information:

$Φ1 52 AE

byte	Meaning
$00	SCSI command: $25
$01	SCSI command flags: $00
$02–0B	(reserved)

Capacity data is returned in the data buffer.

Verify (DStatus subcall)

In the DStatus parameter block for this call, `statusCode` = $802F. In the status list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $2F
$01	SCSI command flags: $00
$02–05	block number to start verify, msb first
$06–07	number of contiguous blocks to verify, msb first
$08–0B	(reserved)

The `bufferPtr` parameter is reserved. The call will return an error if the subcall is not successful.

p. count ΦΦΦ5
dev. num ΦΦΦ9
Status. code 8Φ2F
Status. list Adress
req. count E
Tx. count E

ΦΦ ΦΦ 2F ΦΦ
ΦΦ ΦΦ ΦΦ ΦΦ ← départ (block #)
ΦΦ 1Φ # block
ΦΦ ΦΦ ΦΦ ΦΦ
Adress2

ΦΦ FF XX ΦΦ = 1Φ2

ReadTOC (DStatus subcall)

This subcall applies to CD-ROM only.

In the DStatus parameter block for this call, `statusCode` = $80C1. In the status list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $C1
$01	SCSI command flags: $00
$02	track number
$03–04	(reserved)
$05	TOC type: $00 = type 0 $40 = type 1 $80 = type 2
$06–0B	(reserved)

TOC data is returned in the data buffer.

ReadQSubcode (DStatus subcall)

This subcall applies to CD-ROM only.

In the DStatus parameter block for this call, `statusCode` = $80C2. In the status list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $C2
$01	SCSI command flags: $00
$02–0B	(reserved)

Q subcode data is returned in the data buffer.

ReadHeader (DStatus subcall)

This subcall applies to CD-ROM only.

In the DStatus parameter block for this call, `statusCode` = $80C3. In the status list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $C3
$01	SCSI command flags: $00
$02–05	block address, msb first
$06–0B	(reserved)

Header data is returned in the data buffer.

AudioStatus (DStatus subcall)

This subcall applies to CD-ROM only.

In the DStatus parameter block for this call, `statusCode` = $80CC. In the status list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $CC
$01	SCSI command flags: $00
$02–0B	(reserved)

Audio status data is returned in the data buffer.

DControl ($202E)

Please see Chapter 1 of this Volume for a description of the general format of the DControl call; the SCSI driver supports all standard DControl subcalls.

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

Offset		Description
$00	`$0000`	Reserved, must be zero
$02	`commandData`	12 bytes of data
$0E	`bufferPtr`	pointer to a buffer that may contain additional information

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

RezeroUnit (DControl subcall)

In the DControl parameter block for this call, `controlCode` = $8001. In the control list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $01
$01	SCSI command flags: $00
$02–0B	(reserved)

The data buffer is reserved.

ModeSelect (DControl subcall)

In the DControl parameter block for this call, `controlCode` = $8015. In the control list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $15
$01	SCSI command flags: $00
$02–0B	(reserved)

The data buffer contains the mode-select data to be sent.

Start/StopUnit (DControl subcall)

In the DControl parameter block for this call, `controlCode` = $801B. In the control list, `commandData` contains this information:

byte	Meaning	
$00	SCSI command: $1B	
$01	SCSI command flags: $00	
$02–$03	(reserved)	
$04	start/stop flag:	$00 = stop
		$01 = start
$05–0B	(reserved)	

The data buffer is reserved.

Prevent/AllowRemoval (DControl subcall)

In the DControl parameter block for this call, `controlCode` = $801E. In the control list, `commandData` contains this information:

byte	Meaning	
$00	SCSI command: $1E	
$01	SCSI command flags: $00	
$02–$03	(reserved)	
$04	prevent/allow flag:	$00 = allow
		$01 = prevent
$05–0B	(reserved)	

The data buffer is reserved.

- *Eject call:* On an Eject call (control code = $0002), the SCSI driver always first issues an Allow Removal call before ejecting the disk; any preexisting prevent-removal condition is therefore disabled. If you want prevent-removal to be enabled after ejection, reissue the Prevent Removal call.

Seek (DControl subcall)

In the DControl parameter block for this call, `controlCode` = $802B. In the control list, `commandData` contains this information:

byte	Meaning
$00	SCSI command: $2B
$01	SCSI command flags: $00
$02–05	seek block number, msb first
$06–0B	(reserved)

The data buffer is reserved.

AudioSearch (DControl subcall)

This subcall applies to CD-ROM only.

In the DControl parameter block for this call, `controlCode` = $80C8. In the control list, `commandData` contains this information:

byte	Meaning	
$00	SCSI command:	$C8
$01	SCSI command flags:	$00
$02	play flag:	$00 = pause after search complete
		$10 = play after search complete
$03	play mode:	$00–0F
$04–07	search address, msb first	
$08	address type:	
		$00 = type 0
		$40 = type 1
		$80 = type 2
$09–0B	(reserved)	

The `bufferPtr` parameter is reserved.

AudioPlay (DControl subcall)

This subcall applies to CD-ROM only.

In the DControl parameter block for this call, `controlCode` = $80C9. In the control list, `commandData` contains this information:

byte	Meaning	
$00	SCSI command:	$C9
$01	SCSI command flags:	$00
$02	stop flag:	$00 = playback address is start address for play
		$10 = playback address is stop address for play
$03	play mode:	$00–0F
$04–07	playback address, msb first.	
$08	address type:	
		$00 = type 0
		$40 = type 1
		$80 = type 2
$09–0B	(reserved)	

The `bufferPtr` parameter is reserved. The call will return an error if the subcall is not successful.

AudioPause (DControl subcall)

This subcall applies to CD-ROM only.

In the DControl parameter block for this call, `controlCode` = $80CA. In the control list, `commandData` contains this information:

byte	Meaning	
$00	SCSI command:	$CA
$01	SCSI command flags:	$00
$02	pause flag:	$00 = release pause
		$40 = start pause
$03–0B	(reserved)	

The `bufferPtr` parameter is reserved. The call will return an error if the subcall is not successful.

AudioStop (DControl subcall)

This subcall applies to CD-ROM only.

In the DControl parameter block for this call, `controlCode` = $80CB. In the control list, `commandData` contains this information:

byte	Meaning	
$00	SCSI command:	$CB
$01	SCSI command flags:	$00
$02-05	stop address, msb first	
$06	address type:	
		$00 = type 0
		$40 = type 1
		$80 = type 2
$07–0B	(reserved)	

The `bufferPtr` parameter is reserved. The call will return an error if the subcall is not successful.

AudioScan (DControl subcall)

This subcall applies to CD-ROM only.

In the DControl parameter block for this call, `controlCode` = $80CD. In the control list, `commandData` contains this information:

byte	Meaning	
$00	SCSI command:	$CD
$01	SCSI command flags:	$00
$02	direction flag:	$00 = fast forward
		$40 = fast reverse
$03	(reserved)	
$04-07	scan starting address, msb first.	
$08	address type:	
		$00 = type 0
		$40 = type 1
		$80 = type 2
$09–0B	(reserved)	

The `bufferPtr` parameter is reserved. The call will return an error if the subcall is not successful.

Chapter 3 **The AppleDisk 3.5 Driver**

The Apple 3.5 drive is a block device that can read 3.5-inch disks in formats compatible with the ProDOS or Macintosh file systems, and connects directly to the Apple IIGS disk port.

This chapter describes the GS/OS AppleDisk 3.5 driver, a GS/OS loaded driver that controls the Apple 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. ■

General information

The Apple 3.5 drive is a block device that reads and writes 3.5-inch disks and can handle several types of disk formats, including those used by the ProDOS file system and the Macintosh file systems. Although the Apple 3.5 drive is not an intelligent drive—it cannot interpret software command streams—its controller is accessed through SmartPort firmware, and recognizes a set of device-specific extended SmartPort Control commands. The drive connects directly to the Apple IIGS disk port or to a SmartPort-compatible expansion card in a slot. See the *Apple IIGS Firmware Reference* for more information.

The AppleDisk 3.5 driver is a loaded driver that uses the SmartPort firmware protocol to support one or two Apple 3.5 drives. The AppleDisk 3.5 driver operates independently of the system speed. The driver supports a variety of formatting options: 400 KB or 800 KB disks, and either 2:1 or 4:1 interleave.

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

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 suppports this standard set of DStatus subcalls:

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

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 conditions, see the DriverStatus call in Chapter 11, "GS/OS Driver Call Reference."

GetConfigParameters:

The AppleDisk 3.5 driver has no parameters in its configuration parameter list and returns with a status list length word of zero 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 is option 1
	$0000	Current media formatting is 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 kilobytes
	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 kilobytes
	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 kilobytes

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	EjectMedia
$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. The AppleDisk 3.5 driver does not support overriding interleave factors and must have `interleaveFactor` set to $0000.

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 or $020C (512 or 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 or $020C (512 or 524) bytes per block.

Chapter 4 The UniDisk 3.5 Driver

The UniDisk 3.5 drive is a block device that can read 3.5-inch disks in formats compatible with the ProDOS or Macintosh file systems, and connects directly to the Apple IIGS 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. ■

General information

The UniDisk 3.5 drive is a block device that reads and writes 3.5-inch disks and can handle several types of disk formats, including those used by the ProDOS file system and the Macintosh file systems. It is an intelligent device that supports standard SmartPort protocols. The drive connects directly to the Apple IIGS disk port or to a SmartPort-compatible expansion card in a slot. See the *Apple IIGS Firmware Reference* for more information.

The UniDisk 3.5 driver is a loaded driver that supports up to four total UniDisk 3.5 drives on the diskport.

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

The UniDisk 3.5 driver operates independent of the system speed. The driver supports a variety of formatting options: 400 KB or 800 KB disks, and either 2:1 or 4:1 interleave.

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

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 conditions, see the DriverStatus call in Chapter 11, "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 (DControl) SetFormatOptions subcall prior to issuing a (DControl) Format subcall to a block device.

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

`transferCount`	$0000 0038	(56 bytes returned in list)
`statusList`		*Options list header:*
	$0001	Three options in list
	$0001	One displayed option
	$0001	Default is 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 status 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.

SetWait Mode:

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 earlier in this chapter, under the DStatus subcall GetFormatOptions. Any one of those 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 or $020C (512 or 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 or $020C (512 or 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. This chapter describes how the the AppleDisk 5.25 driver works and what device calls it accepts. It also describes the the physical and logical formats used by the AppleDisk 5.25 driver on 5.25-inch media. ■

◆ 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.

General information

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 IWM interface. The AppleDisk 5.25 driver functions independently of the system speed and does not have the resident slot limitation inherent in the Apple IIGS. This means that, although the Apple IIGS 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 to operate at accelerated speed in all slots (1 through 7), with either one or two Apple 5.25 drives per slot.

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 file system translator (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

The rest of this chapter describes how the AppleDisk 5.25 driver handles any of the above device calls differently from 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 from the standard descriptions given in Chapter 1 of this Volume.

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 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 zero 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 is option 1
	$0000	Formatting option of current media is unknown

	Option-entry 1:
$0001	Option 1
$0002	This option is 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	EjectMedia
$0003	SetConfigParameters
$0004	SetWaitStatus
$0005	SetFormatOptions
$0006	AssignPartitionOwner
$0007	ArmEvent
$0008	DisarmEvent

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.

EjectMedia:

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 subcall

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 a block size of 256 bytes (for DOS 3.3) or 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.

◆ *Disk-switched detection:* 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 directly accessing the AppleDisk 5.25, 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 a block size of 256 bytes (for DOS 3.3) or 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.

◆ *Disk-switched detection:* 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 directly accessing the AppleDisk 5.25, the application has to handle the disk-switched error. The normal procedure is to retry once and only once.

AppleDisk 5.25 formatting

The AppleDisk 5.25 driver supports only 35-track, 16-sector formatting. Media is 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.

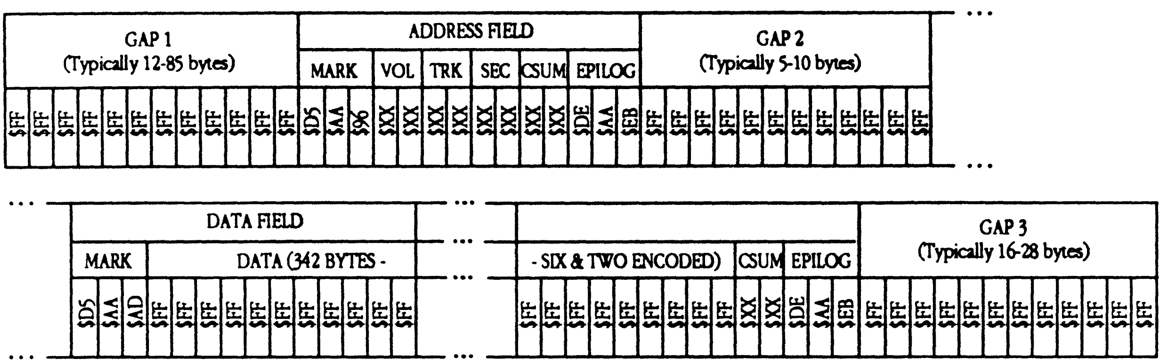
Logical-to-physical sector translations are shown in the interleave translation tables of Figure 5-1. The input block size to a media access call controls which translation table is used.

■ **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 (epilogue) markers.

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



Chapter 6 **The Console Driver**

The **console** is a conceptual component of a computer system; it consists of the principal conduits by which the computer's operator sends commands to the computer and receives messages from the computer. On the Apple IIGS, like most personal computers, the console consists of the keyboard (for input) and the video display screen (for output).

The GS/OS console driver is a loaded driver that allows sophisticated manipulation of the Apple IIGS text page. It runs in both 40-column and 80-column mode. The console driver supports many advanced features, while using the standard Apple II BASIC and Pascal control codes. ■

◆ *Text mode only:* The console driver is for use only by applications that run in text mode. The console driver does not support the standard Apple II Hi-Res or double Hi-Res graphics. If your application uses the Apple IIGS super Hi-Res graphics screen, it writes to the screen with toolbox calls. See the *Apple IIGS Toolbox Reference.*

General information

The GS/OS console driver allows an application to treat both parts of the console (keyboard and screen) as a single device that can be read from or written to. Because the console has two parts, the console driver does also: an input routine and an output routine (see Figure 6-1):

Console output:

The Console Output routine writes to the screen. It supports uppercase, lowercase, inverse, and MouseText characters. It also includes a suite of control characters with functions such as any-direction scrolling, character-set selection, and cursor control. Finally, it permits saving of areas of the screen to off-screen buffers, and selectively saving and restoring text port parameters—in effect, allowing a simple windowing system.

All commands to the Console Output routine are sent as control characters. This allows the programmer to create strings of commands that will be executed one after another, but requiring only a single write call. All operations occur in a rectangular subset of the hardware screen known as the **text port**. All text outside the text port is protected; that is, that text will not be affected by any console calls.

Console input:

The Console Input routine accepts characters from the keyboard. There are two basic input modes: raw mode allows for simple keyboard input, whereas a more advanced user-input mode allows for text-line editing and application-defined terminator keys. User-input mode also supports features such as no-wait mode (which allows an application to continue running while input is pending) and interrupt keys (which allow application-defined editing keystrokes, such as using arrow keys to change a setting or using a key combination—like Apple - ?—to bring up a help screen).

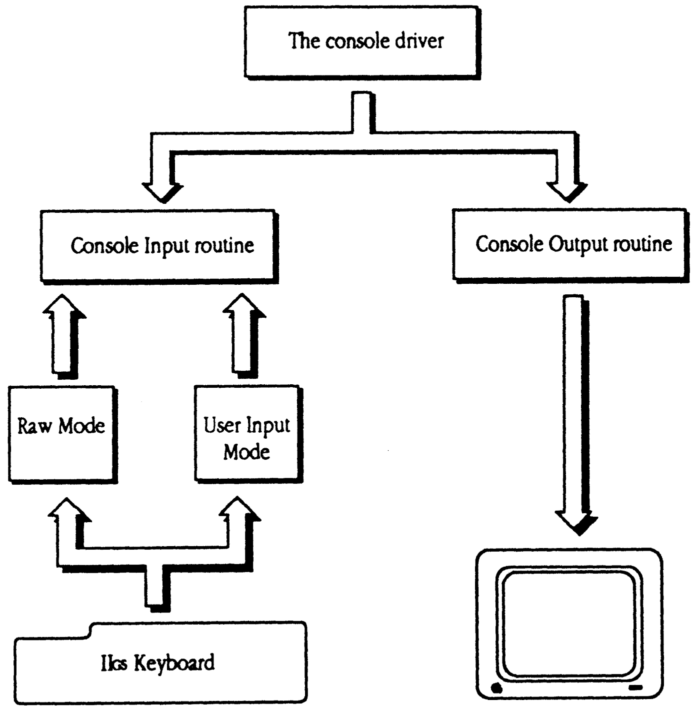
The application can supply a default string to the user input mode. If the default string contains more characters than the width of the input field, the extra characters are retained; however, they are displayed only if characters are deleted from the visible part of the field. Horizontal scrolling of the input field is not supported.

The application can also specify options such as overstrike or insert mode on entry. A flashing block cursor signifies overstrike mode; a flashing underline cursor specifies insertion. The cursor flash rate is based on the current control panel settings.

The user can insert control characters into the input string by pressing Apple-Control-*character*, where *character* is replaced by any keyboard character. Control characters are displayed on the screen in inverse, but are returned in the input string as codes from $00 to $1F. All normal ASCII characters are returned in the range $20–$7F.

The terminators used by the Console Input routine are more advanced than the newline characters specified in GS/OS (see the description of the Newline call in Chapter 7 of Volume 1). User-input specified terminators can include not only ASCII codes for the terminator characters, but also the keyboard modifier bits. For example, the Return and Enter keys could be given different functions by separately specifying terminators, one with the keypad flag set and one with it clear.

■ **Figure 6-1** Console driver I/O routines



The Console Output routine

The Console Output routine handles writing to the screen. It supports different screen sizes and defines subareas of the screen called **text ports,** which can be used to protect parts of the screen. All commands to the Console Output routine are sent as control characters.

Screen size

The default screen size (in columns of width) is always 80 columns. You can change the screen size by the writing the correct screen control code, as described in the section "Screen Control Codes" later in this chapter.

- The 40-column screen consists of 40 columns of text in 24 lines. The upper-left corner is 0,0 and the lower-right corner is 39,23.
- The 80-column screen consists of 80 columns of text in 24 lines. The upper-left corner is 0,0 and the lower-right corner is 79,23.

The text port

The driver maintains an active text port in which all activity occurs. The default size of this text port is the entire screen. However, subsequent calls can be made to resize the port. All text outside the text port is protected—no console driver calls can affect that text.

Two control commands allow the application to save the current text port (the port definitions, not the actual text of the port) and start with a new one, and then to retrieve the original port. This allows a simple windowing system. In addition, driver-specific control calls allow the application to read the text port data structure; however, the values in the data structure can only be changed with control commands (see the section "Screen Control Codes" later in this chapter). This is the structure of the text port record:

```
TextPortRec = {
        byte    ch,
                cv,
                windLeft,
                windTop,
                windRight,
                windBottom,
                windWidth,
                windLength,
                consWrap,
                consAdvance,
                consLF,
                consScroll,
                consVideo,
                consDLE,
                consMouse,
                consFill        }
```

Here are the definitions for the fields:

`ch` `cv`	The current location of the cursor (horizontal and vertical, from the upper-left corner). The cursor is always within the current text port, but is expressed in absolute screen coordinates. Default = 0, 0

`windLeft`
`windTop`
`windRight`
`windBottom`
	Boundaries of the current text port, in absolute screen coordinates.
	`windTop` must be <= `windBottom`, and
	`windLeft` must be <= `windRight`.
	Default = Full Hardware Screen.

`windWidth` `windLength`	Size of the current text port, calculated as follows: windWidth = windLeft – windRight + 1 windLength = windTop – windBottom + 1. Default = Full Hardware Screen

`consWrap`	A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, the cursor wraps to the first column of the next line after printing in the rightmost column. Default = TRUE
`consAdvance`	A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, the cursor moves one space to the right after printing. Default = TRUE
`consLF`	A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. Carriage return characters always move the cursor to the first column of the text port. If `consLF` is TRUE, the cursor will also move to the next line (note that this could cause a scroll—see next flag). Default = TRUE
`consScroll`	A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, the screen will scroll if moved past the top or bottom of the screen. Default = TRUE
`consVideo`	A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, output is displayed in normal video. If FALSE, output is displayed in inverse video. Default = TRUE
`consDLE`	A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, character $10 (DLE) is interpreted as a space expansion character; when it is encountered in the input stream, the ASCII value of the next character minus 32 becomes the number of spaces to output. Default = TRUE
`consMouse`	A Boolean flag: 0 = FALSE, 128 ($80) = TRUE. If TRUE, MouseText is turned on. When MouseText is on, inverse uppercase characters are displayed as MouseText. Default = FALSE
`consFill`	This is the fill character used for clearing areas of the screen. It is an actual screen byte—the value of the character as stored in memory—so the high-order bit must be turned on for normal display. For example, spaces (ASCII $20) should be specified by $A0. The value in this field is altered whenever inverse or normal modes are selected. Default = $A0 (Screen Space)

Character set mapping

Output characters go through a number of stages before they are placed in screen memory and appear on the screen. The console driver always uses the Apple IIGS alternate character set, which includes uppercase and lowercase characters, punctuation, numbers, inverse characters, and MouseText characters.

Normally, the console driver accepts input in the standard, "7-bit" ASCII range (00–$7F). This input is then mapped to the screen based on the current display mode and MouseText mode settings (turned on or off through control codes in the character stream; see the section "Screen Control Codes" later in this chapter). In addition, input ASCII in the range $80–$FF is mapped directly to the inverse of whatever the current display mode is. Thus screen bytes (characters as stored in screen memory) may have values quite different from their original input ASCII values.

Table 6-1 summarizes the output mapping. For both normal and inverse display modes, and with MouseText mapping both enabled and disabled, the table compares input ASCII values with the characters as displayed on the screen and with the equivalent values as stored in screen memory. The table also shows that setting the high-order bit (special direct inverse mode) is a shortcut to getting the inverse of the current mode.

◆ *Mapping is nonsequential:* Note from Table 6-1 that in some cases sequential ASCII values in the input stream (such as $3F and $40) may map to nonsequential values in screen memory (such as $BF and $80, respectively). Specifically, the range of values interpreted as uppercase characters may not be continuous with the ranges interpreted as special characters and lowercase characters. If your application retrieves bytes directly from screen memory, it may have to compensate for this.

■ **Table 6-1** Console driver character mapping
All numbers are hexadecimal

	Normal display mode		**Inverse display mode**	
MouseText disabled:				
Input values	*As displayed*	*As stored*	*As displayed*	*As stored*
00–1F	Control characters	n/a	Control characters	n/a
20–3F	Special characters	A0–BF	Inverse special	20–3F
40–5F	Uppercase letters	80–9F	Inverse upper	00–1F
60–7F	Lowercase letters	E0–FF	Inverse lowercase	60–7F
MouseText enabled:				
Input values	*As displayed*	*As stored*	*As displayed*	*As stored*
00–1F	Control characters	n/a	Control characters	n/a
20–3F	Special characters	A0–BF	Inverse special	20–3F
40–5F	Uppercase letters	80–9F	MouseText characters	40–5F
60–7F	Lowercase letters	E0–FF	Inverse lowercase	60–7F
Special direct inverse mode:				
Input values	*As displayed*	*As stored*	*As displayed*	*As stored*
80–9F	Uppercase inverse	00–1F	Uppercase normal	80–9F
A0–BF	Inverse special	20–3F	Special chars normal mode	A0–BF
C0–DF	MouseText characters	40–5F	Uppercase normal	C0–DF
E0–FF	Inverse lower	60–7F	Lowercase normal	E0–FF

Screen control codes

In any mode, values from $00 to $1F are interpreted as control codes. Some control codes are one-byte commands; others use from two to four bytes of operands, which follow the control character. If an output stream ends in the middle of a multibyte sequence, the console driver simply uses the first bytes of the next output stream. The actual command is not executed until the entire command string has been read. Here are the defined control codes:

$00 **Null**

No operation is performed.

$01 **Save Current Text Port and Reset Default Text Port**

Saves the current text port and resets to the default text port. If the system is out of memory, no error is returned, and the text port is simply reset.

$02 **Set Text Port Size**

Accepts the next four bytes as absolute screen coordinates + 32. Sets the current text port to the new parameters. The parameters are in the following order: `windLeft`, `windTop`, `windRight`, `windBottom`. Any parameter outside of the physical screen boundaries is clipped to a legal value. The cursor is set to the upper-left corner of the new text port.

$03 **Clear from Beginning of Line**

Clears all characters from the left edge to and including the cursor. Sets them to the current `consFill` character.

$04 **Pop Text Port**

Restores the text port to the most recently saved value (see code $01, Push and Reset Text Port). If no saved ports exist, resets the text port to the default values. If an 80-column text port is pushed and subsequently restored in 40-column mode, the text port may be altered to fit in the 40-column screen (see code $11, Set 40-Column Mode).

$05 **Horizontal Scroll**

Interprets the next byte as an 8-bit signed integer depicting the number (*N*) of columns to shift. N of zero is a null operation. If N is less than zero, the text port is shifted to the left; N greater than zero shifts to the right. If the shift magnitude is equal to or greater than `windWidth`, the text port is cleared.

The shifted characters are moved directly to their destination location. The space vacated by the shifted characters is set to the current fill character (see `consFill`). Characters shifted out of the text port are removed from the screen and are not recoverable.

$06 **Set Vertical Position**

Interprets the next byte as a textport-relative vertical position + 32. If the destination is outside the current textport, the cursor is moved to the nearest edge.

$07 **Ring Bell**

Causes the System Beep to be played. It has no effect on the screen.

$08 **Backspace**

Moves the cursor one position to the left. If the cursor was on the left edge of the text port and `consWrap` is TRUE, the cursor is placed one row higher and at the right edge. If the cursor was also on the topmost row and `consScroll` is TRUE, the text port will scroll backwards one line.

$09 **Tab (no operation)**

This command is ignored.

$0A **Line Feed**

Causes the cursor to move down one line. If at the bottom edge of the text port and `consScroll` is TRUE, the text port scrolls up one line.

$0B **Clear to End of Text Port**

Clears all characters from the cursor to the end of the current text port to the current `consFill` character.

$0C **Clear Text Port and Home Cursor**

Clears the entire text port and resets the cursor to `windLeft`, `windTop`.

$0D **Carriage Return**

Resets the cursor to the left edge of the text port; if `consLF` is TRUE, performs a line feed (see $0A line feed).

$0E **Set Normal Display Mode**

After this character, it displays all subsequent characters in normal mode.

$0F **Set Inverse Display Mode**

After this character, it displays all subsequent characters in inverse mode.

$10 **DLE Space Expansion**

If `consDLE` is TRUE, it interprets the next character as number of spaces + 32, and the correct number of spaces is issued to the screen. If `consDLE` is FALSE, the DLE character is ignored and the following character is processed normally.

$11 **Set 40-Column Mode**

Sets the screen hardware for 40-column display. If changing from 80-column display, copies the first 40 columns of the 80-column display into the 40-column display.

If the current text port does not fit in the 40-column screen, it is adjusted by one of two methods:

- If the text port is 40 columns or narrower, the entire text port (left side, right side, and cursor) is slid over until the right edge is collinear with the right edge of the screen.
- If 41 columns or wider, the port becomes 40 columns and the cursor moves to the left edge.

$12 **Set 80-Column Mode**

Sets the screen hardware for 80-column display. If changing from 40-column display, copies the 40-column data to the left half of the 80-column display and clears the right half of the screen to the `consFill` character.

$13 **Clear from Beginning of Text Port**

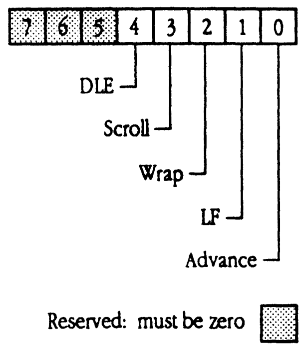
Clears all characters from the beginning of the text port to and including the cursor location.

$14 **Set Horizontal Position**

Interprets the next byte as a textport–relative horizontal position + 32. If the destination is outside the current textport, the cursor is moved to the nearest edge.

$15 **Set Cursor Movement Word**

Interprets the next byte as cursor movement control. It sets the values of these Boolean flags:



The functions of the individual flags are described under the section "The Text Port" earlier in this chapter.

$16 **Scroll Down One Line**

Scrolls the text port down one line. Does not move the cursor.

$17 **Scroll Up One Line**

Scrolls the text port up one line. Does not move the cursor.

$18 **Disable MouseText Mapping**

When MouseText is disabled, uppercase inverse characters are displayed as such (see the section "Character Set Mapping" earlier in this chapter).

$19 **Home Cursor**

Resets the cursor to the upper-left corner of the text port.

$1A **Clear Line**

Clears the line that the cursor is on. Resets the cursor to the leftmost column in the window.

$1B **Enable MouseText Mapping**

When MouseText is enabled, uppercase inverse letters are instead displayed as MouseText symbols (see the section "Character Set Mapping" earlier in this chapter).

$1C **Move Cursor Right**

Performs a nondestructive forward–space of the cursor. If `consWrap` is TRUE, the cursor might go to the next line; and if `consScroll` is TRUE, the screen might scroll up one line.

$1D **Clear to End of Line**

Clears from the position underneath the cursor to the end of the current line.

$1E **Go to X,Y**

Adjusts the cursor position relative to the text port. The parameters passed are X+32 and Y+32. If the new locations are outside the current text port, the cursor is placed on the nearest edge.

$1F **Move Cursor Up**

Moves the cursor up one line (reverse line feed). If the cursor is already on the uppermost line of the text port and `consScroll` is TRUE, it will cause a reverse scroll.

The Console Input routine

The console driver's Console Input routine, especially in user input mode, provides a convenient method for obtaining user input. It is best suited for fixed-field, fill-in-the-blanks type of input with simple line-editing commands and program–defined default strings.

The console driver obtains input directly from the keyboard hardware, or from the Apple IIGS Toolbox Event Manager if it is active. The Console Input routine monitors not only the keystroke but the modifier keys (shift, control, option, and so on) and can make decisions based on both the keystroke and the current modifiers.

The input port

All information about the current input is contained in the **input port,** a data structure that is maintained by the Console Input routine but can be read, modified, and written back by the application program. The data structure is as follows:

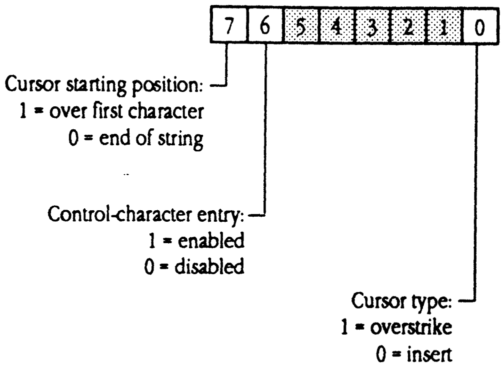
```
InputPortRec = {
       byte   fillChar,
              defCursor,
              cursorMode,
              beepFlag,
              entryType,
              exitType,
              lastChar,
              lastMod,
              lastTermChar,
              lastTermMod,
              cursorPos,
              inputLength,
              inputField,
              originH,
              originX, (word)
              originV       }
```

The meanings of each field are as follows:

`fillChar` The character that fills empty space in the input field. It is displayed by the Console Output routine so it is usually $20 (normal space) (see the section "Character Set Mapping" earlier in this chapter). One other useful fill character is the MouseText "ghost space" character. This can be displayed by setting `fillChar` to $C9. However, since MouseText characters are only available in normal mode, do not use MouseText fill characters when the screen is in inverse mode.
Default = Space ($20)

`defCursor` The default cursor-mode setting. The value in this field is placed into the `cursorMode` field at the beginning of an input cycle from the user. The application controls the cursor mode the user starts with by controlling this setting.
Default = $80 (cursor starts at end of string, control-character entry disabled, cursor type = insert).

`cursorMode` Contains three status bits that describe the current cursor-mode setting:



If control-character entry is enabled, the user can insert control characters into the stream by typing -Control-*character*, where *character* is replaced by any valid keyboard character.

The value in this field may be different from `defCursorMode` because the user can switch between insert and overstrike modes during entry.

`beepFlag` If this flag is nonzero, the Console Input routine beeps on input errors (line too long, and so on).
Default = TRUE

`entryType` Tells the Console Input routine the status of the current input:
0=initial entry
1=interrupt reentry
2=no-wait mode reentry
On exit, the Console Input routine adjusts this value so that it is correct for the next entry. If the application wishes to cancel an in-progress input and start with a new one, it must make the DConbtrol subcall Abort Input.
Default = initial entry

`exitType` Tells the application which type of exit was made. (0 = input not terminated yet, either because of end-of-field on Raw input or a no-wait exit.) Any other value is the number of the terminator that halted the input.
(Set on exit from the input cycle.)

`lastChar` — The ASCII value ($00–$7F) of the most recently typed key.
(Set on exit from the input cycle.)

`lastMod` — The value of the **modifiers mask** of the most recently typed key. See the section "Terminators" later in this chapter, for a description of the modifier bits.
(Set on exit from the input cycle.)

`lastTermChar` — The ASCII value of the terminator (as specified in the the user-supplied terminator list) that caused the most recent input termination.
(Set on exit from the input cycle.)

`lastTermMod` — The value of the **modifiers mask** of the terminator that caused the most recent input termination.
(Set on exit from the input cycle.)

`cursorPos` — Index of the cursor within the input string. (0=over the first character.) The cursor is allowed to move from the beginning of the string to one position past its end.
Default = position of cursor when input begins

`inputLength` — The length of the input string at the current state of editing. This is the length that is returned in the Transfer Count.
Default = length of default input string

`originH` — Contains the cursor's horizontal position.

`originX (word)` — Contains a variable used by the UIR.

`originH` — Contains the cursor's vertical position.

Using raw mode

Raw mode is the simplest form of user input. The keyboard is simply scanned until (1) `requestCount` number of keys have been pressed, or (2) A specified terminator has been typed. As with other serial input drivers, the terminator is included in the transferred string. There is no echo, no cursor, and no editing.

Using user input mode

This input mode provides more functions than raw mode. The following steps are required to use it:

1. If the application wishes to supply a default string, it must do so (see descriptions of the Control subcalls, later in this chapter).
2. If modes other than the default modes are desired, the application should read the input port, adjust it, and write it back.
3. Terminators must be assigned with a SetTerminators call (DControl subcall).
4. The cursor should be positioned to the desired start of the input field with a Go To X,Y instruction.

A Read call is made to initiate user input mode. If only simple terminators have been requested, the Console Input routine will return as soon as one has been pressed. If there are interrupt terminators or if no-wait mode is selected, the application must make calls to determine the type of interruption and determine whether more work (repeated read entries) is necessary.

Terminators

A **terminator** is a character that, when read, terminates or interrupts a Read call. The console driver permits more than one terminator character and also can note the state of modifier keys in considering whether a character is to be interpreted as a terminator.

The console driver keeps track of terminators with a **terminator list.** The terminator list is set using a control call (see the Control subcalls, later in this chapter). This is the format of a terminator list:

```
TermList = {
        word    termMask,
                termCount,
                termList [ 1 ... termCount ]
        }
```

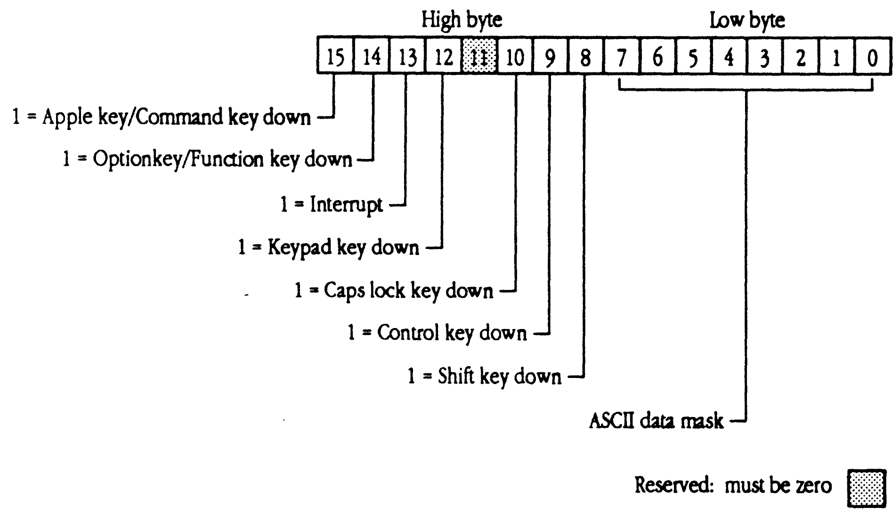
The fields have the following meanings:

`termMask` — A mask that is added to the input data with an AND operator before it is compared to the terminator list entries. The high-order byte is the **modifiers mask;** it is used to mask out irrelevant modifiers (for example, if it doesn't matter whether the keystroke was made from the main keyboard or the keypad). The low-order byte is the **ASCII mask;** it is used to simplify ASCII comparisons (for example, if it doesn't matter whether a character is uppercase or lowercase).

`termCount` — A count of the number of terminators. A count of 0 means terminators are disabled and there is no list. It specifies the number of entries, so it must be multiplied by two to get a byte count. The maximum terminator count is 254.

`termList` — A list of terminator characters and their modifiers. Each entry is in the same format as `termMask`; the high byte is the modifiers mask, and the low byte is the ASCII value of the terminator character. After the incoming data is combined with the terminator mask in a logical AND operation, the data is compared with each of the entries in the terminator list. A match causes a termination. In addition, if the application supplies a term list entry with bit 13 set, this is an interrupt terminator. The Console Input routine will give up control but is set up to restart the input. The application can use this capability to implement help screens or custom editing keys.

The terminator mask has the following format:



How to disable terminators

The application can disable terminators by doing either of the following:

- Set the mask to 0.
- Set the count to 0.

In addition, if a memory error occurs while new terminators are being received, the UIR dumps the terminator list.

If an incorrectly formed list (for example, if count = 255) is sent to the Console Input routine, it is discarded and the original terminators remain in place.

Terminators and newline mode

Newline characters as defined by the Character FST are incompatible with terminators as defined by the console driver's user input mode. If you need a combined newline/termination mode, use only the following combinations:

Character FST	Console driver
Newline mode enabled	Raw Input mode, terminators disabled
Newline mode disabled	Raw Input mode, terminators enabled
Newline mode disabled	User Input modes

User-input editing commands

The following editing commands are supported by the Console user input mode:

← *or* Control-H	Move cursor backward one position
→ *or* Control-U	Move cursor forward one position
-→	Move cursor to end of next word.
-←	Move cursor to beginning of previous word.

-> *or* -.	Move cursor to end of line.
-< *or* -,	Move cursor to beginning of line.
Delete *or* Control-D *or* Control-Delete *or* -Delete *or* -D	Delete character to left of cursor and move cursor and string to left (destructive backspace).
Control-F *or* -F	Delete the character underneath the cursor and move the rest of the string to the left.
Control-X *or* -X *or* Clear	Delete entire input string.
Control-Y *or* -Y	Clear string from cursor to end.
Control-Z *or* -Z	Reset input string to application-specified default.
Control-E *or* -E	Toggle between insertion and overstrike characters.
-Control-*character*	Insert control character into input string (if enabled; control-character insertion is enabled by setting a bit in `cursorMode`).

Using no-wait mode

No-wait mode is defined so that drivers will not hold control of the system. When in wait mode, a Read call does not terminate until the requested number of characters (or a terminator) is received. When in no-wait mode, the system returns immediately from a Read call as soon as there is no more input available. In such a case, it is the responsibility of the application program to continue calling the input routines until the final number of characters have been transferred.

Device calls to the console driver

The GS/OS console driver supports the standard set of device calls:

DInfo
DStatus
DControl
DRead
DWrite

The standard calls are described in Chapter 1 of this Volume. The rest of this chapter documents the driver-specific DStatus and DControl subcalls, and describes how the console driver handles any of the standard device calls differently from the ways documented in Chapter 1. Any calls or subcalls not discussed here are handled exactly as documented in Chapter 1.

DStatus ($202D)

This call is used to request status information from the console driver. For DStatus, the console driver supports most of the standard subcalls and several device-specific subcalls. Status subcalls are specified by the value of the status code parameter. The following status codes are supported:

Status code	**Subcall name**
$0000	GetDeviceStatus
$0001	GetConfigParameters
$0002	GetWaitStatus
$8000	GetTextPort
$8001	GetInputPort
$8002	GetTerminators
$8003	SaveTextPort
$8004	GetScreenChar
$8005	GetReadMode
$8006	GetDefaultString

Calls with status codes less than $8000 are standard Status subcalls; calls with status codes of $8000 and over are device-specific subcalls. The calls are described more fully in the following sections.

Standard DStatus subcalls

Standard DStatus subcalls that are not described here function exactly as documented in Chapter 1, "GS/OS Device Call Reference."

GetConfigParameters:

The console driver obtains its setup information from battery RAM and therefore uses no control parameters. This call returns an empty control parameter record (a zero).

The minimum request count is 2. The maximum transfer count is 2.

GetTextPort (DStatus subcall)

`statusCode` = $8000

status list = a text port record

This subcall copies the contents of the current text port record into the status list buffer. See the section "The Text Port" earlier in this chapter for more details.

The minimum request count is 0. The maximum transfer count is 16.

GetInputPort (DStatus subcall)

`statusCode` = $8001.

status list = input port record

This subcall copies the contents of the current input port record into the status list buffer. See the input port description earlier in this chapter for more details.

The minimum request count is 0. The maximum transfer count is 12.

GetTerminators (DStatus subcall)

`statusCode` = $8002

status list = terminator list record

This subcall copies the current terminator list into the status list buffer. The format of the list is count, enable/mask, terminator list. See the section "Terminators" earlier in this chapter for details.

This call transfers only complete terminator lists. The minimum request count is (number of entries * 2) + 4. The transfer count is set to this value. The maximum transfer count is 514: 4 bytes of header and 255 terminator words.

SaveTextPort (DStatus subcall)

`statusCode` = $8003.

status list = text port size and contents

This subcall copies not the text port record but the actual text port screen data into the status list buffer. The format of the data as written is `windWidth`, `windLength`, screen bytes (the contents of screen memory within the limits of the port). The size of the status list in bytes is therefore (`windWidth` × `windLength`) + 2.

This call transfers only a complete screen data record. The minimum request count is the status list size as calculated.

GetScreenChar (DStatus subcall)

`statusCode` = $8004.

status list = 1 byte

This subcall copies the current screen byte (that is, the byte underneath the cursor) to the status list. Note that this is the actual value of the byte in screen memory, which has a complex relation to the character's ASCII value. See the section "Character Set Mapping" earlier in this chapter.

The minimum request count is 1. The maximum transfer count is 1.

GetReadMode (DStatus subcall)

`statusCode` = $8005.

status list = 2 bytes

This subcall copies the current read mode flag into the status list. If zero, input is in user input mode. If $8000, input is in raw mode. The value of the read mode flag is set by the DControl subcall SetReadMode, described later in this chapter.

The minimum request count is 2. The maximum transfer count is 2.

GetDefaultString (DStatus subcall)

`statusCode` = $8006.

status list = character string

This subcall copies the current default input string into the status list. This string (set with the DControl subcall SetDefaultString) is placed in the input field at the beginning of each cycle of user input. The string can have only standard ASCII ($00–$7F) characters, and can be no more than 254 characters long.

The request count in this case defines the maximum number of bytes that can be returned.

DControl ($202E)

This call is used to send control information to the console driver. For DControl, the console driver supports most of the standard subcalls and several device-specific subcalls. Control subcalls are specified by the value of the control code parameter. The following control codes are supported:

Control code	Meaning
$0000	ResetDevice
$0001	FormatDevice
$0002	EjectMedia
$0003	SetControlParameters
$0004	SetWaitStatus
$8000	SetInputPort
$8001	SetTerminators
$8002	RestoreTextPort
$8003	SetReadMode
$8004	SetDefaultString
$8005	AbortInput

Calls with control codes less than $8000 are standard Control subcalls; calls with control codes of $8000 and over are device-specific subcalls. The calls are described more fully in the following sections.

Standard DControl subcalls

Standard DControl subcalls that are not described here function exactly as documented in Chapter 1, "GS/OS Device Call Reference."

FormatDevice:

This subcall is not applicable to character devices. It returns with no error. The transfer count is 0.

EjectMedia:

This subcall is not applicable to character devices. It returns with no error. The transfer count is 0.

SetConfigParameters:

The console driver obtains its setup information from parameter RAM and has no configuration parameters. The transfer count is 0.

SetInputPort (DControl subcall)

`controlCode` = $8000

control list = input port record

This subcall transfers data from the control list to the input port record. The data must be in the format of an input port record; see the section "The Console Input Routine" earlier in this chapter.

The minimum request count is 12. The maximum transfer count is 12.

SetTerminators (DControl subcall)

`controlCode` = $8001

control list = terminator list record

This subcall copies data from the control list to the terminator list. The format of the list is described in the section "Terminators" earlier in this chapter. The length of a terminator list in bytes is (2 * *count*) + 4, where *count* is the number of entries in the list. The minimum list length is 4; the maximum list length is 514 (2 header words plus 255 terminator characters).

The minimum request count for this subcall is 4. Furthermore, request count must match the calculated length based on the entry count parameter in the list. If there is a match, the transfer count is set to the length of the list. If the length is incorrectly stated, the previous terminators remain in effect and error $22 (invalid parameter) is returned. The driver requests memory from GS/OS Info Manager to store the terminators; if the request fails the previous and new lists of terminators are lost and error $26 (resource not available) is returned.

RestoreTextPort (DControl subcall)

`controlCode` = $8002

control list = a text port record

This subcall copies data (previously obtained through the DStatus subcall GetTextPort) from the control list back into screen memory (and thereby onto the screen). The format of the data is `windWidth`, `windLength`, screen bytes (the data to be written to screen memory within the limits of the port). If the size of the buffer is greater than that of the current text port, only the upper-left part of the data (as much as will fit) is transferred to the screen. If the buffer is smaller than the current text port, only that much of the text port (starting from the upper-left corner) will be changed; the rest of it will remain as it was before the subcall was made.

Only a complete screen record can be transferred. The minimum request count is 4. Furthermore, the request count must match the calculated length based on the width and length parameters in the control list. The total data length is therefore (`windWidth` × `windLength`) + 4. If the list is complete, the transfer count is set to that value.

SetReadMode (DControl subcall)

`controlCode` = $8003

control list = 2 bytes

This subcall sets the flag that specifies the console driver's read mode. Only the high-order bit is significant and all other bits must be set to zero. A value of $0000 selects user input mode; $8000 selects raw mode.

The minimum request count is 2. The maximum transfer count is 2.

SetDefaultString (DControl subcall)

`controlCode` = $8004

control list = character string

This subcall sets the default string for user input. This string is placed in the input field at the beginning of each cycle of user input. The string can have only standard ASCII ($00–$7F) characters, and can be no more than 254 characters long. Control characters will be displayed in inverse video. To disable the current default input string, pass a length of 0 as the request count. The driver requests memory from the GS/OS Info Manager to store the default string; if the request fails, error $26 (resource not available) is returned.

The minimum request count is 0. The maximum transfer count is 254.

AbortInput (DControl subcall)

`controlCode` = $8005

control list = none

This subcall cancels a currently in-progress input session. If `entryType` is zero, there is no input in progress and this call is ignored. Otherwise, `entryType` is reset to zero, and if a cursor is on the screen, it is removed.

The minimum request count is 0. The transfer count is 0.

DRead ($202F)

This call reads characters from the keyboard. Depending on read mode, either the call begins waiting for raw entry values, or it activates the user input mode.

In raw mode, the keyboard is scanned until (a) the transfer count equals the request count, or (b) a terminator has been pressed. The terminator character is returned as the last character of the string.

In user input mode, request count becomes the length of an edit field on the screen. This edit field begins at the current cursor location. An optional default string is displayed in the edit field. The user can edit this field using the standard editing controls, and finish editing by typing a terminator key. The terminator is treated as an editing key—it is not included in the returned string.

In either mode, an additional return condition would be if no-wait mode is selected. On exit, Transfer Count reports the length of the final string.

DWrite ($2030)

This call transfers the contents of the buffer, one byte at a time, through the console driver and to the screen. The entire buffer is transferred, and since all byte values ($00 to $FF) are defined, there are no possible errors (as long as the driver is open).

Chapter 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 generated drivers—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 your application 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. ■

◆ If you are writing a firmware driver for an Apple IIGS peripheral card, read Appendix C, "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 zero, 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, as follows:

- Peripheral card firmware is written in 6502 assembly-language code, and is executable only in emulation mode on the Apple IIGS. 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 three 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-port cards contain firmware drivers that conform to the Pascal 1.1 interface protocol. The Apple Parallel Printer 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, for example, the *Apple IIc Technical Reference Manual*) Each generated character device driver has a single device information block (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.

  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.* 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.

  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.

- Extended **SmartPort drivers:** An extended SmartPort driver has all of 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

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 from the standard ways documented in Chapter 1. Any calls or subcalls not discussed here are handled exactly as documented in Chapter 1.

DStatus

Generated drivers support these DStatus subcalls:

GetDeviceStatus
GetConfigParameters
GetWaitMode
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 added set of calls (*Optional calls*). The format of the options list is identical to the SmartPort specification and is returned unmodified in the status list.

DControl

Generated drivers support these standard DControl subcalls:

ResetDevice
FormatDevice
EjectMedia
SetControlParameters
SetWaitStatus
SetFormatOptions
AssignPartitionOwner
ArmSignal
DisarmSignal

Only the following subcalls 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 added set of calls (*Optional calls*). The format of the options list is identical to the SmartPort specification and is passed directly to the device in the control list.

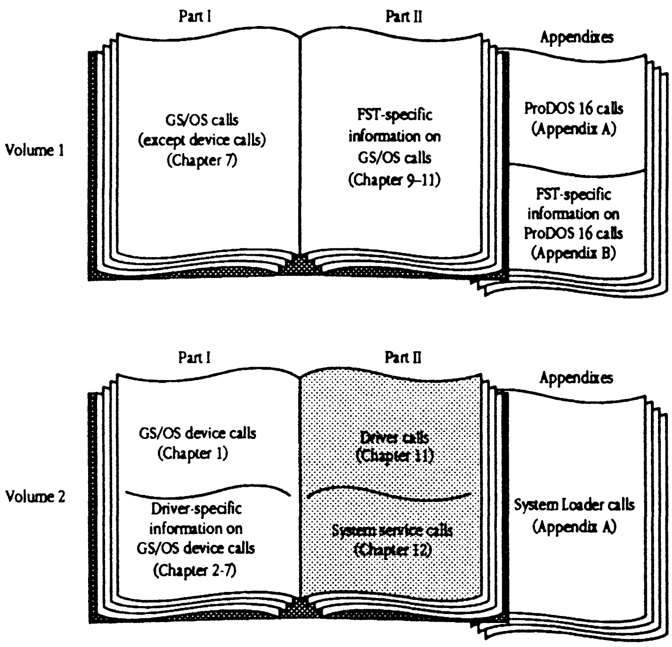
ArmSignal:

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 and supervisory 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 11. System service calls, which drivers can make to get information from GS/OS and perform certain functions, are described in Chapter 12. ■

Driver types and hierarchy

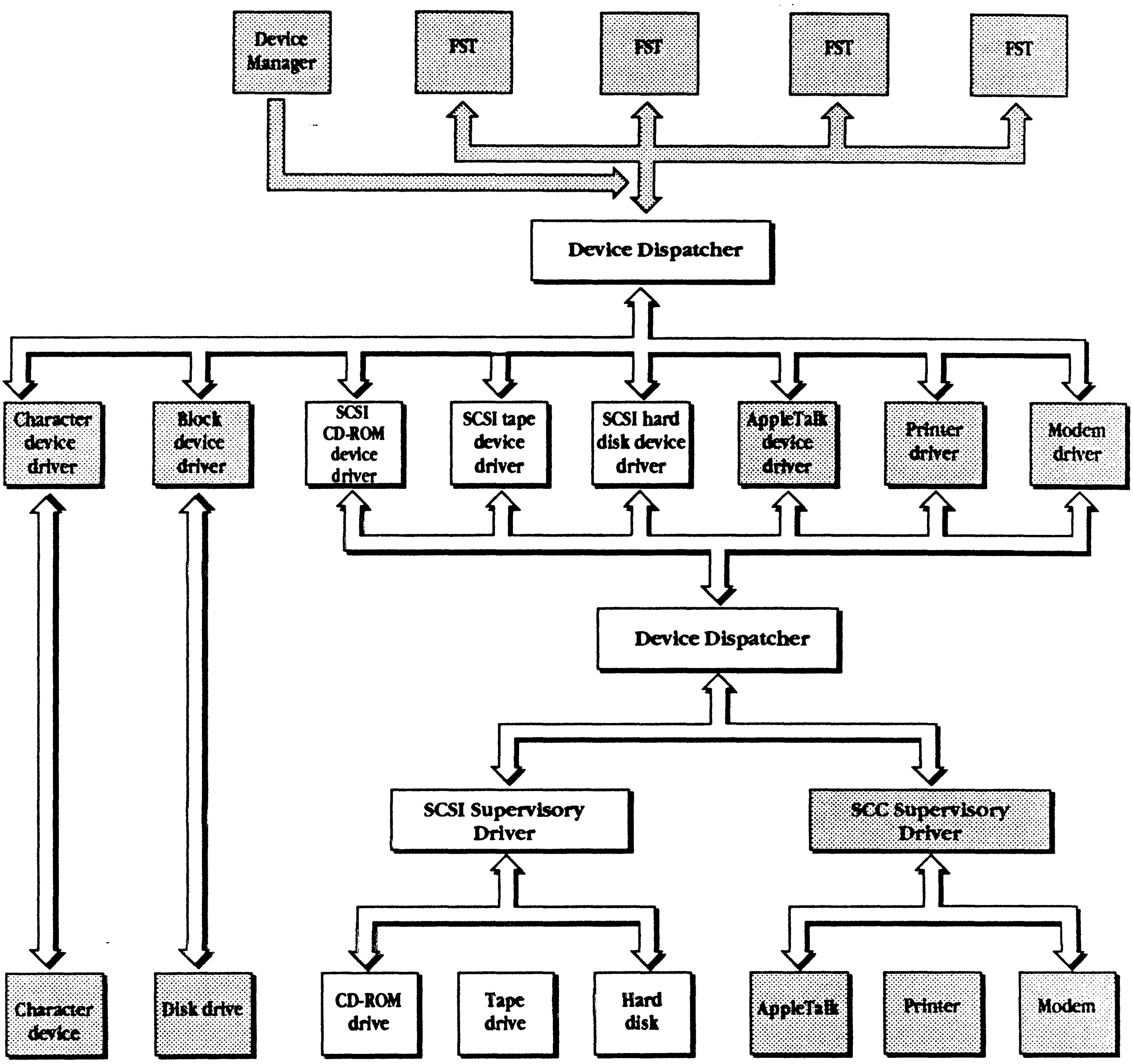
To summarize the discussion in the Introduction to this volume, drivers can be classified in three ways:

- 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. **Supervisory drivers** accept driver calls only through other device drivers and in turn access hardware devices.

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 11. Extensions to the standard calls are available for device-specific operations. Part I of this Volume 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.

■ **Figure 8-1** A hypothetical driver configuration



The diagram 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

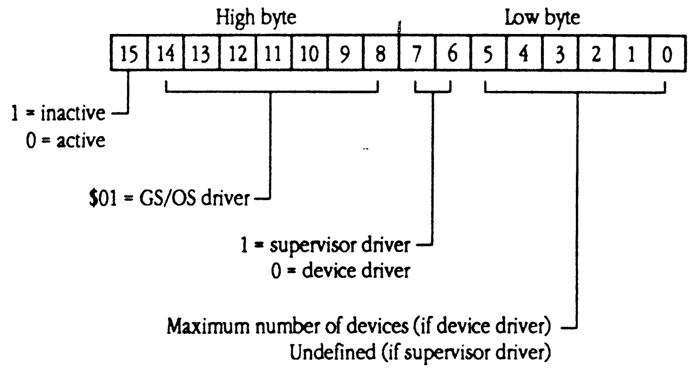
Loaded drivers are executable programs (load files). On disk, they should be in compacted format conforming to version 2.0 of object module format (OMF; see Appendix B). All Apple IIGS driver load files must have a file type of $BB.

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 setting of this flag is part of driver configuration; see Chapter 9.

The two high bits of the low-order byte of `auxType` indicate the type of GS/OS driver. Two types have been defined: device drivers and supervisory drivers. The two remaining possible values are reserved.

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** The auxiliary type field for GS/OS drivers



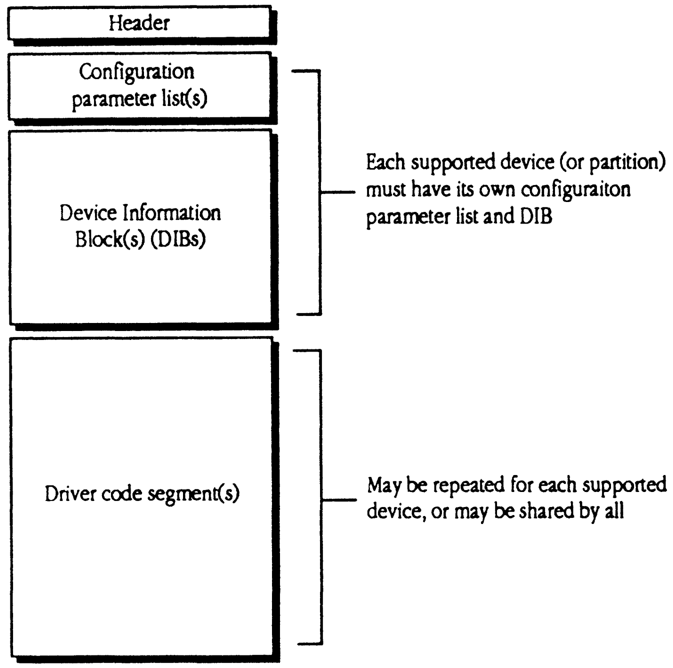
Device driver structure

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 configuration pointer and one default pointer for each device information block (DIB); for example, four DIBs would result in eight pointers
- One or more device information blocks
- 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 configuration pointer, and a default pointer must be provided for each device. The configuration pointer points to a list of configuration parameters, and the default pointer points to a list of default configuration parameters. 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; pointers 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 Appendix A of this Volume), the device dispatcher loads only the driver's static load segment, which contains its code, DIBs, and configuration lists. Configuration scripts, if present, are used only by a configuration program and are not loaded by the device dispatcher.

A device-driver header has this format:

Offset		Size	Description
\$00	`firstDIB`	Word	Offset to first DIB
\$02	`deviceCount`	Word	Count of number of devices
\$04	`list1Offset`	Word	Offset to first configuration list for device 1
\$06	`list2Offset`	Word	Offset to first configuration list for device 2
\$08	⋮	etc.	

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 first 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 zero.

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:

Offset		Size	Description
\$00	length	Word	Length of current configuration list for device 1
\$02	configList1	—	Current configuration list for device 1
	⋮		
	length	Word	Length of default configuration list for device 1
	configList2	—	Default configuration list for device 1
	⋮		

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 in bytes. A length word of zero 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 11.

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 (DIB)

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** The device information block (DIB)

Offset		Size	Description
$00	linkPtr	Longword	Pointer to the next DIB
$04	entryPtr	Longword	Pointer to the driver entry point
$08	characteristics	Word	Characteristics of this device
$0A	blockCount	Longword	Number of blocks on the device
$0E	length		
$0F	devName	String	Name of the device (ASCII, high-bit clear) (Pascal string)
⋮			

■ **Figure 8-4** The device information block (DIB) (continued)

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

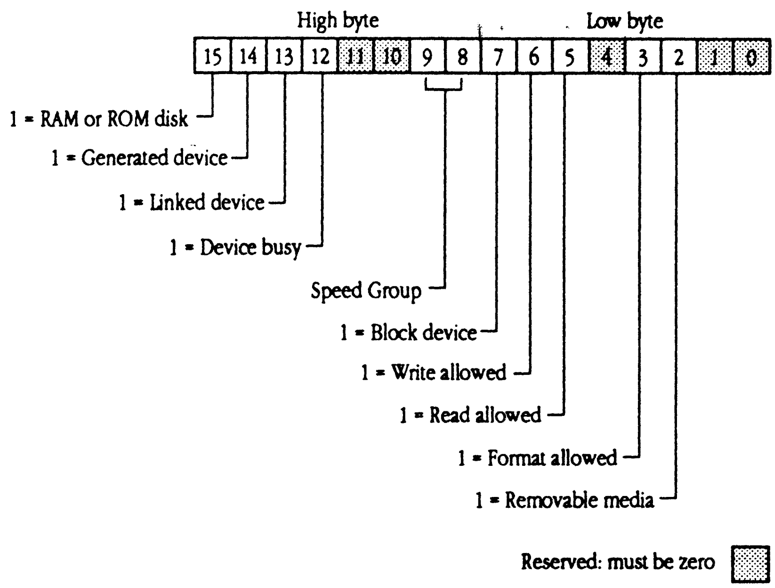
Here is what each parameter in the DIB means:

`linkPtr` Link pointer: A 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: A longword pointer to the device driver's entry point.

`characteristics` Device characteristics: A 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 zero.

■ **Figure 8-5** The 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 is 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

See the system service call SET_SYS_SPEED, in Chapter 12.

`blockCount` Block count: A 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 zero. 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.

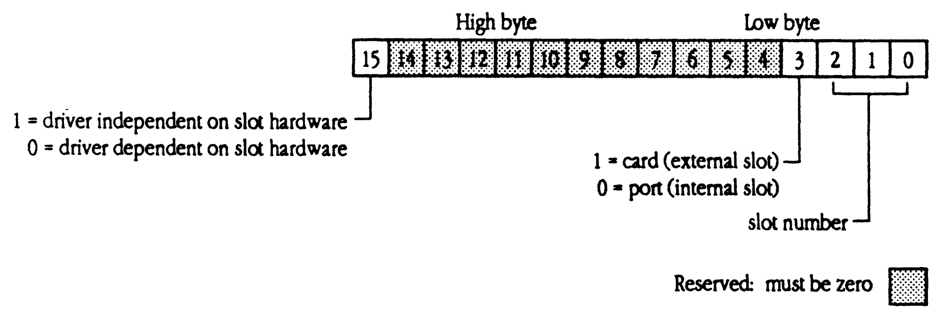
`devName` Device name: A 32-byte that which contains the device's name as a Pascal string. It consists of a length byte followed by up to 31 bytes of ASCII characters—uppercase only, high bit clear (= 0). Note that the initial period (.), which defines a device name to the system, is not part of the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

`slotNum` Slot number: A 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) 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 IIe Technical Reference Manual*. Figure 8-6 shows its format.

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 Manual* for more information on soft switches.

■ **Figure 8-6** Slot-number word



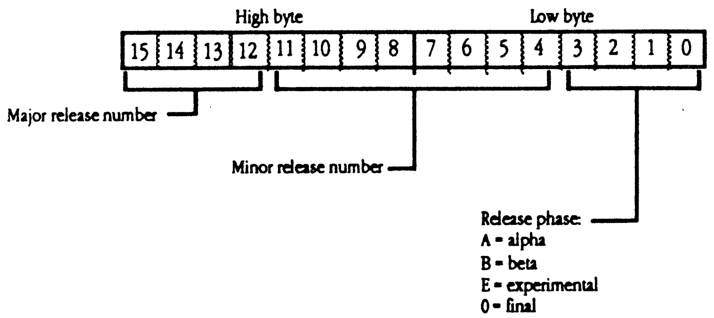
△ **Important** The driver must set bits 14-4 to zero in the slot-number word. △

`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, 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 this unit number.

`version` Driver version: A 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 its fields.

■ **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: A 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.

◆ *ID assignment:* 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)	$0010	File server
		$0011	(reserved)
		$0012	AppleDesktop Bus
$0001	ProFile (5 megabyte)	$0013	Hard-disk drive (generic)
$0002	ProFile (10 megabyte)	$0014	Floppy-disk drive (generic)
$0003	Apple 3.5 drive (includes UniDisk 3.5 drive)	$0015	Tape drive (generic)
		$0016	Character device (generic)
$0004	SCSI device (generic)	$0017	MFM-encoded disk drive
$0005	SCSI hard disk drive	$0018	AppleTalk network (generic)
$0006	SCSI tape drive	$0019	Sequential access device
$0007	SCSI CD-ROM drive	$001A	SCSI scanner
$0008	SCSI printer	$001B	Other scanner
$0009	Modem	$001C	LaserWriter SC
$000A	Console	$001D	AppleTalk main driver
$000B	Printer	$001E	AppleTalk file service driver
$000C	Serial LaserWriter	$001F	AppleTalk RPM driver
$000D	AppleTalk LaserWriter		
$000E	RAM disk		
$000F	ROM disk		

`headLink` Head device link: A 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 zero indicates that there are no devices linked to this device.

`forwardLink` Forward device link: A 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 zero 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.

◆ *DIB extensions:* 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 Get_Format_Options 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 Set_Format_Options followed by Format_Device. Device drivers that do not support media variables return a transfer count of zero 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

Offset		**Size**	**Description**
\$00	`numOptions`	Word	Number of format-options entries in the list
\$02	`numDisplayed`	Word	Number of options to be displayed
\$04	`recommendedOption`	Word	Recommended default formatting option
\$06	`currentOption`	Word	The option with which the currently on-line media was formatted
\$08	`formatOption1`	(16 bytes)	The first format-option entry
\$0C			
⋮	⋮		
	`formatOptionN`	(16 bytes)	The last format-option entry

The value specified in the `currentOption` parameter is the format option of the current on-line media. If a driver can report it, it should. 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 7 of Volume I). 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.)

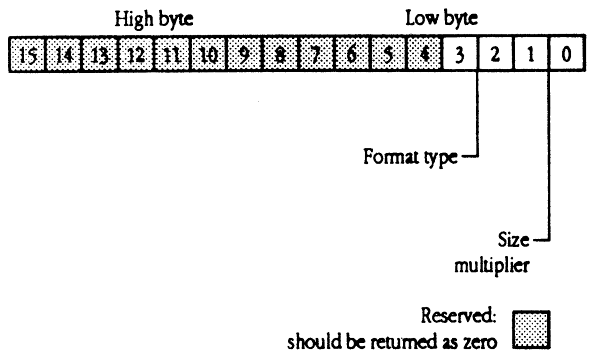
■ **Figure 8-9** Format-options entry

Offset		Size	Description
$00	formatOptionNum	Word	The number of this option
$02	linkRefNum	Word	Number of linked option
$04	flags	Word	(see definition below)
$06	blockCount	Longword	Number of blocks supported by the 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 which may appear different at the FST level. Linked options are in sets; one of the set is displayed, whereas all others are not, so that the user is not presented with several choices on the initialization dialog.

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	(for any file system)
01	Apple format	(for an Apple file system)
10	Non-Apple format	(for other file systems)
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 an example, see the description of the GS/OS call DStatus in Chapter 1, or the driver call Driver_Status in Chapter 11. See also the sample driver code in Appendix D.

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.

For a sample driver, see Appendix D.

Some points to consider when designing a device driver are as follows:

- 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 returan error if appropriate. See Chapter 11 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 re-entrant and therefore cannot accept a call while another is in progress, device drivers normally cannot make GS/OS calls. Exceptions are the calls BindInt and UnbindInt, usually made during driver startup and shutdown, respectively.

  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. Alternatively, consider making some routines into signal handlers instead. See Chapter 10.
- A small workspace is available on the GS/OS direct page for device-driver and supervisory-driver use; that workspace is described later in this chapter in the section "How device drivers (and GS/OS) call supervisory drivers.". The workspace is not permanent; it may be corrupted between driver calls. Except for that workspace, supervisory drivers should not permanently modify any other GS/OS direct-page location that is not within the bounds of the small workspace. A supervisory driver requiring direct-page space should save and restore the contents of any other direct-page location that it uses.

△ **Important** If the driver makes system service calls, those calls can corrupt any direct page location not in the small workspace. △

Alternatively, a supervisory driver requiring large amounts of 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. △

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 11.

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 them 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 sucessfully 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 12. Because of this call, the GS/OS device list can grow during program execution. (On the other hand, the device list cannot shrink; there is no mechanism for removing devices from the device list.)

To dynamically install and startup a driver, take the following steps:

1. Make the INSTALL-DRIVER call.
2. Check for out-of-memory or busy errors. If either of those errors occurred, no drivers were installed. Postpone installation until later.

   If neither of those errors occurred, the drivers will be installed in the system as soon as the system is not busy (that is, as soon as the current driver finishes executing).

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.

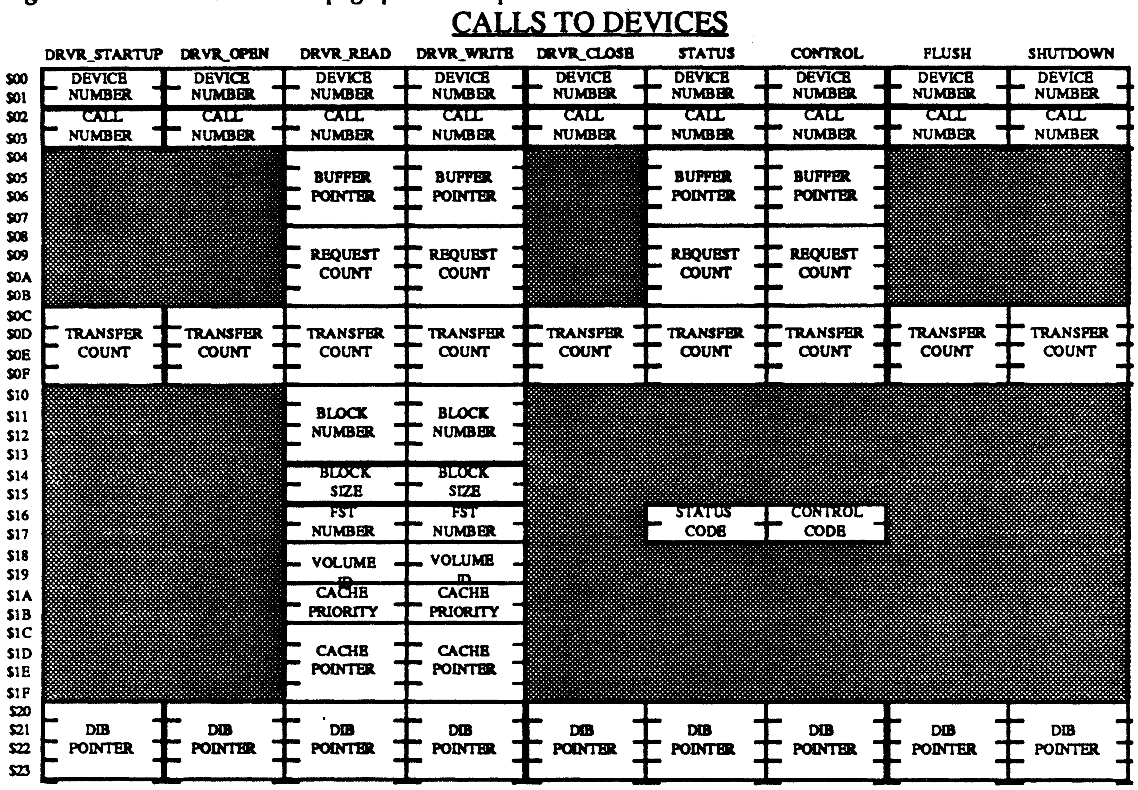
If applications scan devices in this manner, dynamically installed devices will always be included in the scan operation.

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 to be 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 zero, 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[1]
X	Unspecified
Y	Unspecified
D	**Base of GS/OS direct page**
S	**Top of GS/OS stack**
DBR	Current value
P register flags	
e	**0 (native mode)**
m	**0 (16-bit)**
x	**0 (16-bit)**
i	0 (enabled)
c	Unspecified[2]
decimal	0
Speed	Fast

[1]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).
[2]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 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. △

A small workspace is available for device-driver use on the GS/OS direct page. Locations \$5A through \$5F are available for device drivers; locations \$66 through \$6B are shared by device drivers and supervisory drivers (and may be corrupted by either a driver call or supervisory driver call). This workspace is not permanent; it may be corrupted between driver calls.

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 footnotes 2 and 3 to Table 8-2.

◆ *Disk-switched status:* When a driver call returns to the device dispatcher, the device dispatcher post-processes any error codes from the device. If either a disk-switched or 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 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 off-line condition. See descriptions of the driver call Driver_Status (Chapter 11) and the system service call SET_DISKSW (Chapter 12).

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. Here is a list and brief description of them:

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 non-open 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 11, "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, or 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 the available routines:

Name	**Function**
CACHE_FIND_BLK	Searches for a disk block in the cache
CACHE_ADD_BLK	Adds a block of memory to the cache
SET_SYS_SPEED	Controls processor execution speed
MOVE_INFO	Moves data between memory buffers
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.

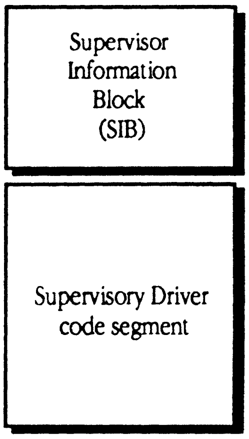
For more information, see Chapter 12.

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** The supervisor information block (SIB)

Offset		Size	Description
$00	`entryPtr`	Longword	Pointer to the supervisory driver entry point
$04	`supervisorID`	Word	The general type of supervisory driver
$06	`version`	Word	The version of the supervisory driver
$08	`extDIBPtr`	Longword	Pointer to name string; string length byte followed by ASCII string
$0C	`(reserved)`	Word	(reserved)
$0E	`(reserved)`	Word	(reserved)

The defined parameters in the SIB have these meanings:

`entryPointer` 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)

- *ID assignment:* 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 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 11). 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.

Appendix D shows the overall structure a supervisory driver might have. 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[1]
X	Unspecified
Y	Unspecified
D	**Base of GS/OS direct page**
S	**Top of GS/OS stack**
DBR	Current value
P register flags	
e	**0 (native mode)**
m	**0 (16-bit)**
x	**0 (16-bit)**
i	0 (enabled)
c	Unspecified[2]
Speed	Fast

[1]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, 0 if no error). See individual call descriptions.
[2]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 11-3, under "About Supervisory-Driver Calls" in Chapter 11.

A small workspace is available on the GS/OS direct page for device-driver and supervisory-driver use. By convention, locations $5A through $5F are available for device drivers; locations $60 through $65 are available for supervisory drivers. Locations $66 through $6B are shared by device drivers and supervisory drivers (and may be corrupted by either a driver call or supervisory-driver call). This workspace is not permanent; it may be corrupted between driver calls. Naturally, a supervisory driver and its device drivers may set up their own scratchpad workspace allocation.

Supervisory drivers should not permanently modify any GS/OS direct-page location that is not within the bounds of the small workspace. A supervisory driver requiring direct-page space should save and restore the contents of any other direct-page location that it uses.

△ **Important** If the driver makes system service calls, those calls can corrupt any direct page location not in the small workspace. △

Alternatively, a supervisory driver requiring large amounts of 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 footnotes 1 and 2 to Table 8-4.

Here are a list and 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 11 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 Disk Cache desk accessory or through the Control Panel program) can control whether caching is enabled and what the maximum cache size can be. It is the driver, however, that 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 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 32KB 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 32KB 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.

◆ *Macintosh:* These features differ from caching on the Macintosh, 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 12.

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	Tries 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 and writing to its device. See also the device-driver sample code in Appendix D.

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 11. If the cache priority is nonzero, the driver should support caching by doing this:

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_BLOCK.

3. If the block is not in the cache:
   a. Call CACHE_ADD_BLOCK 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. Go to step 4.
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 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 11. If the cache priority is nonzero, the driver should support caching by doing this:

1. Search for the block in the cache by calling CACHE_FIND_BLOCK.
2. If the block is not in the cache:
   a. Call CACHE_ADD BLOCK 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 read from the device, and then transfer those blocks to the cache.

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 zero 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 **Handling Interrupts and Signals**

Interrupt handlers are programs that execute in response to a hardware interrupt. Interrupts and interrupt handlers are commonly used by device drivers to operate their devices more efficiently and to make possible simple background tasks such as printer spooling.

Under GS/OS, a **signal** is a software message from one subsystem to a second that something of interest to the second has happened. The most common kind of signal is a software response to a hardware interrupt, but signals need not be triggered by interrupts. **Signal handlers** are programs that execute in response to the occurrence of a signal. They are similar to interrupt handlers except that signal handlers can make operating system calls. The GS/OS Call Manager is responsible for managing and dispatching to both interrupt handlers and signal handlers.

An interrupt handler is commonly written in conjunction with a driver and is installed when the driver starts up. A signal handler is commonly written in conjunction with either a driver or an application, and it is installed by the driver or application during execution. This chapter discusses requirements for designing and installing both types of handlers. ■

Interrupts

An **interrupt** is a hardware signal that is sent from an external or internal device to the CPU. On the Apple IIGS, when the CPU receives an interrupt the following actions occur:

1. The CPU suspends execution of the current program, saves the program's state, and transfers control to the Apple IIGS firmware interrupt dispatcher. The firmware dispatcher sets up a specific firmware interrupt environment.
2. If it is an interrupt that has a GS/OS interrupt handler, the firmware dispatcher passes control to GS/OS. GS/OS sets up a specific GS/OS interrupt environment and in turn transfers control to the proper handler.
3. The interrupt handler performs the functions required by the occurrence of the interrupt. After it has done its job, the interrupt handler returns control to GS/OS.
4. GS/OS restores the firmware interrupt environment and returns control to the firmware dispatcher. The firmware dispatcher restores the state of the interrupted application and returns execution to it as if nothing had happened.

In a non-multitasking system such as GS/OS, interrupts are commonly used by device drivers to operate their devices more efficiently and to make possible simple background tasks such as printer spooling.

This section discusses what the sources of interrupts are, how interrupt handlers are dispatched to, how interrupt handlers function within their execution environment, and how interrupt sources are connected to interrupt handlers. It also discusses interrupt-handler lifetime and how GS/OS treats unclaimed interrupts.

Interrupt sources

Each distinct hardware device that can generate an interrupt is known as an **interrupt source.** For example, each Apple IIGS expansion slot with a hardware card is an interrupt source, and internal devices as the mouse and serial ports are also sources. Every interrupt source that is explicitly identifiable by the firmware has a unique identifier known as its **vector reference number (VRN).** VRNs are used to associate interrupt sources with interrupt handlers.

VRNs are permanently associated with specific interrupt sources; they will not change with future revisions to GS/OS or the Apple IIGS computer. If your interrupt handler now appropriately handles an interrupt source with VRN=*n*, it will be able to handle VRN=*n* on any future versions of GS/OS on any Apple IIGS.

Table 10-1 lists the currently defined VRNs and their associated interrupt sources.

■ **Table 10-1** VRNs and interrupt sources

VRN	Interrupt source	VRN	Interrupt source
$0008	AppleTalk port		
$0009	Serial input port		
$000A	Scan line		
$000B	Sound-chip waveform completion		
$000C	VBL		
$000D	Mouse button or movement		
$000E	Quarter–second timer		
$000F	Keyboard		
$0010	ADB response (keyboard)		
$0011	SRQ **(keyboard)**		
$0012	Desk Manager		
$0013	Flush command (keyboard)		
$0014	Microcontroller abort **(keyboard)**		
$0015	Clock chip 1-second timer		
$0016	External interrupt source (slot)		
$0017	Other interrupt source		

As new interrupt sources (such as internal and external slots, timers, counters, etc.) are defined in future versions of the Apple IIGS, each will be assigned a unique VRN by Apple Computer, Inc.

Interrupt dispatching

Interrupt dispatching is the process of handing control to the appropriate interrupt handler after an interrupt occurs. In the Apple IIGS, most interrupt dispatching and interrupt handling is performed by firmware. Although the Apple IIGS hardware generates a number of distinct interrupt notifications—ABORT, COP, BRK, NMI, and IRQ—the only interrupt of interest to GS/OS interrupt-handler writers is IRQ (Interrupt Request). The firmware dispatches each IRQ by polling the interrupt handlers through the firmware interrupt vectors (one for each VRN defined in Table 10-1) until one of them signals that it has handled the interrupt.

Because of critical timing constraints, the firmware interrupt dispatcher polls the AppleTalk and serial port vectors first, before polling the less time-critical vectors such as vertical blanking, quarter-second timer, and keyboard. If none of the firmware handlers associated with defined sources accepts the interrupt, the firmware dispatcher polls through vector $0017 (other interrupt source). If the interrupt still remains unhandled, the firmware dispatcher passes control through the **user interrupt vector** at $00 03FE. Finally, if no handlers associated with the user interrupt vector accept the interrupt, it becomes an **unclaimed interrupt,** described later in this section.

There are two ways in which GS/OS can get control from the firmware dispatcher during this process, in order to pass control on to a GS/OS interrupt handler:

1. Through one of the firmware interrupt vectors. When GS/OS gets control this way, it polls only the interrupt handlers that are associated with the particular vector reference number (VRN) of that interrupt vector. These handlers are installed with the GS/OS call BindInt, described later in this section.
2. Through the user interrupt vector ($00 03FE). When GS/OS gets control this way, it polls all the installed ProDOS 16 interrupt handlers. ProDOS 16 interrupt handlers are installed with the GS/OS ProDOS 16-compatible call ALLOC_INTERRUPT, described in Appendix A of Volume 1.

Within a polling sequence, the polling order is undefined.

Interrupt handler structure and execution environment

A GS/OS **interrupt handler** consists of code in either a device driver, application, or desk accessory. The interrupt handler must have a single defined entry point. When an interrupt occurs, GS/OS sets up a specific **execution environment** and then calls the interrupt handler with a JSL instruction to that entry point.

The code beginning at the specified entry point should first determine whether or not the interrupt is the one to be handled by this interrupt handler. If it is not, the interrupt handler should restore the execution environment as set up by GS/OS, set the carry flag (c=1), and return with an RTL. If the interrupt is the proper one, the interrupt handler should perform whatever tasks necessary to handle the interrupt, restore the proper execution environment, clear the carry flag (c=0), and return with an RTL.

What execution environment GS/OS sets up for an interrupt handler depends on its type. As far as execution environments are concerned, there are three basic types:

- A GS/OS interrupt handler bound to the AppleTalk or serial port firmware vector
- A GS/OS interrupt handler bound to any other firmware vector.
- A ProDOS 16 interrupt handler installed through the user interrupt vector

Table 10-2 shows the execution environment of each of these handlers when it starts executing. The table also notes which parts of the environment need to be preserved (or restored on exit). Boldface entries in the table indicate the components of the environment that the handler must restore before returning.

■ **Table 10-2** Interrupt-handler execution environments

Component	**GS/OS AppleTalk or serial handler**	**Other GS/OS handler**	**ProDOS 16 handler**	
Registers				
A, X, Y	Undefined	Undefined	Undefined	
D	**Undefined**	**$0000**	Undefined	
S	Undefined[1]	Undefined[1]	Undefined[1]	
DB	**Undefined**	**$00**	Undefined	
PB	Handler entry point		Handler entry point	
	Handler entry point			
PC	Handler entry point		Handler entry point	
	Handler entry point			
P register flags				
e	**0 (native mode)**		**0 (native mode)**	**0**
(native mode)				
m	**1 (8-bit)**	**1 (8-bit)**	**0 (16-bit)**	
x	**1 (8-bit)**	**1 (8-bit)**	**0 (16-bit)**	
i	1 (**disabled**)[2]	1 (**disabled**)[2]	1 (**disabled**)[2]	
c	Undefined[3, 4]	Undefined[3, 4]	1[4]	
Speed	**Fast**	**Fast**	**Fast**	

[1]On entry, the three-byte return address to GS/OS is on top of the stack. When the interrupt handler executes its RTL, this three-byte address is popped from the stack.
[2]An interrupt handler must never enable interrupts.
[3]If c=0 on entry, the interrupt has not yet been handled; if c=1 on entry, the interrupt has already been handled.
[4]If the interrupt handler handles the interrupt, it sets c=0 before returning. If not, it sets c=1 before returning.

Note from Table 10-2 that the carry flag is always set (c=1) on entry to a ProDOS 16 interrupt handler, whereas it can be either 0 or 1 on entry to a GS/OS interrupt handler. ProDOS 16 handlers are polled only as long as the interrupt is still unclaimed; as soon as one handler takes it and clears the carry flag, polling stops. On the other hand, all GS/OS handlers bound to a particular VRN are polled during an interrupt, even if another handler with that VRN has already cleared the interrupt. That way, all handlers associated with a VRN can do updating or other desired tasks at each interrupt.

The first GS/OS handler to respond to an interrupt should perform its normal functions, including reenabling the interrupt source, clearing the carry flag, and returning. Subsequent handlers, on seeing that c=1 on entry, may perform other tasks as desired but should not themselves reenable the interrupt source, change the value of the carry flag, or permanently modify the environment.

Here are some other points to remember in designing an interrupt handler:

- If the interrupt handler needs to use direct-page space, it must save and restore the contents of any locations that it uses.
- An interrupt handler must never enable interrupts.
- Because interrupts cannot be disabled for longer than 0.25 seconds in the Apple IIGS (an AppleTalk requirement), interrupt handlers must execute in less than a quarter-second.
- Because GS/OS is not reentrant, an interrupt handler should not make GS/OS calls. If your interrupt handler needs to make operating system calls, you should make it a signal handler instead. See "Signals," later in this chapter.

Connecting interrupt sources to interrupt handlers

You install and remove GS/OS interrupt handlers by making the standard GS/OS calls BindInt and UnbindInt, respectively.

To avoid unclaimed interrupts, make sure that the code that installs an interrupt handler does not enable the interrupt source until the interrupt handler is installed. Likewise, the code that removes an interrupt handler must disable the interrupt source before removing the handler.

BindInt call

This call establishes a binding, or correspondence, between a specified interrupt source and a specified GS/OS interrupt handler. GS/OS adds the interrupt handler to the set of handlers to be polled when the specified (by VRN) interrupt occurs. The polling order is undefined within the handlers bound to that interrupt vector.

The interrupt identification number returned by the call uniquely identifies the binding between interrupt source and interrupt handler. Its only use is in the GS/OS UnbindInt call. Note that several interrupt handlers may be bound to the same interrupt source.

For a description of the BindInt call, see Chapter 7 of Volume 1.

UnbindInt call

This call severs the binding previously established between an interrupt source and interrupt handler by a BindInt call. It makes the interrupt handler unavailable.

For a description of the UnbindInt call, see Chapter 7 of Volume 1.

◆ *ProDOS 16:* ProDOS 16 interrupt handlers are installed and removed with the ProDOS 16 calls ALLOC_INTERRUPT and DEALLOC_INTERRUPT. See Appendix A of Volume 1.

Interrupt handler lifetime

The lifetime of an interrupt handler is the time during which its code is resident in memory and capable of being executed. During its lifetime, the interrupt handler may be installed (able to handle its interrupts) or removed (still resident in memory but unable to handle its interrupts).

The interrupt handler is installed when the device driver or application makes a BindInt call for it, and removed when the device driver or application makes an UnbindInt call. *The program that performs the BindInt call must perform an UnbindInt call before the lifetime of the interrupt handler ends*. There is no automatic mechanism for removing GS/OS interrupt handlers when an application quits, and a dispatch to the previous entry point of an installed but now completely gone interrupt handler could cause a system crash or loss of data.

◆ *Drivers making GS/OS calls:* Note that BindInt and UnbindInt are exceptions to the rule that drivers cannot make operating system calls.

A GS/OS interrupt handler has a lifetime equivalent to the code containing it. For example, if the interrupt handler is part of a device driver, it lives as long as the device driver is in memory and capable of being executed. Thus, the lifetime of a GS/OS interrupt handler may span several GS/OS applications. In this case, the lifetime ends when the user executes a non-GS/OS application or the hardware reboots.

Unclaimed interrupts

If none of the interrupt handlers on an Apple IIGS accepts a given interrupt, it is known as an **unclaimed interrupt**. Possible causes of unclaimed interrupts include the following:

- software problems, such as a failure to bind the interrupt handler before enabling the interrupt source it handles
- interrupt-related hardware problems, such as failure by the interrupting device to maintain an "I am the source of the interrupt" flag after signalling an interrupt to the processor.
- hardware failures such as intermittent shorts of the interrupt line to ground
- random transient phenomena such as cosmic-ray or subatomic-particle bombardment

An unclaimed interrupt is a serious problem but shouldn't cause a system failure if the interrupt was due to a random transient phenomenon. Therefore, GS/OS maintains an unclaimed interrupt counter that is initialized to 0 at GS/OS startup time. Whenever an unclaimed interrupt occurs, GS/OS increments the counter. Whenever an interrupt is serviced by an interrupt handler, GS/OS sets the counter back to 0. If the counter ever reaches 65,536, GS/OS causes a system failure.

Signals

A **signal** is a message from one software subsystem to a second that something of interest to the second has occurred. When a signal occurs, GS/OS typically places it in the **signal queue** for eventual handling. As soon as it can, GS/OS suspends execution of the current program, saves the program's state, removes the signal from the queue, calls the signal handler in the receiving subsystem to process the signal, and finally restores the state and returns to the suspended program.

The most important feature of signal handlers is that they are allowed to make GS/OS calls. That is why the signal queue exists; GS/OS removes signals from the queue and executes their signal handlers only when GS/OS is free to accept a call.

The most common kind of signal is a software response to a hardware interrupt. For example, a modem driver may use a *loss of carrier* interrupt to trigger a corresponding signal, whose signal handler calls GS/OS to close a file of terminal input data. Similarly, a spooling printer driver may translate a *line completion* interrupt into a corresponding signal whose signal handler uses GS/OS calls to read the next line from a spool file and move it into the printer's output buffer.

In principle, however, signals need not be triggered by interrupts: a signal can indicate, for example, a ***message received*** condition on a network interface or a ***new volume mounted*** condition on a disk drive.

◆ Signals are not meant to provide a general mechanism for interprocess communication in a multitasking environment. Their principal capability is synchronization of handler execution with time periods when the operating system is able to accept calls.

Signals are analogous to interrupts but are handled with less urgency. If immediate response to an interrupt request is needed, and if the routine that handles the interrupt needn't make any operating system calls, then it should be an interrupt handler. On the other hand, if a certain amount of delay can be tolerated, the full range of operating system calls are available to a handler if it is a signal handler.

This section discusses what signal sources are, how GS/OS dispatches to signal handlers, how signal handlers function within their execution environment, how signal sources are connected with signal handlers, and how the occurrence of a signal is announced.

Signal sources

A **signal source** is software; it is a routine that announces the occurrence of a signal when it detects the prerequisite conditions for that signal. For example, a modem device driver may contain an interrupt handler capable of detecting the conditions needed to announce the *loss of carrier* signal. In that case the interrupt handler's primary purpose is to be a signal source. The most common class of signal sources is probably interrupt handlers within device drivers.

Signal sources announce signals to GS/OS by making the system service call SIGNAL, described in Chapter 12. When a signal source announces a signal to GS/OS, it passes along the information needed to execute the source's signal handler. (That information was sent to the signal source when the signal was **armed;** see "Arming and Disarming Signals," later in this chapter.) GS/OS accepts that information and either executes that signal's signal handler immediately or saves the information for later; GS/OS then returns control to the process that announced the signal.

◆ A signal source that announces a signal as the result of an interrupt should generate no more than one signal per interrupt, to avoid the possibility of overflowing the signal queue.

Signal dispatching and the signal queue

Signal dispatching is the process of calling signal handlers. GS/OS dispatches signals only when it is not busy processing a GS/OS call, so that signal handlers are always able to make system calls.

When a signal occurs, if GS/OS is not busy handling a GS/OS call and if the system is in a non-interrupt state, the GS/OS Call Manager executes the signal handler immediately. On the other hand, if a GS/OS call is in progress when the signal occurs, the signal cannot be dispatched; the Call Manager instead places the signal in the signal queue. Signals are placed in the queue in order of **signal priority**; queued signals with higher priority numbers are placed in front of signals with lower priorities, meaning that they will be executed first.

The signal queue can hold a maximum of 16 signals. If a signal arrives and the queue is full, the queue overflows, and the signal call returns an error.

GS/OS dispatches a queued signal by pulling it off the front of the queue (that is, by taking the oldest signal with the highest priority) and calling the signal's handler. To process signals as quickly as possible, minimize the time during which interrupts are disabled, and assure that all signals are eventually handled, GS/OS uses the signal dispatching strategy described in Table 10-3.

■ **Table 10-3** GS/OS signal-dispatching strategy

Situation	Action taken
GS/OS is exiting from a system call, system is in non-interrupt state.	Execute all queued signals.
GS/OS is exiting from a system call, system is in interrupt state.	Execute only the first queued signal.
Signal arrives while GS/OS is inactive and the system is in non-interrupt state.	Execute all queued signals, including the one being signaled.
Signal arrives while GS/OS is inactive and the system is in interrupt state.	Queue the arriving signal and execute only the first queued signal.
Signal arrives while GS/OS is active.	Do not execute any signals and queue the arriving signal.

In addition, to make absolutely sure that no signals are left unexecuted, GS/OS uses the VBL interrupt to execute all remaining signals in the queue every 0.5 seconds.

Signal handler structure and execution environment

A **signal handler** is a subroutine somewhere in memory that is called by GS/OS in response to the signal that it handles. The signal handler must have a single defined entry point. When it dispatches to the signal handler, GS/OS saves the state of the current application and sets up a specific **signal-handler environment**; GS/OS then calls the signal handler with a JSL instruction to its entry point. The features of the signal handler environment are shown in Table 10-4. Boldface entries in the table indicate the components of the environment that the handler must restore before returning.

■ **Table 10-4** Signal-handler execution environment

Component	State
Registers	
A	Undefined
X	Undefined
Y	Undefined
D	Current direct page
S	Current stack pointer
DBR	Undefined
P register flags	
e	**0 (native mode)**
m	**0 (16-bit)**
x	**0 (16-bit)**
i	1 **(disabled)**[1]
Speed	High

[1]A signal handler must never enable interrupts.

Here are some other points related to signal handler design:

- Signal handlers must return with an RTL.
- Because interrupts cannot be disabled for longer than 0.25 seconds on the Apple IIGS (an AppleTalk requirement), and because signal handlers may run in an interrupt environment (during which interrupts are diababled), signal handlers must execute in less that a quarter-second.
- Signal handlers must never enable interrupts.
- An interrupt may preempt execution of a signal handler, but a signal handler is never preempted to execute another signal handler, even one of higher priority. Any signal handler that you write can count on execution without interference from another signal handler.
- The lifetime of a signal handler is the same as the lifetime of the software that contains it. Therefore, if your signal handler is part of a device driver, it can span several applications.

Arming and disarming signals

A program needs to **arm**, or install, a signal in order to use it. Arming a signal is the process of providing its signal source with the information needed to execute its signal handler. This information includes the signal handler's code entry point and the signal's priority. Arming implies that the signal handler is ready to process occurrences of the signal.

When the program no longer needs to use the signal, it must disarm (remove) it. Disarming a signal is the process of telling the signal source that the signal handler will no longer process occurrences of the signal.

Therefore, every signal source must support the ArmSignal and DisarmSignal functions for its signal. How the source implements the functions is source-specific; however, it must at least save the information passed to it by ArmSignal and maintain a flag noting whether the signal is currently armed or disarmed. Two standards exist for ArmSignal and DisarmSignal calls: one for signal sources in device drivers and one for all other signal sources.

Arming device driver signal sources

To arm a signal that is generated by a device driver, the caller (application or device driver) performs an ArmSignal subcall of the GS/OS call DControl, passing the following information to the driver that contains the signal source:

- the **signal code**, an arbitrary value defined by the signal source to identify the signals that the source generates. The signal code is used only in the DisarmSignal call.
- the **signal priority** to be given to signals from this source; $0000 is the lowest priority and $FFFF is the highest.
- the **signal handler address**, the entry point of the handler for signals generated by this source.

The driver receives the call (from the device dispatcher) as an Arm_Signal subcall of the driver call Driver_Control. The format in which these parameters are passed, and the procedure for making the ArmSignal subcall, are documented under "DControl" in Chapter 1; the format in which the driver receives the parameters is documented under "Driver_Control" in Chapter 11.

△ **Important** Before it arms a given signal, the program making the ArmSignal call must ensure that the signal handler for that signal is ready to process the signal. △

The ArmSignal subcall can return error number $22 (invalid signal code) or error number $29 (driver busy, which is this case means that the signal is already armed).

Disarming device driver signal sources

To disarm a signal that is generated by a device driver, the caller (application or device driver) performs a DisarmSignal subcall of the GS/OS call DControl, passing the following information to the driver that contains the signal source:

- the **signal code**, the value assigned by the caller when the signal was armed (with the ArmSignal call).

The driver receives the call (from the device dispatcher) as a Disarm_Signal subcall of the driver call Driver_Control. The format in which the parameter is passed, and the procedure for making the DisarmSignal subcall, are documented under "DControl" in Chapter 1; the format in which the driver receives the parameters is documented under "Driver_Control" in Chapter 11.

△ **Important** The program making the DisarmSignal call must not disable or remove the signal handler from memory until after the call is made. △

The Disarm Signal subcall can return error $22 (invalid signal code, which in this case means that the signal was never armed)

Arming other signal sources

A signal source that is not part of a device driver must have an ArmSignal entry point that behaves essentially like the ArmSignal subcall of DControl. The application or device driver calls the entry point by using a JSL instruction, as shown in this APW assembly-language example:

```
pea     parameter_block|-16   ;push high word of param block ptr
pea     parameter_block       ;push low word of para block ptr
jsl     arm_signal_e          ;long jump to arm procedure
```

The parameter block should have the following form:

```
dc      i2'signal_code'
dc      i2'priority'
dc      i4'handler_address'
```

These parameters have the same format and meaning as those described under "Arming Device Driver Signal Sources," earlier in this section.

On an ArmSignal call, a non-device-driver signal source must return with the carry flag clear (c = 0) if no error occurred, or with the flag set (c = 1) and the error code in the accumulator if an error occurred. The call should support these errors:

Code	**Meaning**
A=$0001	Invalid signal code
A=$0002	Signal already armed

Disarming other signal sources

A signal source that is not part of a device driver must have a DisarmSignal entry point that behaves essentially like the DisarmSignal subcall of DControl. The application or device driver calls the entry point, as shown in this APW assembly-language example:

```
        pea     signal_code         ;push signal code onto stack
        jsl     disarm_signal_e     ;call disarm procedure for the specific signal
```

On an DisarmSignal call, a non–device–driver signal source must return with the carry flag clear (c = 0) if no error occurred, or with the flag set (c = 1) and the error code in the accumulator if an error occurred. The call should support this error:

Code	Meaning
A=$0001	Invalid signal code

Chapter 11 **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 other GS/OS calls described elesewhere; 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 volume.

Table 11-1 lists the driver calls every GS/OS device driver must accept.

■ Table 11-1 GS/OS driver calls

No.	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 non-open 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 Volume that GS/OS recognizes both device drivers and supervisory drivers. Supervisory drivers handle a different set of calls from those listed in Table 11-1; 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 11-1 is a diagram of that parameter block.

■ **Figure 11-1** Direct-page parameter space for driver calls

All driver calls use the same memory locations.

Offset		Description
$00	deviceNum	The number of the device to whom the call is made
$02	callNum	The number of the call being made
$04	bufferPtr	Pointer to a buffer for reading or writing data
$08	requestCount	The number of bytes to transfer to or from driver
$0C	transferCount	The number of bytes transferred by the call
$10	blockNum	The number of the block to start a read or write at
$14	blockSize	How many bytes per block for this device
$16	fstNum OR code	This device's FST number *or* status code *or* control code
$18	volumeID	The VRN for blocks on this device
$1A	cachePriority	What sort of caching to implement
$1C	cachePointer	Pointer to the current block in the cache (this parameter is used only indirectly in driver calls)
$20	dibPointer	Pointer to the DIB for this device

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 one 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.

◆ *Transfer count:* 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.

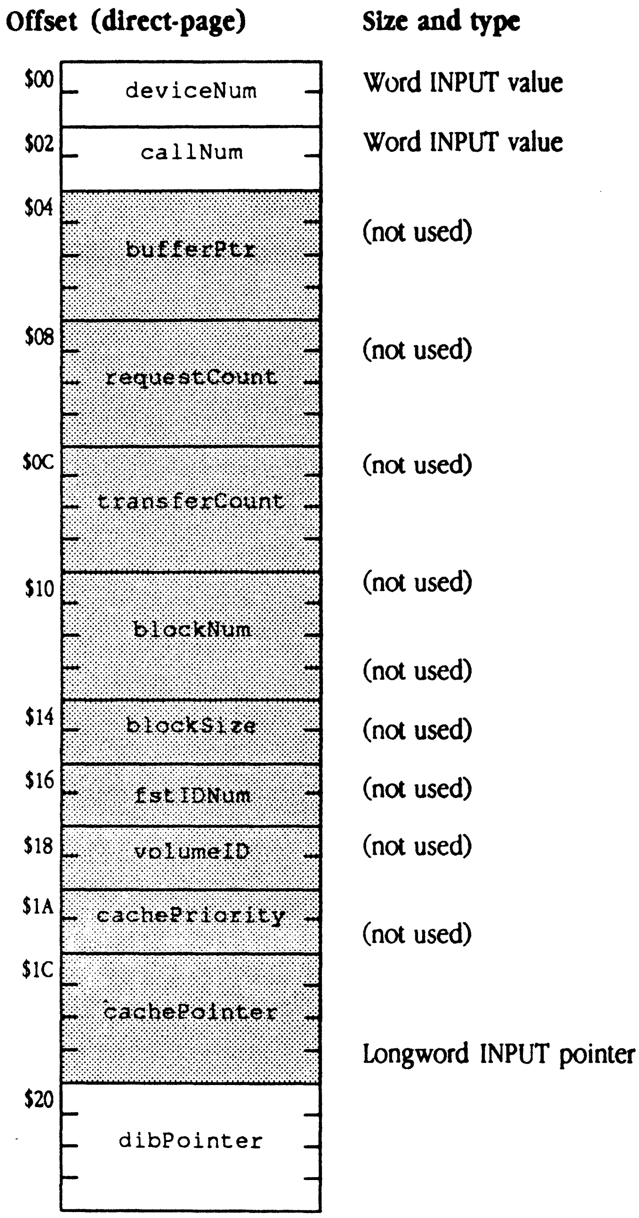
- **Unused parameters:** Although all calls use the same direct-page parameter space, not all parameters are used for every call. For each call description, parameters that are not used are shaded in the parameter-block diagram.

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.

$0000 Driver_Startup

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.

△ **Important** 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.△

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.

△ **Important** 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 device information block (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 possibe 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:
   - Set the auxiliary type of the driver file to 3F.
   - Determine the number of devices.
   - Allocate the memory for the DIBs.
   - 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.

$0001 Driver_Open

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:

Offset (direct-page)		Size and type
$00	deviceNum	Word INPUT value
$02	callNum	Word INPUT value
$04	bufferPtr	(not used)
$08	requestCount	(not used)
$0C	transferCount	(not used)
$10	blockNum	(not used)
$14	blockSize	(not used)
$16		(not used)
$18	volumeID	(not used)
$1A	cachePriority	(not used)
$1C	cachePointer	(not used)
$20	dibPointer	Longword INPUT pointer

`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.

$0002 Driver_Read

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:

Offset (direct-page)		Size and type
$00	deviceNum	Word INPUT value
$02	callNum	Word INPUT value
$04	bufferPtr	Longword INPUT pointer
$08	requestCount	Longword INPUT value
$0C	transferCount	Longword RESULT value
$10	blockNum	Longword INPUT value
$14	blockSize	Word INPUT value
$16	fstNum	Word INPUT value
$18	volumeID	Word INPUT value
$1A	cachePriority	Word INPUT value
$1C	cachePointer	(used indirectly)
$20	dibPointer	Longword INPUT pointer

`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 tranferred. 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 $0000–$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, but is set automatically by the Cache Manager. See Chapter 9, "Cache Control," for details.

`dibPointer` Longword input pointer: points to the device information block (DIB) for the device being accessed.

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 address), and with the transfer count indicating the total number of bytes that were successfully read from the device.

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 increment the transfer count as each byte is received from the device. The driver terminates the I/O transaction when the transfer count equals the request count.

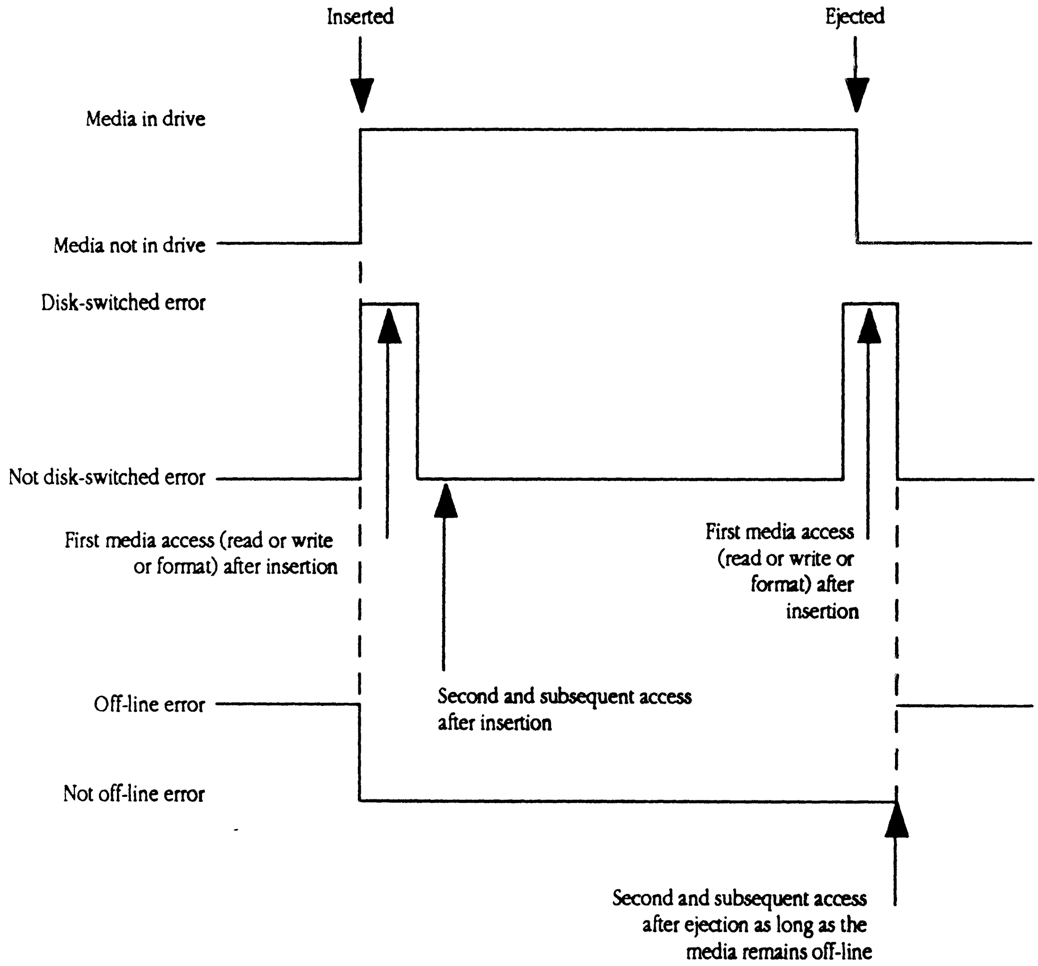
Block device requirements

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 zero by 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 11-3.

■ **Figure 11-3** 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 Support." The calls themselves are described in Chapter 12, "System Service Calls."

$0003 Driver_Write

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:

Offset (direct-page)		Size and type
$00	deviceNum	Word INPUT value
$02	callNum	Word INPUT value
$04	bufferPtr	Longword INPUT pointer:
$08	requestCount	Longword INPUT value
$0C	transferCount	Longword RESULT value
$10	blockNum	Longword INPUT value
$14	blockSize	Word INPUT value
$16	fstNum	Word INPUT value
$18	volumeID	Word INPUT value
$1A	cachePriority	Word INPUT value
$1C	cachePointer	(used indirectly)
$20	dibPointer	Longword INPUT pointer

`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 tranferred. 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 automatically by the Cache Manager. See Chapter 9, "Cache Control," for details.

`dibPointer` Longword input pointer: points to the device information block (DIB) for the device being accessed.

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 address), and with the transfer count indicating the total number of bytes that were successfully written to the device.

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 zero by 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 11-3 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 Support." The calls themselves are described in Chapter 12, "System Service Calls."

$0004 Driver_Close

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:

Offset (direct-page)		Size and type
$00	deviceNum	Word INPUT value
$02	callNum	Word INPUT value
$04	bufferPtr	(not used)
$08	requestCount	(not used)
$0C	transferCount	(not used)
$10	blockNum	(not used)
$14	blockSize	(not used)
$16		(not used)
$18	volumeID	(not used)
$1A	cachePriority	(not used)
$1C	cachePointer	(not used)
$20	dibPointer	Longword INPUT pointer

`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 device information block (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.

$0005 Driver_Status

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:

Offset (direct-page)		Size and type
$00	deviceNum	Word INPUT value
$02	callNum	Word INPUT value
$04	statusListPtr	Longword INPUT pointer
$08	requestCount	Longword INPUT value
$0C	transferCount	Longword RESULT value
$10	blockNum	(not used)
$14	blockSize	(not used)
$16	controlCode	Word INPUT value
$18	volumeID	(not used)
$1A	cachePriority	(not used)
$1C	cachePointer	(not used)
$20	dibPointer	Longword INPUT pointer

`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.

`statusListPtr` 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 $8000 through $FFFF. These are the currently defined status codes and subcalls:

$0000	Get_Device_Status
$0001	Get_Config_Parameters
$0002	Get_Wait_Status
$0003	Get_Format_Options
$0004	Get_Partition_Map
$0005–$7FFF	(reserved)
$8000–$FFFF	(device-specific)

`dibPointer` Longword input pointer: points to the device information block (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 zero before calling the device driver. If the call is successful, the device driver should set the transfer count to the number of bytes returned.

- *Disk-switched:* 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 offline 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.

Get_Device_Status (Driver_Status subcall)

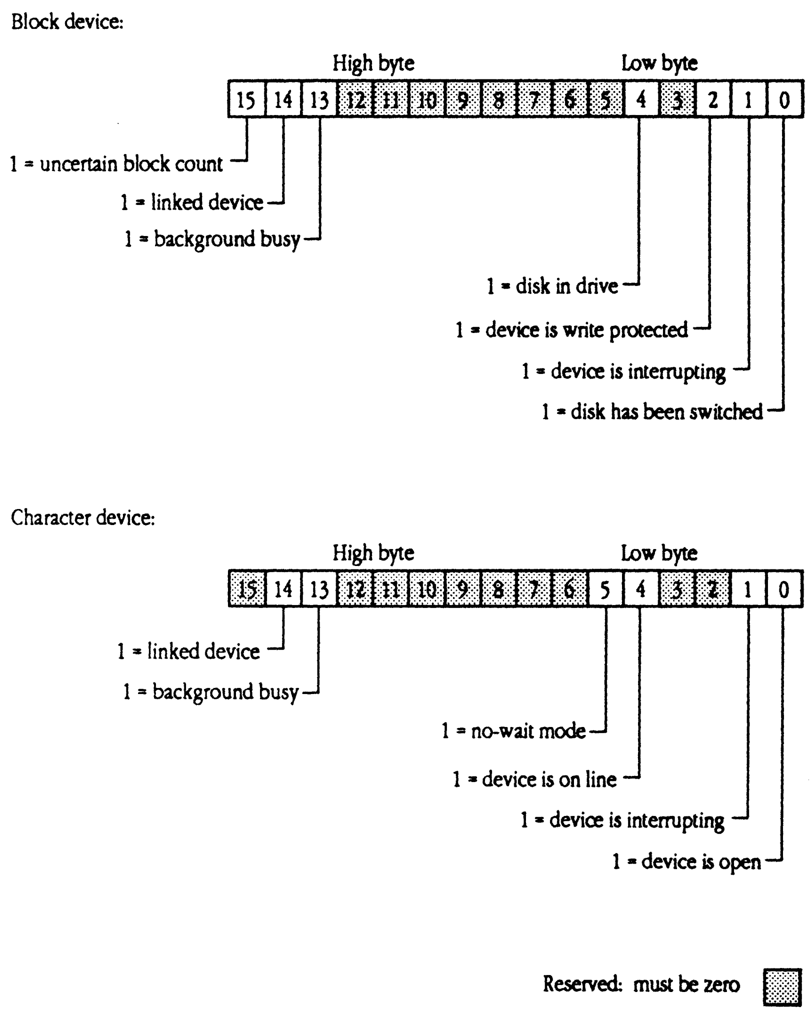
Status code = $0000.

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. The status list is 6 bytes long. This is its format:

Offset		Size	Description
$00	`statusWord`	Word	The status word (see the following definition)
$02	`numBlocks`	Longword	The number of blocks on the 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 11-2.

■ **Figure 11-2** Device status word

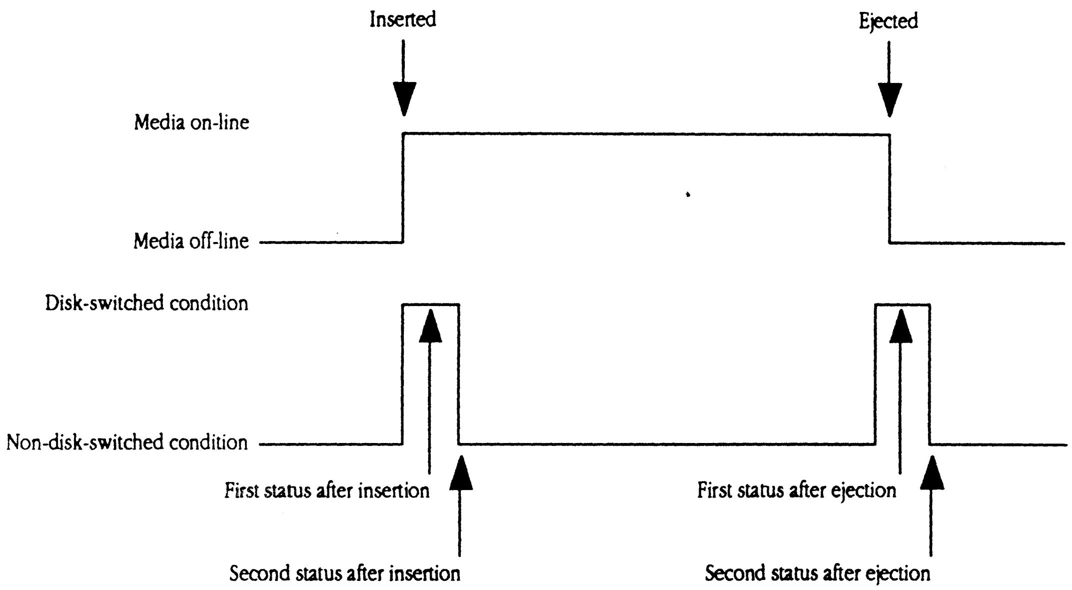


Character device drivers should return a block count of zero.

If the 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 11-4 illustrates the disk-switched condition.

■ **Figure 11-4** Disk-switched condition



◆ *Error codes:* 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.

Get_Config_Parameters (Driver_Status subcall)

Status code = $0001.

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:

Offset		Size	Description
$00	Length word	Word	The length of the list (in bytes)
$02	Configuration parameters list	—	The configuration list

Get_Wait_Status (Driver_Status subcall)

Status code = $0002.

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

The status list for this subcall contains $0000 if the device is operating in wait mode, $8000 if it is operating in no-wait mode. The size of the status list is 2 bytes:

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

Get_Format_Options (Driver_Status subcall)

Status code = $0003.

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

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 Set_Format_Options followed by Format_Device. Devices that do not support media variables should return a transfer count of zero 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 4-word header followed by a set of entries, each of which describes a formatting option. The status list looks like this:

Offset		Size	Description
$00	numOptions	Word	Number of format-option entries in the list
$02	numDisplayed	Word	Number of options to be displayed
$04	recommendedOption	Word	Recommended default formatting option
$06	currentOption	Word	The option with which the currently on-line media was formatted
$08	formatOption1	(16 bytes)	The first format-options entry
$0C	⋮		
	formatOptionN	(16 bytes)	The last format-options entry

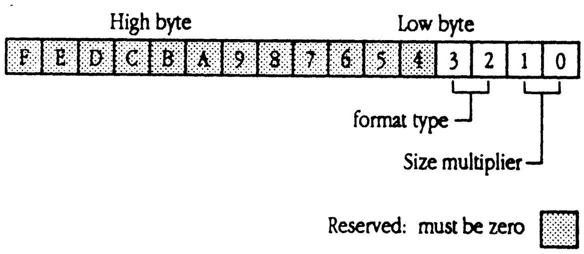
Of the total number of options in the list, one or more may be displayed on the initialization dialog presented to the user when initializing a disk (see the calls Format and EraseDisk in Chapter 7 of Volume 1). The options to be displayed are always the first ones in the list. (Undisplayed options are 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 it, it should. 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	The number of this option
$02	`linkRefNum`	Word	Number of linked option
$04	`flags`	Word	(see definition below)
$06	`blockCount`	Longword	No. of blocks supported by the 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	(for any file system)
01	Apple format	(for an Apple file system)
10	Non-Apple format	(for other file systems)
11	(not valid)	

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

00	`mediaSize` is in bytes
01	`mediaSize` is in Kbytes
10	`mediaSize` is in Mbytes
11	`mediaSize` is in Gbytes

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 is option 1
$0001	Current media is 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 kilobytes

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 kilobytes

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 kilobytes

Get_Partition_Map (Driver_Status subcall)

Status code = $0004.

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

Device-specific 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 $8000–$FFFF.

$0006 Driver_Control

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:

Offset (direct-page)		Size and type
$00	deviceNum	Word INPUT value
$02	callNum	Word INPUT value
$04	controlListPtr	Longword INPUT pointer
$08	requestCount	Longword INPUT value
$0C	transferCount	Longword RESULT value
$10	blockNum	(not used)
$14	blockSize	(not used)
$16	controlCode	Word INPUT value
$18	volumeID	(not used)
$1A	cachePriority	(not used)
$1C	cachePointer	(not used)
$20	dibPointer	Longword INPUT pointer

`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: This parameter 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	Reset_Device
$0001	Format_Device
$0002	Eject_Medium
$0003	Set_Configuration_Parameters
$0004	Set_Wait_Status
$0005	Set_Format_Options
$0006	Assign_Partition_Owner
$0007	Arm_Signal
$0008	Disarm_Signal
$0009	Set_Partition_Map
$000A–$7FFF	(reserved)
$8000–$FFFF	(device-specific)

`dibPrinter`	Longword input pointer: points to the device information block (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 zero; the driver should ignore the control list and request count entirely, and pass a transfer count of zero.

Reset_Device (Driver_Control subcall)

Control code = $0000.

The Reset_Device 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.

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

If successful, this call has a transfer count of zero and no error is returned. Request count should be ignored; the control list is not used.

Format_Device (Driver_Control subcall)

Control code = $0001.

The Format_Device 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. Format_Device simply prepares all blocks on the medium for reading and writing.

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

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

If successful, this call has a transfer count of zero. Request count should be ignored; the control list is not used.

Eject_Medium (Driver_Control subcall)

Control code = $0002.

The Eject_Medium 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 ofthis 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 zero. Request count should be ignored; the control list is not used.

Set_Config_Parameters (Driver_Control subcall)

Control code = $0003.

The Set_Config_Parameters 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:

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

The structure of the configuration list is device-dependent. See Chapter 9, "Cache Control," for more information.

This subcall is most typically used in conjunction with the status subcall Get_Config_Parameters. 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 to send the new parameters to the device driver.

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

Set_Wait_Status (Driver_Control subcall)

Control code = $0004.

The Set_Wait_Status subcall sets 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, or a newline character is encountered during the read and newline mode is enabled. In no-wait mode, a read call returns immediately after reading the available characters, with a transfer count indicating the number of characters returned. If one or more characters was available, the transfer count has a nonzero value; if no character was available, the transfer count is zero.

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

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

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

Set_Format_Options (Driver_Control subcall)

Control code = $0005.

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

The Set_Format_Options subcall sets these media-specific formatting parameters prior to the execution of a Format_Device subcall. Set_Format_Options does not itself cause or require a formatting operation. The control list for Set_Format_Options consists of two word-length parameters:

Offset		Size	Description
$00	`formatRefNum`	Word	The number of the format option
$02	`interleaveFactor`	Word	The override interleave factor (if nonzero)

The format option number (`formatOptionNum`) specifies a particular **format-options entry** from the driver's **format options list** (returned from the Driver_Status subcall Get_Format_Options). The format-option entry has this format:

Offset		Size	Description
$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 Driver_Status subcall Get_Format_Options, 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 or FST can take the following steps (if you use the Format call, the Initialization Manager takes these steps for you):

1. Issue a (Driver_Status) Get_Format_Options 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) Set_Format_Options subcall to the device, specifying the desired format option.
3. Issue a (Driver_Control) Format_Device subcall to the device.

△ **Important** Set_Format_Options is meant to set the parameters for *one* subsequent formatting operation only. Drivers should expect Set_Format_Options 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 Set_Format_Options subcall applies to block devices only; character devices should return error $20 (invalid request) if they receive this call.

Assign_Partition_Owner (Driver_Control subcall)

Control code = $0006.

The Assign_Partition_Owner 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 the Driver_Control subcall Erase_Disk or Format_Device 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. 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	length	Word	The length of the name (in bytes)
$02	ownerName	—	The 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 chapter in *Inside Macintosh, Volume V.*

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

Arm_Signal (Driver_Control subcall)

Control code = $0007.

The Arm_Signal subcall provides a means for a device driver to install a signal handler into the GS/OS signal handler list. This is the control list for the subcall:

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

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

Disarm_Signal (Driver_Control subcall)

Control code = $0008.

The Disarm_Signal 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.

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

Set_Partition_Map (Driver_Control subcall)

Status code = $0009.

This call passes to a device, in the control list, the partition map for a partitioned disk or other medium. 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 DStatus call into a GS/OS driver call. The status code must be in the range $8000–$FFFF.

$0007 Driver_Flush

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:

Offset (direct-page)		Size and type
$00	deviceNum	Word INPUT value
$02	callNum	Word INPUT value
$04	bufferPtr	(not used)
$08	requestCount	(not used)
$0C	transferCount	(not used)
$10	blockNum	(not used)
$14	blockSize	(not used)
$16		(not used)
$18	volumeID	(not used)
$1A	cachePriority	(not used)
$1C	cachePointer	(not used)
$20	dibPointer	Longword INPUT pointer

`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 device information block (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.

$0008 Driver_Shutdown

Description Driver_Shutdown is issued by GS/OS in preparation for removng 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:

Offset (direct-page)		Size and type
$00	deviceNum	Word INPUT value
$02	callNum	Word INPUT value
$04	bufferPtr	(not used)
$08	requestCount	(not used)
$0C	transferCount	(not used)
$10	blockNum	(not used)
$14	blockSize	(not used)
$16		(not used)
$18	volumeID	(not used)
$1A	cachePriority	(not used)
$1C	cachePointer	(not used)
$20	dibPointer	Longword INPUT pointer

`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 device information block (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 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 those that 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 11-2 shows those calls. 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 11-3 shows those calls.

■ Table 11-2 Supervisory-driver calls available to device drivers

Call no.	Supervisor no.	Call name	Explanation
$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 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 11-2 with nonzero supervisor numbers appear also in Table 11-3; they are the only calls in Table 11-2 that are actually handled by supervisory drivers.

■ Table 11-3 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

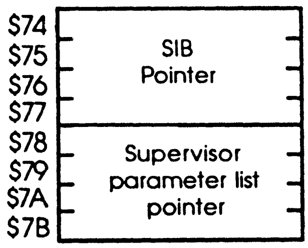
A device driver or other program makes a call to a supervisory driver by making the system service call SUP_DRVR_DISP (see Chapter 12). Parameters for supervisory-driver calls are passed both in registers and in locations $74–$7B on the GS/OS direct page, called the **supervisor direct page** (Figure 11-5).

A small workspace is available for device-driver use on the GS/OS direct page. Locations $5A through $5F are available for device drivers; locations $66 through $6B are shared by device drivers and supervisory drivers (and may be corrupted by either a driver call or supervisory driver call). This workspace is not permanent; it may be corrupted between driver calls. Supervisory drivers should not permanently modify any GS/OS direct-page location that is not within the bounds of that workspace. A supervisory driver requiring direct-page space should save and restore the contents of any other direct-page location that it uses.

Note also that the parts of the GS/OS direct page used by driver calls (locations $00–$23) are available for use in device-specific supervisory-driver calls.

■ **Figure 11-5** The supervisor direct page: parameter space

Offset (direct-page) **Description**



Longword pointer to the supervisor information block (SIB)

Longword pointer to a device-specific parameter list

On input to the supervisory driver, the A register (accumulator) contains the **supervisor number,** which specifies the supervisory driver to whom 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 11-2) are handled by the supervisor dispatcher; calls with a nonzero supervisor number (see Table 11-3) are handled by supervisory drivers.

The rest of this chapter documents the currently defined supervisory-driver calls.

$0000 Get_Supervisor_Number

Description

When it is started up, a device driver 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 "AppleTalk" or "SCSI") of its supervisory driver to this call; the call then returns the supervisor number.

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` = zero)
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 zero 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	AppleTalk 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 zero 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 (no device 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.

$0000 Supervisor_Startup

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 the 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 zero 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, just 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.

$0001 Set_SIB_Pointer

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 the 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 zero 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.

$0001 Supervisor_Shutdown

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 the supervisor direct page (`sibPtr`) *plus:*
A register = `supervisorNumr`
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 suprvisory driver needs it.

Notes GS/OS shuts down supervisory drivers only after shutting down all device drivers.

$0002–$FFFF Driver-specific calls

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 the GS/OS direct page, including the supervisor direct page, *plus:*
A register = `supervisorNum`
X register = `callNum` ($0002–$000F)

Output:

Contents of the 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–$000F for this call.

`supervisorNum` Word input value: this input A-register 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 ipoints to the supervisor information block (SIB) for the supervisory driver being accessed. This parameter is set up by the supervisor dispatcher, in case the supervisory driver needs it.

Notes Not only `sibPtr`, but the rest of the supervisor direct page (the supervisor parameter-list pointer) and all of the device-driver portion of the GS/OS direct page are available for device drivers and supervisor drivers to use as parameters for device-specific supervisory-driver calls. However, those drivers should not permanently modify any GS/OS direct-page location that is not within the bounds of the small workspace; see "About Supervisory Calls" in this chapter. A supervisory driver requiring direct-page space should save and restore the contents of any other direct-page location that it uses.

Driver error codes

GS/OS can recognize the device driver error codes listed in Table 11-4. 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 11-4 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 is unavailable

Chapter 12 **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, or 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 12-1.

■ **Table 12-1** System service calls

Call name	Dispatch 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
SWAP_OUT	$01FC34	Marks a volume as off-line
SET_SYS_SPEED	$01FC50	Controls processor execution speed
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

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, or 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. Calls in this chapter are ordered alphabetically by name; for cross-reference, Table 12-1 shows the same calls ordered by call number (system service table entry point).

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 8. Figure 12-1 shows the GS/OS direct-page parameters.

■ **Figure 12-1** GS/OS direct-page parameter space
Used by driver calls and some system service calls

Offset (direct-page)		Description
$00	`deviceNum`	The number of the device to whom the call is made
$02	`callNum`	The number of the call being made
$04	`bufferPtr`	Pointer to a buffer for reading or writing data
$08	`requestCount`	The number of bytes to transfer to or from driver
$0C	`transferCount`	The number of bytes transferred by the call
$10	`blockNum`	The number of the block to start a read or write at
$14	`blockSize`	How many bytes per block for this device
$16	`fstNum OR code`	This device's FST number *or* status code *or* control code
$18	`volumeID`	The VRN for blocks on this device
$1A	`cachePriority`	What sort of caching to implement
$1C	`cachePointer`	Pointer to the current block in the cache
$20	`dibPointer`	Pointer to the DIB for this device

$01FC08 CACHE_ADD_BLK

Description

This routine attempts to add the requested block into the cache. The block is added at the start of the LRU chain (= most recently used). If there is not enough room in the cache, the block(s) at the end of the chain (= 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:

```
cachePtr
```

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; the block was added to the cache.
If c = 1:	Error; the block was not added to the cache.

$01FC04 CACHE_FIND_BLK

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: by device number (used by drivers), or 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:

`cachePtr`	Pointer to the start of the block in the 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; the block is in the cache.
If c = 1:	Error; the block is not in the cache.

$01FCBC DYN_SLOT_ARBITER

Description

This call might, in the future, provide support for dynamic switching between devices on internal and external slots. At the time of publication, the call indicates only whether the slot is available.

Parameters

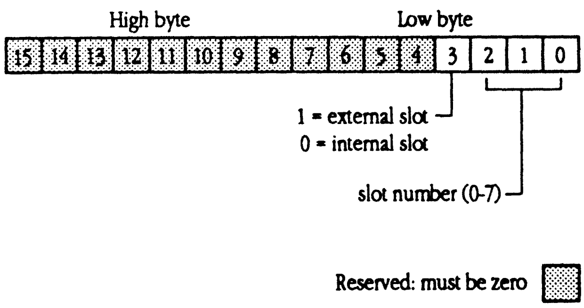
Input:

A register:	Requested slot
X register:	Undefined
Y register:	Undefined

Return:

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:



Errors

Carry flag set if request denied

$01FCA8 INSTALL_DRIVER

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 device call (or at the end of the current one if a device call is in progress). 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 error) occurs, it is not possible to install any drivers; if error $29 (driver 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.

$01FC70 MOVE_INFO

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):

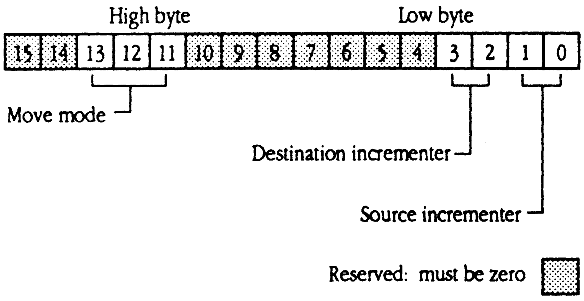
Parameters on stack	Size and type	Description
⋮		
previous contents		
`sourcePtr`	Longword pointer	Pointer to the source buffer
`destinationPtr`	Longword pointer	Pointer to the destination buffer
`requestCount`	Longword value	Number of bytes to transfer
`commandWord`	Word value	Flags (see description)
⋮	***<— stack pointer***	

The high bytes of `sourcePtr`, `destinationPtr`, and `transferCount` must be zero.

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 covers 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

$01FC90 SET_DISKSW

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` The device number of the 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.

Notes

If the current device is a linked device, then SET_DISKSW also calls CACHE_DEL_VOL and SWAP_OUT for each of the devices linked to the current device, if the device dispatcher-maintained disk-switched flag is not currently set.

Errors

None

$01FC50 SET_SYS_SPEED

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.

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.

Errors

None

$01FC88 SIGNAL

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 Arm_Signal 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 10 ("Handling Interrupts and Signals") of this Volume.

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: the priority-ranking of the signal, with $0000 being the lowest priority and $FFFF being the highest.

Signal-handler address: the 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

$01FCA4 SUP_DRVR_DISP

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	Get_Supervisor_Number
$0001	$0000	Set_SIB_Pointer
$0002–$FFFF	$0000	(Reserved)

The following calls are dispatched by the supervisor dispatcher to a supervisory driver:

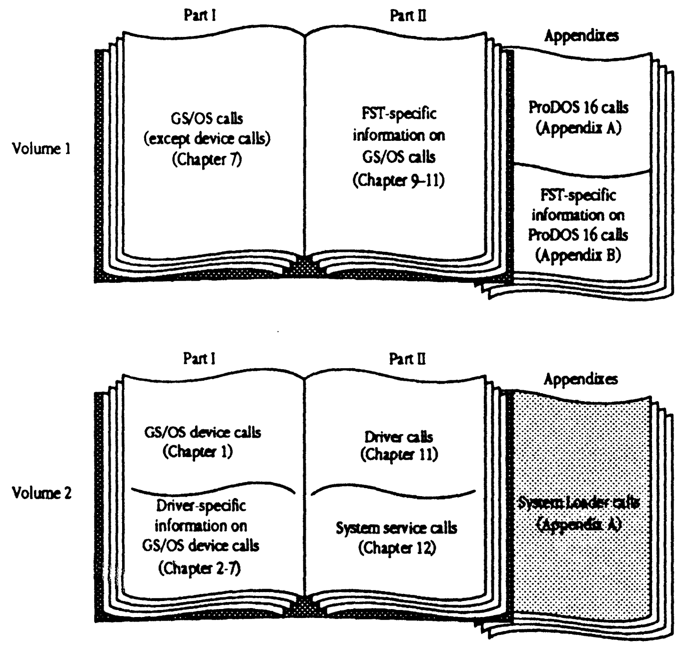
Call No.	Sup. No.	Function
$0000	(Nonzero)	Supervisor_Startup
$0001	(Nonzero)	Supervisor_Shutdown
$0002–$FFF	(Nonzero)	(Driver-specific calls)

These subcalls and other supervisory–driver calls are described in detail in Chapter 11, "GS/OS Driver Call Reference."

Errors

$28 no device connected

Appendixes


Part I
Part II
Appendixes
Volume 1
GS/OS calls (except device calls) (Chapter 7)
FST-specific information on GS/OS calls (Chapter 9-11)
ProDOS 16 calls (Appendix A)
FST-specific information on ProDOS 16 calls (Appendix B)
Part I
Part II
Appendixes
Volume 2
GS/OS device calls (Chapter 1)
Driver-specific information on GS/OS device calls (Chapter 2-7)
Driver calls (Chapter 11)
System service calls (Chapter 12)
System Loader calls (Appendix A)

Appendix A **The System Loader**

Because the Apple IIGS has a large amount of available memory, a flexible, dynamic facility for loading program files is required. Programs should be able to be loaded in any available location in memory. The burden of determining where to load a program should be on the system, not on the application writer. Furthermore, programs should be able to be broken into smaller program segments that can be loaded independently.

To provide these capabilities, GS/OS comes with a relocating segment loader called the **System Loader.** The System Loader provides a very powerful and flexible facility that is not available on standard Apple II computers. ■

How the System Loader works

Apple II computers running under ProDOS 8 have a very simple program loader. The loader is the part of the boot code that searches the boot disk for the first **System file** (any file of ProDOS file type $FF whose name ends with ".SYSTEM") and loads it into location $2000. If a program wants to load another program, it has to do all the work by making ProDOS 8 calls.

Some programming environments such as Apple II Pascal and AppleSoft BASIC provide loaders for programs running under them. The AppleSoft loader loads either System files, BASIC files, or binary code files. All these files are loaded either at a fixed address in memory or at an address specified in the file.

The Apple IIGS System Loader under GS/OS can load programs in any available part of memory, relieving the application writer of deciding where to put the code and how to make it execute properly at that location. Furthermore, the System Loader can load individual segments rather than whole files, either at program start or during execution.

The System Loader loads programs or program segments by first calling the Memory Manager to find available memory. It loads each segment independently and performs relocation during the load as necessary. Therefore, a large application can be broken up into smaller program segments, each of which is put into separate locations in memory. The application's segments can also be loaded dynamically, as they are referenced, rather than at program boot time. Additionally, the System Loader can be called by the application itself to load and unload program (or data) segments.

Definitions

The System Loader processes **load files,** generated from **object files** by a **linker.** Definitions of these and related terms may help make the following discussion clearer:

Object files are the output from an assembler or compiler and are the input to a linker.

A **linker** is the program that combines object files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

Load files are the output of a linker and contain memory images, which the System Loader can load into memory. There are several types of load files, reflecting the types of programs they contain.

The **System Loader** is the part of system software that reads the files generated by the linker and loads them into memory (performing relocation if necessary).

Relocation is the process of modifying a load file in memory so that it will execute correctly. It consists of patching operands to reflect the code's current memory location.

Library files are special object files, containing general program segments that the linker can search.

Run-time library files are special load files, containing general program segments that can be loaded as needed by the System Loader and shared between applications.

Object module format (OMF) is the general format used in object files, library files, and load files.

An **OMF file** is a file in object module format (an object file, library file or load file).

A **segment** is an individual component of an OMF file. Each file contains one or more segments; object files contain object segments, and load files contain load segments.

A **controlling program** is a program that uses System Loader calls to load and execute another program, and is responsible for shutting down the program when it exits. Operating systems and shells are controlling programs.

Segments and the System Loader

The System Loader processes only those files that conform to the Apple IIGS definition of a load file (see Appendix B). A load file consists of load segments, each of which can be loaded independently. The load segments are numbered sequentially from 1.

Certain load segments are **static** load segments. They are loaded into memory at program start (initial load) and must stay in memory until program completion.

The other general type of load segment is **dynamic**. Dynamic segments are loaded not at boot time but during program execution. This can happen automatically (by means of the jump table mechanism) or manually (at the specific request of the application). When dynamic segments are not needed by a program, they can be **purged** (their contents deallocated) by the program.

Load segments can have several other attributes; see Appendix B for a complete list of attributes.

Segments are classified numerically by kind (the value of the KIND field in the segment header; see Appendix B). In addition to segments containing program code or data, there are several special kinds of load segments:

- The jump-table segment (KIND=$02), when loaded into memory, becomes part of the **jump table**. The jump table provides a mechanism whereby segments in memory can trigger the loading of other segments not yet in memory. The jump table is described later in this section, under "Loader Data Structures."
- The pathname table segment (KIND=$04) contains information about the run-time library files that are referenced. The pathname table is described later in this section, under "Loader Data Structures"; run-time library files are described in Appendix B.
- Initialization segments (KIND=$10) in a load file are used for code that is to be executed before all the rest of the load segments are loaded.
- The direct-page/stack segment (KIND=$12) defines the application's direct–page and stack requirements. This segment is loaded into bank $00 and its starting address and length are passed to the controlling program, which in turn sets the Direct register and Stack pointer to the start and end of this segment before transferring control to the program.

If the System Loader is called to perform the initial load of a program, it loads all the static load segments and the jump table and pathname table segments (if they exist). The loader also constructs a RAM-based memory–segment table during this process. The memory–segment table is described later in this section, under "Loader Data Structures."

◆ *References to dynamic segments:* During the initial load, the System Loader has all the information needed to resolve all intersegment references between the static load segments. But during the dynamic loading of dynamic load segments, it can only resolve references in the dynamic load segment to the already loaded static load segments. Therefore, the general rule is that static segments can be referenced by any type of segment but dynamic segments can only be referenced through JSL calls through the jump table.

- *Unmounted volumes:* If the System Loader references a file on a volume that is not mounted, GS/OS either returns with error $45 (volume not found) or displays a mount-volume message (depending on the state of the system preferences at the time of the call; see "SetSysPrefs" in Chapter 7 of Volume 1). If a mount message is displayed, GS/OS handles the user interface and returns control to the System Loader only when the I/O operation is complete or the user has canceled the request for the mount. For all user-callable System Loader functions, system preferences are controlled by the user. For the internal Jump-Table Load function, the System Loader sets system preferences to display mount messages and then restores them to their original state.

The System Loader and the Memory Manager

The System Loader and the Memory Manager work together closely. Depending on how the System Loader defines a segment, the Memory Manager needs to allocate a memory block for that segment with the proper properties.

The System Loader defines load segments as static or dynamic (already defined), and as **absolute** (needs to be loaded at a specific address), **relocatable** (can be loaded at any address, but cannot be moved once loaded), or **position-independent** (can be loaded anywhere and then moved anywhere after loading). The Memory Manager uses its own terminology to describe memory blocks; see the chapter "Memory Manager" in the *Apple IIGS Toolbox Reference*. Loader and Memory Manager terminology are related in this way:

- When the System Loader loads a static segments, it calls the Memory Manager to allocate a corresponding memory block that is **unpurgeable** (purge level = 0; the Memory Manager cannot remove it from memory) and **locked** (the Memory Manager cannot move it unless it is first unlocked).
- When the loader loads a dynamic segment, the Memory Manager allocates a memory block that is marked as **purgeable** (purge level >0) but locked.
- Position-independent segments are placed in blocks that are **movable** (the Memory Manager can change their locations in memory if they are not locked); all other segments (whether static or dynamic) are placed in blocks that are **fixed** (not movable, even if not locked).

The typical load segment, which is relocatable, is loaded into a memory block having these attributes:

Locked
Fixed
Purge level=0 (if static)
Purge level=3 (if dynamic)

When the System Loader *unloads* a specific segment, it calls the Memory Manager to make the corresponding memory blocks purgeable.

To unload *all* of a program's segments (all segments associated with a particular user ID), a controlling program calls the System Loader's UserShutdown routine—which in turn calls the Memory Manager—to purge all the program's dynamic segments and make all its static segments purgeable. The purpose of this is to keep the essential parts of an application in memory, in case it needs to be rerun in the near future. Keeping programs **dormant** in memory, and executing them again with the System Loader's Restart routine, can greatly speed up execution of a program selector such as the Finder. However, once the Memory Manager has to actually purge one of the static segments of a dormant program, it is incomplete and must be reloaded from file (with InitialLoad) before running.

◆ *Note:* If many incomplete (partially purged) applications are in memory, the system may get bogged down with NIL memory handles. To avoid this situation, the System Loader disposes all NIL memory handles it knows about before executing every InitialLoad or Restart call.

Depending on the ORG, KIND, BANKSIZE, and ALIGN fields in the segment header (see "OMF and the System Loader," later in this chapter), other memory-block attributes are possible:

■ **Table A-1** Segment characteristics and memory-block attributes

Segment header attribute	Memory-block attribute
If ORG>0	Fixed address
If BANKSIZE=$10000	May not cross bank boundary
If 0<Align Factor[1]<=$100	Page aligned
If Align Factor[1]>$100	Bank aligned (forced by System Loader[2])
Bit 13 of KIND=0	Fixed block (not moveable)
Bit 12 of KIND=1	May not use special memory
Bit 11 of KIND=1	Fixed bank (not fixed address)
Bit 8 of KIND=1	Bank-relative (fixed address in any bank); forced by System Loader
KIND = 12	Fixed bank (bank $00), page aligned (Direct page/stack segment)

[1]If 0<BANKSIZE<$10000, Align factor=the greater of BANKSIZE or ALIGN; if BANKSIZE has any other value (except for $10000), Align factor=ALIGN.
[2]Although the Memory Manager does not provide bank alignment, the System Loader forces it in this instance by requesting successive fixed-address blocks at the beginning of each bank until successful.

A memory block can be made purgeable (unloaded) by a call to the System Loader. However, other memory-block attributes must be changed through Memory Manager calls. Since the memory handle for a memory block is stored in the memory-segment table, Memory Manager information is accessible. Other memory block information that may be useful to a program is as follows:

Start location
Size of segment
User ID
Purge Level: 0 = Unpurgeable
1 = Least purgeable
3 = Most purgeable

Note also that if the memory handle is NIL (its address value is 0), the memory block has been purged.

OMF and the System Loader

Object module format (OMF) defines the internal format for Apple IIGS object files, library files, and load files. OMF files consist of **segments**, each of which has a **segment header** and a series of **OMF records.** As Table A-1 shows, a load segment's characteristics, the type of memory block it inhabits, and its segment-header values are all closely interrelated. OMF is documented in detail in Appendix B.

Object module format includes general capabilities beyond the requirements of the Apple IIGS computer. The System Loader, on the other hand, is designed specifically for the Apple IIGS. Therefore, there are certain OMF features that the System Loader either does not support or supports in a restricted manner. Here are some examples (see Appendix B for definitions of OMF features):

- The NUMSEX field of the segment header must be 0.
- The NUMLEN field of the segment header must be 4.
- The BANKSIZE field of the segment header must be <=$10000.
- The ALIGN field of the segment header must be <=$10000.

If any of the above is not true, the System Loader returns error $110B (segment is foreign). The BANKSIZE and ALIGN restrictions are enforced by the linker, and violations of them are unlikely in a load file.

The System Loader uses BANKSIZE and ALIGN to force memory alignment of segments as follows:

- Under OMF, ALIGN and BANKSIZE can be any power of 2. But the Memory Manager does not support so general a requirement. The Memory Manager can currently only be told that a memory block must be page aligned or must not cross a bank boundary. To force bank alignment where needed, the System Loader uses this method:
- Any value of BANKSIZE other than 0 and $10000 results in a memory block that is either page aligned (if BANKSIZE<=$100) or bank aligned (if BANKSIZE>$100). Since the linker makes sure that the segment is smaller than BANKSIZE, the requirement that the segment not extend past the BANKSIZE boundary is met (there will be wasted space in the memory block, however).
- Any value of ALIGN is bumped to either page alignment or bank alignment.
- If there is a BANKSIZE other than 0 and $10000 and a non-zero ALIGN, the greater of the two determines the alignment to be used.

Loader data structures

The System Loader creates several types of data structures to track which segments are in memory or need to be loaded. This section briefly describes the structures.

Memory-segment table

The memory-segment table is a linked list created by the System Loader. Each entry corresponds to one memory block known to the System Loader. The memory blocks are allocated by the Memory Manager when the System Loader loads segments from a load file. Each entry in the memory-segment table contains a handle to the memory block, the block's user ID, and the load-file number and load-segment number of the segment occupying the block.

The System Loader uses the memory-segment table to keep track of all its loaded segments: where they are, who owns their memory, and where on disk they came from.

Pathname table

The pathname table is created by System Loader to keep track of the pathnames associated with all load files and run-time library files it processes. The pathnames in the pathname table are fully-expanded pathnames, stored as GS/OS strings (preceded by a word-length character-count field). At initial load, the System Loader adds the pathname specified in the Initial Load call to the pathname table. During the load, if the System Loader comes across a pathname segment (KIND=$04), it adds all the pathname entries to the pathname table. Pathname segements are created by the linker.

Each entry in the pathname table includes the pathname, load-file number, user ID, and address and size of direct-page/stack space for a particular load file. It also includes other information pertinent to run-time libraries. The System Loader uses the pathname table to locate files on disk that are identified by load-file number in the loader's other tables.

Jump table

The jump table is the data structure that makes it possible for programs to reference dynamic segments (segments that are loaded into memory only when they are needed). The jump table consists of the jump-table directory and one or more jump-table segments. The jump-table directory is a linked list constructed by the System Loader. It contains a handle to and the user ID of each jump-table segment (KIND=$02) that the System Loader has encountered while loading load segments. Any load file or run-time library file may contain a jump-table segment.

Jump-table segments are created by the linker. When processing an object file, each time the linker encounters a JSL to an external dynamic segment, it does the following:

1. It creates an entry in the jump-table segment.
2. It links the JSL in the object file to that jump-table segment entry.

Each entry in the jump-table segment contains the load-file number and load-segment number of the referenced dynamic segment, the offset of the referenced location within that segment, and a JSL instruction to a location within the System Loader that will take care of loading and executing that segment when called.

During program execution, the jump table functions this way:

1. When the JSL instruction actually executes, control passes to the jump–table entry, and then to the System Loader. The System Loader extracts the segment information from the jump–table entry and the file information from the pathname table.
2. The System Loader loads the dynamic segment, changes the JSL instruction in the jump table to a JML to the proper location in the just loaded segment, and transfers control to that location.
3. Typically, the location in the loaded segment is a subroutine. When it exits with an RTL, control is eventually transferred to the location following the original JSL instruction, as expected.

Restarting, reloading, and dormant programs

By working closely with the Memory Manager and GS/OS, the System Loader provides a mechanism whereby programs can stay in memory after they terminate and can be relaunched very quickly if they are called again.

When making the GS/OS Quit call, an application always specifies (1) whether it is capable of being relaunched from memory, and (2) whether it wishes to quit to another specific application, and—if so—whether it wants to be relaunched after that application quits. GS/OS notes those specifications and treats a quitting program accordingly:

- If a quitting application is capable of being restarted from memory—that is, if it does not require initialization data to be loaded from disk—GS/OS puts it into a dormant state with the System Loader's UserShutdown call: it keeps all the application's static segments in memory so that the application can start up very quickly if it is ever called again. When that application is relaunched from memory, it is said to be **restarted.** GS/OS uses the System Loader's Restart call for this.

- If an application will be relaunched at a future time, the System Loader keeps track of its pathname, so that when the time comes it can be **reloaded**—loaded and executed automatically from disk, using the System Loader's InitialLoad (or InitialLoad2) call. Of course, if the program is already in memory in a dormant state, it can simply be restarted.

A dormant application's static segments are not protected; if the Memory Manager needs memory, it can purge one or more of them. Once that happens, the application is no longer dormant; it must be reloaded from disk if it is ever relaunched.

◆ *Reload segments and restartability:* In some programming languages it is impractical to make completely restartable applications; initialization data must be read from disk every time a program is launched. To permit restartability in such cases, the System Loader allows for **reload segments,** load segments that are always loaded from disk at program launch, even if the program is in a dormant state. Therefore, if a program can be designed with all its initialization information in one or more reload segments, it can call itself restartable when it quits.

Making System Loader calls

Because the System Loader is a Apple IIGS tool set, its functions are called by making stack-based calls through the Apple IIGS Tool Locator. The calling sequence for System Loader functions is the standard tool-calling sequence:

1. First, push space for the output parameters (if any) onto the stack.
2. Push all input parameters *in the order specified in the call descriptions.*
3. Execute this call block (syntax in this example is for APW):

```
ldx    #$11+FuncNum|8
jsl    Dispatcher
```

   where `FuncNum` is the System Loader function number (the number of the call), $11 is the tool number for the System Loader, and `Dispatcher` is the Tool Locator entry point.
4. Upon return from the call, the A register contains the call status (zero if no error, error number otherwise), and the carry flag is set if an error has occurred.
5. If there is output, pull each output parameter off the stack *in the order specified in the call descriptions.*

Table A-2 lists and briefly describes the System Loader calls available to applications (plus its standard tool-set calls, some of which are not available to applications). The calls in Table A-2 are in numerical order by call number, except that newer calls that use GS/OS-specific data structures (such as InitialLoad2) are listed next to their ProDOS-16-compatible counterparts (such as initialLoad).

The rest of this appendix consists of detailed call descriptions; they are presented in alphabetical order by call name.

■ **Table A-2** System Loader calls

Call number	Call name	Description
$01	LoaderInitialization	Initializes the loader
$02	LoaderStartup	(Does nothing)
$03	LoaderShutDown	(Does nothing)
$04	LoaderVersion	Returns loader version
$05	LoaderReset	(Does nothing)
$06	LoaderStatus	Returns loader status
$09	InitialLoad	Loads a program into memory
$20	InitialLoad2	Loads a program into memory
$0A	Restart	Re-executes a dormant program in memory
$0B	LoadSegNum	(Load segment by number:) loads a single segment
$0C	UnloadSegNum	(Unload segment by number:) unloads a single segment
$0D	LoadSegName	(Load segment by name:) loads a single segment
$0E	UnloadSeg	Unloads the segment containing a specific address
$0F	GetLoadSegInfo	Returns a segment's memory-segment table entry
$10	GetUserID	Returns the user ID for a given pathname
$21	GetUserID2	Returns the user ID for a given pathname
$11	LGetPathname	Returns the pathname for a given user ID
$22	LGetPathname2	Returns the pathname for a given user ID
$12	UserShutDown	Shuts down a program

$0F GetLoadSegInfo

Description

This function returns the memory-segment–table entry corresponding to the specified load segment. The memory-segment table is searched for the specified entry; if the entry is not found, error $1101 is returned. If the entry is found, the contents (except for link pointers to other entries) are moved into the user buffer.

Parameters

Name	Size	Description
Input		
`userID`	Word	User ID of the load segment
`fileNum`	Word	Load-file number
`segNum`	Word	Load-segment number
`buffAddr`	Longword	User buffer address

Output:

[filled user buffer]

Errors

$1101 Entry not found

$10 GetUserID

Description

This function searches the pathname table for the specified pathname. The input pathname is a standard Pascal-type string (a byte count followed by the string of characters). The pathname is first expanded to a full pathname (in GS/OS string format) before the search. If a match is found, the corresponding user ID is returned. A controlling program can use this function to determine whether to perform a Restart of an application or an InitialLoad.

Parameters

Name	Size	Description
Input:		
`pathnameAddr`	Longword	Address of pathname
Output:		
`userID`	Word	Corresponding user ID

Errors

$1101 Entry not found

$21 GetUserID2

Description This function is identical to GetUserID except that the input pathname is a GS/OS string rather than a Pascal string.

Parameters

Name	Size	Description
Input:		
`pathnameAddr`	Longword	Address of pathname
Output:		
`userID`	Word	Corresponding user ID

Errors

$1101	Entry not found

$09 InitialLoad

Description

A controlling program (such as GS/OS or a shell program) uses this call to load another program into memory, in preparation for executing it.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID to be assigned
`pathnameAddr`	Longword	Address of the load file's pathname
`flagWord`	Word	Don't-use-special-memory flag
Output:		
`userID`	Word	The user ID assigned
`startAddr`	Longword	Starting address of the program
`dPageAddr`	Word	Address of direct-page/stack buffer
`buffSize`	Word	Size of direct-page/stack buffer

Notes

If a complete user ID is specified, the System Loader uses that when allocating memory for the load segments. If the `mainID` portion of the user ID is 0, a new user ID is obtained from the User ID Manager, based on the `typeID` portion of the user ID. If the Type portion is 0, an Application type user ID is requested from the User ID Manager. User IDs are explained under "Miscellaneous Tools," in the *Apple IIGS Toolbox Reference.*

If the don't-use-special-memory flag is TRUE (nonzero), the System Loader does not load any static load segments into **special memory.** (Special memory is the part of memory equivalent to that used by a standard Apple II computer under ProDOS 8: all of banks $00 and $01 and parts of banks $E0 and $E1.) However, dynamic load segments are loaded into any available memory, regardless of the state of the don't-use-special-memory flag.

GS/OS is called to open the specified load file using the input pathname. Note that the input pathname is a Pascal string. If any GS/OS errors occurred or if the file is not a load file type ($B3–$BE), the System Loader returns the appropriate error.

If the load file is successfully opened, the System Loader adds the load file information to the pathname table and calls the Load Segment by Number function for each static load segment in the load file.

If an initialization segment (KIND=$10) is loaded, the System Loader immediately transfers control to that segment in memory. When the System Loader regains control, the rest of the static segments are loaded normally.

If the direct-page/stack segment (KIND=$12) is loaded, its starting address and length are returned as output.

If any of the static segments cannot be loaded, the System Loader aborts the load and returns an error.

After all the static load segments have been loaded, execution returns to the controlling program with the starting address of the first load segment (not an initialization segment) of the load file. Note that the controlling program is responsible for setting up the stack pointer and Direct register, and actually transferring control to the loaded program.

Errors

$1102	OMF version error
$1104	File is not load file
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign

$20 InitialLoad2

Description

This function is similar to InitialLoad except that four variations of the input information are possible.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID to be assigned
`buffAddr`	Longword	Address of the load-file pathname or load-file image
`flagWord`	Word	Don't-use-special-memory flag
`inputType`	Word	input type
Output:		
`userID`	Word	The user ID assigned
`startAddr`	Longword	Starting address of the program
`dPageAddr`	Word	Address of direct-page/stack buffer
`buffSize`	Word	Size of direct-page/stack buffer

Input type

If `inputType` = 0, this function is exactly equivalent to the InitialLoad call.

If `inputType` = 1, the input load-file pathname is a GS/OS string rather than a Pascal string.

If `inputType` = 2, the input address points to a parameter block rather than a pathname. The parameter block contains two parameters: `memoryAddress` (4 bytes) and `fileLength` (2 bytes). The `memoryAddress` parameter specifies where a load file resides in memory and the `fileLength` parameter specifies its size in bytes. The System Loader loads the file from memory rather than from a file, in this case.

This input type is used by GS/OS at system startup to load load files that were previously read into memory as binary images. In this mode, the System Loader does not make any GS/OS calls and can therefore be used when GS/OS is not in memory or has not yet been initialized.

If `inputType` = 3, the input address points to an entry in the pathname table. The pathname, user ID, and file number from the pathname table entry are used as input for InitialLoad. This entry is used by the jump table Load function (an internal function) to load all the static segments in a run-time library.

Errors

$1102	OMF version error
$1104	File is not load file
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign

$11 LGetPathname

Description

This function searches the pathname table for the specified user ID and file number. If a match is found, the address of the pathname in the pathname table is returned. The output pathname is a Pascal string.

GS/OS uses this call to get the pathname of an existing application so that it can set the Application prefix before restarting it. Note that the output address is within a System Loader internal data structure, and nothing should be written to that address or the following addresses.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID to find
`fileNum`	Word	The file number to find
Output:		
`pathnameAddr`	Longword	Address of pathname (if found)

Errors

$1101	Entry not found
$1103	Pathname error

$22 LGetPathname2

Description

This function is identical to LGetPathname except that the output pathname is a GS/OS string rather than a Pascal string.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID to find
`fileNum`	Word	The file number to find
Output:		
`pathnameAddr`	Longword	Address of pathname (if found)

Errors

$1101	Entry not found
$1103	Pathname error

$01 LoaderInitialization

Description This routine initializes the System Loader. It is called at system initialization time only. All System Loader tables are cleared, and no assumptions are made about the current or previous state of the system.

Parameters

Name	Size	Description
Input:	None	
Output:	None	

Errors None

$05 LoaderReset

Description This routine does nothing and need not be called.

Parameters

Name	Size	Description
Input:	None	
Output:	None	

Errors None

$03 LoaderShutDown

Description This routine does nothing and need not be called.

Parameters

Name	Size	Description
Input:	None	
Output:	None	

Errors None

$02 LoaderStartup

Description This routine does nothing and need not be called.

Parameters

Name	Size	Description
Input:	None	
Output:	None	

Errors None

$06 LoaderStatus

Description

This routine returns the status (initialized or not initialized) of the System Loader. It always returns TRUE because the System Loader is always in the initialized state.

Parameters

Name	Size	Description
Input:	None	
Output:		
`status`	Word	Current System Loader status; always TRUE (= initialized)

Errors

None

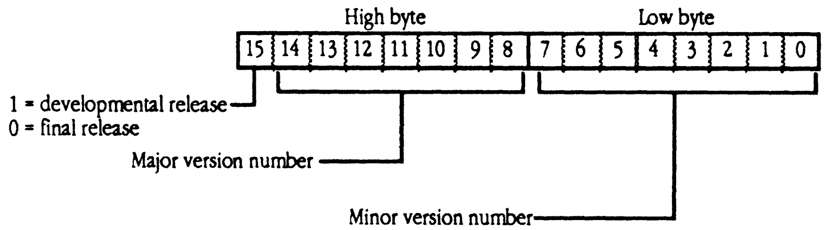
$04 LoaderVersion

Description This routine returns the version number of the System Loader. The version number is in the same format as that returned by the GS/OS call GetVersion:

Parameters

Name	Size	Description
Input:	None	
Output:		
`version`	Word	Present System Loader version

Version word This is the format of the version word returned by this call:



Errors None

$0D LoadSegName (Load Segment by Name)

Description

This function loads a named load segment into memory.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID of the caller
`filenameAddr`	Longword	The address of the load-file name
`segNameAddr`	Longword	The address of the load-segment name
Output:		
`segAddr`	Longword	The starting address of the segment
`userID`	Word	The user ID assigned
`fileNum`	Word	The load-file number of the segment
`segNum`	Word	The load-segment number of the segment

Notes

The input pathname is a Pascal string. The loader calls GS/OS to open the specified load file. If GS/OS has a problem, a GS/OS error code is returned. If the file is not a load file (types $B3–$BE), error $1104 is returned.

Next the load file is searched for a load segment corresponding to the specified load-segment name. If no segment has the segment–name requested, error $1101 is returned.

Once the System Loader has located the requested load segment (and knows its load-segment number), it checks the pathname table to see whether the load file is represented. If so, it uses the file number from the table. Otherwise, the System Loader adds a new entry to the pathname table with an unused file number. If necessary, the System Loader loads the jump-table segment (if any) from the load file.

Next the System Loader attempts to load the load segment by calling the Load Segment by Number function (LoadSegNum). If LoadSegNum returns an error, then LoadSegName returns the error. If LoadSegNum is successful, LoadSegName returns the load-file number, the load-segment number, and the starting address of the segment in memory.

Errors	\$1101	Segment not found
	\$1104	File is not load file
	\$1107	File version error
	\$1109	SegNum out of sequence
	\$110A	Illegal load record found
	\$110B	Load segment is foreign

$0D LoadSegName (Load Segment by Name)

Description This function loads a named load segment into memory.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID of the caller
`filenameAddr`	Longword	The address of the load-file name
`segNameAddr`	Longword	The address of the load-segment name
Output:		
`segAddr`	Longword	The starting address of the segment
`userID`	Word	The user ID assigned
`fileNum`	Word	The load-file number of the segment
`segNum`	Word	The load-segment number of the segment

Notes The input pathname is a Pascal string. The loader calls GS/OS to open the specified load file. If GS/OS has a problem, a GS/OS error code is returned. If the file is not a load file (types $B3–$BE), error $1104 is returned.

Next the load file is searched for a load segment corresponding to the specified load-segment name. If no segment has the segment–name requested, error $1101 is returned.

Once the System Loader has located the requested load segment (and knows its load-segment number), it checks the pathname table to see whether the load file is represented. If so, it uses the file number from the table. Otherwise, the System Loader adds a new entry to the pathname table with an unused file number. If necessary, the System Loader loads the jump-table segment (if any) from the load file.

Next the System Loader attempts to load the load segment by calling the Load Segment by Number function (LoadSegNum). If LoadSegNum returns an error, then LoadSegName returns the error. If LoadSegNum is successful, LoadSegName returns the load-file number, the load-segment number, and the starting address of the segment in memory.

Errors	\$1101	Segment not found
	\$1104	File is not load file
	\$1107	File version error
	\$1109	SegNum out of sequence
	\$110A	Illegal load record found
	\$110B	Load segment is foreign

$0B LoadSegNum (Load Segment by Number)

Description

This function loads a specific load segment into memory. This is the workhorse function of the System Loader. Normally, a program calls this function to manually load a dynamic load segment. If a program calls this function to load a static load segment, the System Loader does not patch any existing references to the newly loaded segment.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID to be assigned
`fileNum`	Word	The load-file number of the segment
`segNum`	Word	The load-segment number of the segment
Output:		
`segAddr`	Longword	The starting address of the segment

Sequence

First the memory-segment table is searched to see if there is an entry for the requested load segment. If there is already an entry, the handle to the memory block is checked to verify it is still in memory. If the block is still in memory, this function does nothing further and returns without an error. If the memory block has been purged, the memory-segment table entry is deleted.

Next the load-file number is looked up in the pathname table to get the load file pathname. From the file's directory entry, the load-file type is checked; if it is not a load file (types $B3–$BE), error $1104 is returned. The load file's modification date/time values are compared to the file date and file time values in the pathname table. If these values do not match, error $1107 is returned. This indicates that the run-time library file at the specified pathname is not the run-time library file that was scanned when the application was linked together.

The System Loader then calls GS/OS to open the specified load file. If GS/OS has a problem, a GS/OS error code is returned.

Next the load file is searched for a load segment corresponding to the specified load-segment number. If there is no segment corresponding to the load-segment number, error $1101 is returned. If the VERSION field of the segment header contains a value that is not supported by the System Loader, error $1102 is returned. If the SEGNUM field does not correspond to the load-segment number, error $1109 is returned. If the NUMSEX and NUMLEN fields are not 0 and 4, respectively, error $110B is returned.

If the load segment is found and its segment header is correct, a memory block is requested from the Memory Manager of size specified in the LENGTH field in the segment header. If the ORG field in the segment header is not 0, a memory block starting at that address is requested. Other attributes are set according to segment header fields (see "The System Loader and the Memory Manager," earlier in this chapter).

If the input user ID is not 0, it is used as the user ID of the memory block. If the input user ID is 0, the memory block is marked as belonging to the user ID of the current user (in USERID).

If the requested memory is not available, the Memory Manager and the System Loader will try several techniques to free up memory:

- the Memory Manager purges memory blocks that are marked purgeable;
- the Memory Manager moves movable segments to enlarge contiguous memory;
- and the System Loader calls its Cleanup routine (an internal function) to free its own unused internal memory.

If all these techniques fail, the System Loader returns with the last Memory Manager error.

If enough memory is available, the System Loader loads the load segment into memory and processes its **relocation dictionary,** the part of every relocatable segment that the loader uses to patch the code for correct execution at its current address. See Appendix B.

The loader adds a new entry to the memory-segment table and returns with the memory handle of the segment's memory block.

OMF records

Only the following object module format records are supported by the System Loader:

LCONST	($F2)
DS	($F1)
RELOC	($E2)
INTERSEG	($E3)
cRELOC	($F5)
cINTERSEG	($F6)
SUPER	($F7)
END	($00)

Any other records encountered while loading result in error $110A.

Errors

$1101	Segment not found
$1102	OMF version error
$1104	File is not load file
$1107	File version error
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Segment is foreign

$0A Restart

Description

A controlling program (such as GS/OS, Basic, Switcher, etc.) uses this call to restart (relaunch) a dormant application in memory. Only software that is **restartable** can be successfully restarted. For a program to be restartable, it must initialize its variables and not assume that they will be preset at load time. A reload segment can be used for initializing data because it is reloaded from the file during a restart. The controlling program is responsible for knowing whether a given program can be restarted; the System Loader does no checking.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID of the program to restart
Output:		
`userID`	Word	The user ID of the restarted program
`startAddr`	Longword	The starting address of the program
`buffAddr`	Word	Address of direct-page/stack buffer
`buffSize`	Word	Size of direct-page/stack buffer

Notes

An existing user ID must be specified; otherwise, the System Loader returns error $1108. If the user ID is not known to the System Loader, error $1101 is returned.

Applications can be restarted only if all the segments in the memory-segment table with the input user ID are in memory; these are the application's static segments. If all are there, the System Loader resurrects the application from its dormant state by calling the Memory Manager to lock and make unpurgeable all its segments.

The Restart call returns the user ID and the starting address of the first segment, as well as the direct-page/stack information from the pathname table. After all the static segments are resurrected, the System Loader looks for initialization segments and reload segments; it executes the former and reloads the latter.

If there is a pathname table entry for the user ID but not all the segments are in memory, the System Loader first calls UserShutdown, which purges the user ID from all its tables, and then performs an InitialLoad from the original load file.

Errors	$1101	Application not found
	$1108	User ID error

$0E UnloadSeg (Unload Segment by Address)

Description This function unloads the load segment that contains the specified input address.

Parameters

Name	Size	Description
Input		
`address`	Longword	An address within the segment to be unloaded
Output		
`userID`	Word	The user ID of the segment
`fileNum`	Word	The load-file number of the segment
`segNum`	Word	The load-segment number of the segment

Notes The System Loader calls the Memory Manager to locate the memory block containing the specified address. If no allocated memory block contains the address, error $1101 is returned. The user ID associated with the handle of the memory block returned by the Memory Manager is extracted, and the memory-segment table is scanned to find the user ID and handle. If an entry is not found, error $1101 is returned.

If the entry in the memory-segment table is for a jump-table segment, the specified address should be pointing to the jump–table entry for a dynamic segment reference. The load-file number and segment number of the jump–table entry are extracted.

If the entry in the memory–segment–table is not for a jump–table segment, the load-file number and segment number of the memory-segment table entry are extracted.

UnloadSeg now calls UnloadSegNum to actually unload the segment. The results returned by UnloadSeg can be used as input to other System Loader functions, such asUnloadSedNum.

Errors $1101 Segment not found

$0C UnloadSegNum (Unload Segment by Number)

Description This function unloads a specified (by number) load segment that is currently in memory.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID of the segment to be unloaded
`fileNum`	Word	The load-file number of the segment
`segNum`	Word	The load-segment number of the segment

Output:

None

Notes The System Loader searches the memory-segment table for the input load-file number and load-segment number. If there is no such entry, error $1101 is returned.

Next the Memory Manager is called to make the memory block purgeable, using the memory handle in the table entry.

All entries in the jump table referencing the unloaded segment are changed to their unloaded states.

If the input user ID is 0, the user ID of the current user (in USERID) is assumed.

If both the load-file number and the load-segment number are specified, the specific load segment is made purgeable whether it is static or dynamic. Note that if a static segment is unloaded, the application can not be restarted. If either input is 0, only dynamic segments are made purgeable.

If the input load-segment number is 0, all dynamic segments in the specified load file are unloaded.

If the input load-file number is 0, all dynamic segments for the user ID are unloaded.

Errors $1101 Segment not found

$12 UserShutDown

Description

This function is called by the controlling program to close down an application that has just terminated. If the specified user ID is 0, the current user ID (USERID) is assumed.

Parameters

Name	Size	Description
Input:		
`userID`	Word	The user ID of the program to shut down
`flag`	Word	The quit flag
Output:		
`userID`	Word	The user ID of the program **that** was shut down

Notes

The quit flag corresponds to the quit flag used in the GS/OS Quit call:

- If the quit flag is 0, all memory blocks for the user ID are discarded and all the System Loader's internal tables are purged of the user ID. The application cannot be restarted. The user ID is also removed from the system so that it can be reused.
- If the quit flag is $8000, all memory blocks for the user ID are discarded and all the System Loader's tables (except the pathname table) are purged of the user ID. The application can be reloaded (but not restarted), because its pathname is remembered.
- If the quit flag is any other value, the memory blocks associated with the specified user ID (with `auxID` cleared) are processed as follows:
  - all memory blocks corresponding to dynamic load segments are discarded
  - all memory blocks corresponding to static load segments are made purgeable
  - all other memory blocks are purged

  In addition, all dynamic segment entries in the memory-segment table and all entries in the jump-table directory for the specified user ID are removed. The application is now in a **dormant** state and can be restarted (resurrected by the System Loader very quickly because all the static segments are still in memory). However, as soon as any one static segment is purged by the Memory Manager for whatever reason, the System Loader must reload the application from its original load file.

Errors

None

Appendix B **Object Module Format**

Object module format (OMF) is the general file format followed by all object files, library files, and executable load files that run on the Apple IIGS computer under ProDOS 16 or GS/OS. It is a general format that allows dynamic loading and unloading of file segments, both at startup and while a program is running.

Most application writers need not be concerned with the details of OMF. If, however, you are writing a compiler or other program that must create or modify executable files, or if you want to understand the details of how the System Loader functions, you need to understand OMF.

△ **Important** This appendix describes Version 2.1 of the Apple IIGS object module format (OMF). △

What files are OMF files?

The Apple IIGS object module format (OMF) supports language, linker, library, and loader requirements, and it is extremely flexible, easy to generate, and fast to load.

Under ProDOS 8 on the Apple IIe and Apple IIc, there is only one loadable file format, called the binary file format. This format consists of one absolute memory image along with its destination address. ProDOS 8 does not have a relocating loader, so that even if you write relocatable code, you must specify the memory location at which the file is to be loaded.

The Apple IIGS uses a more general format that allows dynamic loading and unloading of file segments while a program is running and that supports the various needs of many languages and assemblers. Apple IIGS linkers (supplied with development environments) and the System Loader fully support relocatable code; in general, you do not specify a load address for an Apple IIGS program, but let the loader and Memory Manager determine where to load the program.

Four kinds of files use object module format: object files, library files, load files, and run-time library files.

- **Object files** are the output from an assembler or compiler and the input to a linker. Object files must be fast to process, easy to create, independent of the source language, and able to support libraries in a convenient way. In some development environments object files also support segmentation of code. They support both absolute and relocatable program segments.

  Apple IIgs object files contain both machine-language code and relocation information for use by the linker. Object files cannot be loaded directly into memory; they must first be processed by the linker to create load files.

- **Library files** contain general object segments that a linker can find and extract to resolve references unresolved in the object files. Only the code needed during the link process is extracted from the library file.

- **Load files,** which are the output of a linker, contain memory images that a loader loads into memory. Load files must be very fast to process. Apple IIGS load files contain load segments that can be relocatable, movable, dynamically loadable, or have any combination of these attributes. Shell applications are load files that can be run from a shell program without requiring the shell to shut down. Startup load files are load files that GS/OS loads during its startup.

  Load files are created by the linker from object files and library files. Load files can be loaded into memory by the System Loader; they cannot be used as input to the linker.

- **Run-time library files** are load files containing general routines that can be shared between applications. The routines are contained in file segments that can be loaded as needed by the System Loader and then purged from memory when they are no longer needed. The run-time library files are also input to the linker which scans them for unresolved references. However, segments that satisfy references are not included in the link.

All four types of files consist of individual components called segments. Each file type uses a subset of the full object module format. Each compiler or assembler uses a subset of the format depending on the requirements and complexity of the language.

Some common GS/OS file types related to program files are listed in Table B-1.

- **Table B-1** GS/OS program-file types

Hex.	Dec.	Mnemonic	Meaning
$B0	176	SRC	Source
$B1	177	OBJ	Object
$B2	178	LIB	Library
$B3	179	S16	GS/OS or ProDOS 16 application
$B4	180	RTL	Run-time library
$B5	181	EXE	Shell application
$B6	182	PIF	Permanent initialization
$B7	183	TIF	Temporary initialization
$B8	184	NDA	New desk accessory
$B9	185	CDA	Classic desk accessory
$BA	186	TOL	Tool set file

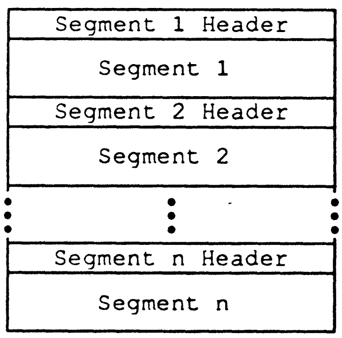
The rest of this appendix defines object module format. First, the general format specification for all OMF files is described. Then, the unique characteristics of each of the following file types are discussed:

- object files
- library files
- load files
- run-time library files
- shell applications

General format for OMF files

Each OMF file contains one or more segments. Each segment consists of a segment header and a segment body. The segment header contains general information about the segment, such as its name and length. The segment body is a sequence of records; each record consists of either program code or information used by a linker or by the System Loader. Figure B-1 represents the structure of an OMF file.

- **Figure B-1** The structure of an OMF file



Each segment in an OMF file contains a set of records that provide relocation information or contain code or data. If the file is an object file, each segment includes the information the linker needs to generate a relocatable load segment; the linker processes each record and generates a load file containing load segments. If the file is a load file, each segment consists of a memory image followed by a relocation dictionary; the System Loader loads the memory image and then processes the information in the relocation dictionary. (Load file segments on the Apple IIGS are usually relocatable.) Relocation dictionaries are discussed in the section "Load Files," later in this appendix.

Segments in object files can be combined by the linker into one or more segments in the load file. (See the discussion of the LOADNAME field in the section "Segment Header," later in this appendix.) For instance, each subroutine in a program can be compiled independently into a separate (object) code segment; then the linker can be told to place all the code segments into one load segment.

Segment types and attributes

Each OMF segment has a segment type and can have several attributes. The following segment types are defined by OMF:

- code segment
- data segment
- jump-table segment
- pathname segment
- library dictionary segment
- initialization segment
- direct-page/stack segment

The following segment attributes are defined by the object module format:

- reloadable or not reloadable
- absolute-bank or not restricted to a particular bank
- loadable in special memory or not loadable in special memory
- position-independent or position-dependent
- private or public
- static or dynamic
- bank-relative or not bank-relative

- skipped or not

Code and data segments are object segments provided to support languages (such as assembly language) that distinguish program code from data. If a programmer specifies a segment by using a PROC assembler directive, the linker flags it as a code segment; if the programmer uses a RECORD directive instead, the linker flags it as a data segment.

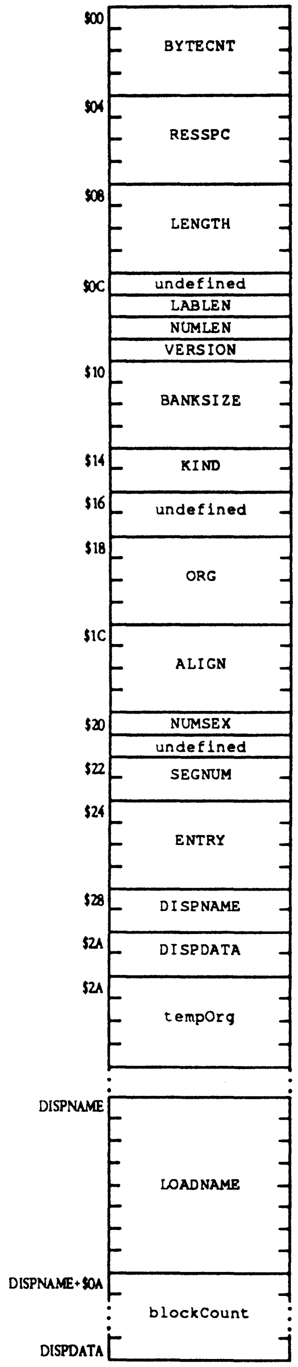
- **Jump-table segments** and **pathname segments** are load segments that facilitate the dynamic loading of segments; they are described in the section "Load Files," later in this appendix.
- **Library dictionary segments** allow the linker to scan library files quickly for needed segments; they are described in the section "Library Files," later in this appendix.
- **Initialization segments** are optional parts of load files that are used to perform any initialization required by the application during an initial load. If used, they are loaded and executed immediately as the System Loader encounters them and are re-executed any time the program is restarted from memory. Initialization segments are described in the section "Load Files," later in this appendix.
- **Direct-page/stack segments** are load segments used to preset the location and contents of the direct page and stack for an application. See the section "Direct-Page/Stack Segments," later in this appendix for more information.
- **Reload segments** are load segments that the loader must reload even if the program is restartable and is restarted from memory. They usually contain data that must be restored to its initial values before a program can be restarted.
- **Absolute-bank segments** are load segments that are restricted to a specified bank but that can be relocated within that bank. The ORG field in the segment header specifies the bank to which the segment is restricted.
- **Loadable in special memory** means that a segment can be loaded in banks $00, $01, $E0, and $E1. Because these are the banks used by programs running under ProDOS 8 in standard-Apple II emulation mode, you may wish to prevent your program from being loaded in these banks so that it can remain in memory while programs are run under ProDOS 8.
- **Position-independent segments** can be moved by the Memory Manager during program execution if they have been unlocked by the program.
- A **private code segment** is a code segment whose name is available only to other code segments within the same object file. (The labels within a code segment are local to that segment.)

- A **private data segment** is a data segment whose labels are available only to code segments in the same object file.
- **Static segments** are load segments that are loaded at program execution time and are not unloaded during execution; **dynamic segments** are loaded and unloaded during program execution as needed.
- **Bank-relative segments** must be loaded at a specified address within any bank. The ORG field in the segment header specifies the bank-relative address (the address must be less than $10000).
- **Skip segments** will not be linked by the Linker or loaded by the System Loader. However, all references to global definitions in a Skip object segment will be processed by a Linker as if the object segment
- A segment can have only one segment *type* but can have any combination of *attributes*. The segment types and attributes are specified in the segment header by the KIND segment-header field, described in the next section.

Segment header

Each segment in an OMF file has a header that contains general information about the segment, such as its name and length. Segment headers make it easy for the linker to scan an object file for the desired segments, and they allow the System Loader to load individual load segments. The format of the segment header is illustrated in Figure B-2. A detailed description of each of the fields in the segment header follows the figure.

■ **Figure B-2** The format of a segment header



△ **Important** In future versions of the OMF, additional fields may be added to the segment header between the DISPDATA and LOADNAME fields. To ensure that future expansion of the segment header does not affect your program, always use DISPNAME and DISPDATA instead of absolute offsets when referencing LOADNAME, SEGNAME, and the start of the segment body, and always be sure that all undefined fields are set to 0. △

BYTECNT A 4-byte field indicating the number of bytes in the file that the segment requires. This number includes the segment header, so you can calculate the starting Mark of the next segment from the starting Mark of this segment plus BYTECNT. Segments need not be aligned to block boundaries.

RESSPC A 4-byte field specifying the number of bytes of zeros to add to the end of the segment. This field can be used in an object segment instead of a large block of zeros at the end of the segment. This field duplicates the effect of a DS record at the end of the segment.

LENGTH A 4-byte field specifying the memory size that the segment will require when loaded. It includes the extra memory specified by RESSPC.

LENGTH is followed by one undefined byte, reserved for future changes to the segment header specification.

LABLEN A 1-byte field indicating the length, in bytes, of each name or label record in the segment body. If LABLEN is 0, the length of each name or label is specified in the first byte of the record (that is, the first byte of the record specifies how many bytes follow). LABLEN also specifies the length of the SEGNAME field of the segment header, or, if LABLEN is 0, the first byte of SEGNAME specifies how many bytes follow. (The LOADNAME field always has a length of 10 bytes.) Fixed-length labels are always left justified and padded with spaces.

NUMLEN A 1-byte field indicating the length, in bytes, of each number field in the segment body. This field is 4 for the Apple IIGS.

VERSION A 1-byte field indicating the version number of the object module format with which the segment is compatible. At the time of publication, this field is set to 2 for the current object module format.

REVISION A 1-byte field indicating the revision number of the object module format with which the segment is compatible. Together with the VERSION field, REVISION specifies the OMF compatibility level of this segment. At the time of publication, this field is set to 1 for the current object module format.

BANKSIZE A 4-byte binary number indicating the maximum memory-bank size for the segment. If the segment is in an object file, the linker ensures that the segment is not larger than this value. (The linker returns an error if the segment is too large.) If the segment is in a load file, the loader ensures that the segment is loaded into a memory block that does not cross this boundary. For Apple IIGS code segments, this field must be $00010000, indicating a 64K bank size. A value of 0 in this field indicates that the segment can cross bank boundaries. Apple IIGS data segments can use any number from $00 to $00010000 for BANKSIZE.

KIND A 2-byte field specifying the type and attributes of the segment. The bits are defined as shown in Table B-2. The column labeled Where Described indicates the section in this appendix where the particular segment type or attribute is discussed.

■ **Table B-2** KIND field definition

Bit(s)	Values	Meaning	Where described
0–4		*Segment Type subfield*	
	\$00	code	Segment Types and Attributes
	\$01	data	Segment Types and Attributes
	\$02	Jump-table segment	Load Files
	\$04	Pathname segment	Segment Types and Attributes
	\$08	Library dictionary segment	Library Files
	\$10	Initialization segment	Load Files
	\$12	Direct-page/stack segment	Direct-Page/Stack Segments
10–15		*Segment Attributes bits*	
8	if = 1	Bank-relative segment	Segment Types and Attributes
9	if = 1	Skip segment	Segment Types and Attributes
10	if = 1	Reload segment	Segment Types and Attributes
11	if = 1	Absolute-bank segment	Segment Types and Attributes
12	if = 0	Can be loaded in special memory	Segment Types and Attributes
13	if = 1	Position independent	Segment Types and Attributes
14	if = 1	Private	Segment Types and Attributes
15	if = 0	Static; otherwise dynamic	Segment Types and Attributes

A segment can have only one *type* but any combination of *attributes*. For example, a position-independent dynamic data segment has KIND = (\$A001).

△ **Important** If segment KINDs are specified in the source file, and the KINDs of the object segments placed in a given load segment are not all the same, the segment KIND of the first object segment determines the segment kind of the entire load segment. △

KIND is followed by two undefined bytes, reserved for future changes to the segment header specification.

ORG A 4-byte field indicating the absolute address at which this segment is to be loaded in memory, or, for an absolute-bank segment, the bank number. A value of 0 indicates that this segment is relocatable and can be loaded anywhere in memory. A value of 0 is normal for the Apple IIGS.

ALIGN A 4-byte binary number indicating the boundary on which this segment must be aligned. For example, if the segment is to be aligned on a page boundary, this field is $00000100; if the segment is to be aligned on a bank boundary, this field is $00010000. A value of 0 indicates that no alignment is needed. For the Apple IIGS, this field must be a power of 2, less than or equal to $00010000. Currently, the loader supports only values of 0, $00000100, and $00010000; for any other value, the loader uses the next higher supported value.

NUMSEX A 1-byte field indicating the order of the bytes in a number field. If this field is 0, the least significant byte is first. If this field is 1, the most significant byte is first. This field is set to 0 for the Apple IIGS.

NUMSEX is followed by one undefined byte, reserved for future changes to the segment header specification.

SEGNUM A 2-byte field specifying the segment number. The segment number corresponds to the relative position of the segment in the file (starting with 1). This field is used by the System Loader to search for a specific segment in a load file.

ENTRY A 4-byte field indicating the offset into the segment that corresponds to the entry point of the segment.

DISPNAME A 2-byte field indicating the displacement of the LOADNAME field within the segment header. Currently, DISPNAME = 44. DISPNAME is provided to allow for future additions to the segment header; any new fields will be added between DISPDATA and LOADNAME. DISPNAME allows you to reference LOADNAME and SEGNAME no matter what the actual size of the header.

DISPDATA A 2-byte field indicating the displacement from the start of the segment header to the start of the segment body. DISPDATA is provided to allow for future additions to the segment header; any new fields will be added between DISPDATA and LOADNAME. DISPDATA allows you to reference the start of the segment body no matter what the actual size of the header.

tempORG A 4-byte field indicating the temprorary origin of the Object segment. A nonzero value indicates that all references to globals within this segment will be interpreted as if the Object segment started at that location. However, the actual load address of the Object segment is still determined by the ORG field.

LOADNAME A 10-byte field specifying the name of the load segment that will contain the code generated by the linker for this segment. More than one segment in an object file can be merged by the linker into a single segment in the load file. This field is unused in a load segment. The position of LOADNAME may change in future revisions of the OMF; therefore, you should always use DISPNAME to reference LOADNAME.

SEGNAME A field that is LABLEN bytes long, and that specifies the name of the segment. The position of SEGNAME may change in future revisions of the OMF; therefore, you should always use DISPNAME to reference SEGNAME.

Segment body

The body of each segment is composed of sequential records, each of which starts with a 1-byte operation code. Each record contains either program code or information for the linker or System Loader. All names and labels included in these records are LABLEN bytes long, and all numbers and addresses are NUMLEN bytes long (unless otherwise specified in the following definitions). For the Apple IIGS, the least significant byte of each number field is first, as specified by NUMSEX.

Several of the OMF records contain expressions that have to be evaluated by the linker. The operation and syntax of expressions are described in the next section, "Expressions." If the description of the record type does not explicitly state that the opcode is followed by an expression, then an expression cannot be used. Expressions are never used in load segments.

The operation codes and segment records are described in this section, listed in order of the opcodes. Table B-3 provides an alphabetical cross-reference between segment record types and opcodes. Library files consist of object segments and so can use any record type that can be used in an object segment. Table B-3 also lists the segment types in which each record type can be used.

■ **Table B-3** Segment-body record types

Record type	Opcode	Found in what segment types
ALIGN	$E0	object
BEXPR	$ED	object
cINTERSEG	$F6	load
CONST	$01–$DF	object
cRELOC	$F5	load
DS	$F1	all
END	$00	all
ENTRY	$F4	run-time library dictionary
EQU	$F0	object
EXPR	$EB	object
GEQU	$E7	object
GLOBAL	$E6	object
INTERSEG	$E3	load
LCONST	$F2	all
LEXPR	$F3	object
LOCAL	$EF	object
MEM	$E8	object
ORG	$E1	object
RELEXPR	$EE	object
RELOC	$E2	load
STRONG	$E5	object
SUPER	$F7	load
USING	$E4	object
ZEXPR	$EC	object

The rest of this section defines each of the segment-body record types. The record types are listed in order of their opcodes.

Record type	**Opcode**	**Explanation**
END	$00	This record indicates the end of the segment.
CONST	$01–$DF	This record contains absolute data that needs no relocation. The operation code specifies how many bytes of data follow.
ALIGN	$E0	This record contains a number that indicates an alignment factor. The linker inserts as many zero bytes as necessary to move to the memory boundary indicated by this factor. The value of this factor is in the same format as the ALIGN field in the segment header and cannot have a value greater than that in the ALIGN field. ALIGN must equal a power of 2.
ORG	$E1	This record contains a number that is used to increment or decrement the location counter. If the location counter is incremented (ORG is positive), zeros are inserted to get to the new address. If the location counter is decremented (ORG is a complement negative number of 2), subsequent code overwrites the old code.
RELOC	$E2	This is a relocation record, which is used in the relocation dictionary of a load segment. It is used to patch an address in a load segment with a reference to another address in the same load segment. It contains two 1-byte counts followed by two offsets. The first count is the number of bytes to be relocated. The second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, the number is shifted to the left, filling vacated bit positions with zeros (arithmetic shift left). If the bit-shift operator is (two's complement) negative, the number is shifted right (logical shift right) and zero-filled.

The first offset gives the location (relative to the start of the segment) of the first byte of the number that is to be patched (relocated). The second offset is the location of the reference relative to the start of the segment; that is, it is the value that the number would have if the segment containing it started at address $000000. For example, suppose the segment includes the following lines:

```
35  LABEL • • •
 •
 •
 •
400 LDA  LABEL+4
```

The RELOC record contains a patch to the operand of the LDA instruction. The value of the patch is LABEL+4, so the value of the last field in the RELOC record is $39—the value the patch would have if the segment started at address $000000. LABEL+4 is two bytes long; that is, the number of bytes to be relocated is 2. No bit-shift operation is needed. The location of the patch is 1025 ($401) bytes after the start of the segment (immediately after the LDA, which is one byte).

The RELOC record for the number to be loaded into the A register by this statement would therefore look like this (note that the values are stored low-byte first, as specified by NUMSEX):

E2020001 04000039 000000

This sequence corresponds to the following values:

$E2	operation code
$02	number of bytes to be relocated
$00	bit-shift operator
$00000401	offset of value from start of segment
$00000039	value if segment started at $000000

- *Illegal expressions:* Certain types of arithmetic expressions are illegal in a relocatable segment; specifically, any expression that the assembler cannot evaluate (relative to the start of the segment) a cannot be used. The expression LAB|4 can be evaluated, for example, since the RELOC record includes a bit-shift operator. The expression LAB|4+4 cannot be used, however, because the assembler would have to know the absolute value of LAB to perform the bit-shift operation before adding 4 to it. Similarly, the value of LAB*4 depends on the absolute value of LAB and cannot be evaluated relative to the start of the segment; so multiplication is illegal in expressions in relocatable segments.

INTERSEG $E3 This record is used in the relocation dictionary of a load segment. It contains a patch to a long call to an external reference; that is, the INTERSEG record is used to patch an address in a load segment with a reference to another address in a different load segment. It contains two 1-byte counts followed by an offset, a 2-byte file number, a 2-byte segment number, and a second offset. The first count is the number of bytes to be relocated, and the second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, the number is shifted to the left, filling vacated bit positions with zeros (arithmetic shift left). If the bit-shift operator is (two's complement) negative, the number is shifted right (logical shift right) and zero-filled.

The first offset is the location (relative to the start of the segment) of the (first byte of the) number that is to be relocated. If the reference is to a static segment, the file number, segment number, and second offset correspond to the subroutine referenced. (The linker assigns a file number to each load file in a program. This feature is provided primarily to support run-time libraries. In the normal case of a program having one load file, the file number is 1. The load segments in a load file are numbered by their relative locations in the load file, where the first load segment is number 1.) If the reference is to a dynamic segment, the file and segment numbers correspond to the jump-table segment, and the second offset corresponds to the call to the System Loader for that reference.
For example, suppose the segment includes an instruction such as

```
JSL EXT
```

The label EXT is an external reference to a location in a static segment.

If this instruction is at relative address $720 within its segment and EXT is at relative address $345 in segment $000A in file $0001, the linker creates an INTERSEG record in the relocation dictionary that looks like this (note that the values are stored low-byte first, as specified by NUMSEX):

E3030021 07000001 000A0045 030000

This sequence corresponds to the following values:

$E3	operation code
$03	number of bytes to be relocated
$00	bit-shift operator
$00000721	offset of instruction's operand
$0001	file number
$000A	segment number
$00000345	offset of subroutine referenced

When the loader processes the relocation dictionary, it uses the first offset to find the JSL and patches in the address corresponding to the file number, segment number, and offset of the referenced subroutine.

If the JSL is to an external reference in a dynamic segment, the INTERSEG records refer to the file number, segment number, and offset of the call to the System Loader in the jump-table segment.

If the jump-table segment is in segment 6 of file 1, and the call to the System Loader is at relative location $2A45 in the jump-table segment, then the INTERSEG record looks like this (note that the values are stored low-byte first, as specified by NUMSEX):

E3030021 07000001 00060045 2A0000

This sequence corresponds to the following values:

$E3	operation code
$03	number of bytes to be relocated
$00	bit-shift operator
$00000721	offset of instruction's operand
$0001	file number of jump-table segment
$0006	segment number of jump-table segment
$00002A45	offset of call to System Loader

The jump-table segment entry that corresponds to the external reference EXT contains the following values:

User ID	
$0001	file number
$0005	segment number
$00000200	offset of instruction call to System Loader

INTERSEG records are used for any long-address reference to a static segment.

See the section "Jump-Table Segment," later in this appendix, for a discussion of the function of the jump-table segment.

USING $E4 This record contains the name of a data segment. After this record is encountered, local labels from that data segment can be used in the current segment.

STRONG $E5 This record contains the name of a segment that must be included during linking, even if no external references have been made to it. If you are using the APW assembler, the following statement generates a STRONG record:

```
DC      R'xxxx'
```

where *xxxx* is label.

GLOBAL	$E6	This record contains the name of a global label followed by three attribute fields. The label is assigned the current value of the location counter. The first attribute field is two bytes long and gives the number of bytes generated by the line that defined the label. If this field is $FFFF, it indicates that the actual length is unknown but that it is greater than or equal to $FFFF. The second attribute field is one byte long and specifies the type of operation in the line that defined the label. The following type attributes are defined (uppercase ASCII characters with the high-bit off):

A	address-type DC statement
B	Boolean-type DC statement
C	character-type DC statement
D	double-precision floating-point-type DC statement
F	floating-point-type DC statement
G	EQU or GEQU statement
H	hexadecimal-type DC statement
I	integer-type DC statement
K	reference-address-type DC statement
L	soft-reference-type DC statement
M	instruction
N	assembler directive
O	ORG statement
P	ALIGN statement
S	DS statement
X	arithmetic symbolic parameter
Y	Boolean symbolic parameter
Z	character symbolic parameter

The third attribute field is one byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private. (See the section "Segment Types and Attributes," earlier in this appendix, for a definition of private segments.)

GEQU	$E7	This record contains the name of a global label followed by three attribute fields and an expression. The label is given the value of the expression. The first attribute field is 2 bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is 1 byte long and specifies the type of operation in the line that defined the label, as listed in the discussion of the GLOBAL record. The third attribute field is 1 byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private. (See the section "Segment Types and Attributes," earlier in this appendix, for a definition of private segments.)
MEM	$E8	This record contains two numbers that represent the starting and ending addresses of a range of memory that must be reserved. If the size of the numbers is not specified, the length of the numbers is defined by the NUMLEN field in the segment header.
EXPR	$EB	This record contains a 1-byte count followed by an expression. The expression is evaluated, and its value is truncated to the number of bytes specified in the count. The order of the truncation is from most significant to least significant.
ZEXPR	$EC	This record contains a 1-byte count followed by an expression. ZEXPR is identical to EXPR, except that any bytes truncated must be all zeros. If the bytes are not zeros, the record is flagged as an error.
BEXPR	$ED	This record contains a 1-byte count followed by an expression. BEXPR is identical to EXPR, except that any bytes truncated must match the corresponding bytes of the location counter. If the bytes don't match, the record is flagged as an error. This record allows the linker to make sure that an expression evaluates to an address in the current memory bank.
RELEXPR	$EE	This record contains a 1-byte length followed by an offset and an expression. The offset is NUMLEN bytes long. RELEXPR is used to generate a relative branch value that involves an external location. The length indicates how many bytes to generate for the instruction, the offset indicates where the origin of the branch is relative to the current location counter, and the expression is evaluated to yield the destination of the branch. For example, a BNE LOC instruction, where LOC is external, generates this record. For the 6502 and 65816 microprocessors, the offset is 1.

LOCAL	$EF	This record contains the name of a local label followed by three attribute fields. The label is assigned the value of the current location counter. The first attribute field is two bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is one byte long and specifies the type of operation in the line that defined the label, as listed in the discussion of the GLOBAL record. The third attribute field is one byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private. (See the section "Segment Types and Attributes," earlier in this appendix, for a definition of private segments.) Some linkers (such as the APW Linker) ignore local labels from code segments and recognize local labels from other data segments only if a USING record was processed. See the preceding discussion of the USING statement.
EQU	$F0	This record contains the name of a local label followed by three attribute fields and an expression. The label is given the value of the expression. The first attribute field is two bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is one byte long and specifies the type of operation in the line that defined the label, as listed in the discussion of the GLOBAL record. The third attribute field is one byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private. (See the section "Segment Types and Attributes," earlier in this appendix, for a definition of private segments.)
DS	$F1	This record contains a number indicating how many bytes of zeros to insert at the current location counter.
LCONST	$F2	This record contains a 4-byte count followed by absolute code or data. The count indicates the number of bytes of data. The LCONST record is similar to CONST except that it allows for a much greater number of data bytes. Each relocatable load segment consists of LCONST records, DS records, and a relocation dictionary. See the discussions on INTERSEG records, RELOC records, and the relocation dictionary for more information.

LEXPR	$F3	This record contains a 1-byte count followed by an expression. The expression is evaluated, and its value is truncated to the number of bytes specified in the count. The order of the truncation is from most significant to least significant. Because the LEXPR record generates an intersegment reference, only simple expressions are allowed in the expression field, as follows: LABEL ± const LABEL | ± const (LABEL ± const) | ± const In addition, if the expression evaluates to a single label with a fixed, constant offset, and if the label is in another segment and that segment is a dynamic code segment, then the linker creates an entry for that label in the jump-table segment. (The jump-table segment provides a mechanism to allow dynamic loading of segments as they are needed—see the section "Load Files," later in this appendix.)
ENTRY	$F4	This record is used in the run-time library entry dictionary; it contains a 2-byte number and an offset followed by a label. The number is the segment number. The label is a code-segment name or entry, and the offset is the relative location within the load segment of the label. Run-time library entry dictionaries are described in the section "Run-Time Library Files," later in this appendix.
cRELOC	$F5	This record is the compressed version of the RELOC record. It is identical to the RELOC record, except that the offsets are two bytes long rather than four bytes. The cRELOC record can be used only if both offsets are less than $10000 (65536). The following example compares a RELOC record and a cRELOC record for the same reference:

RELOC	**cRELOC**
$E2	$F5
$02	$02
$00	$00
$00000401	$0401
$00000039	$0039
(11 bytes)	(7 bytes)

For an explanation of each line of these records, see the preceding discussion of the RELOC record.

cINTERSEG $F6 This record is the compressed version of the INTERSEG record. It is identical to the INTERSEG record, except that the offsets are two bytes long rather than four bytes, the segment number is one byte rather than two bytes, and this record does not include the 2-byte file number. The cINTERSEG record can be used only if both offsets are less than $10000 (65536), the segment number is less than 256, and the file number associated with the reference is 1 (that is, the initial load file). References to segments in run-time library files must use INTERSEG records rather than cINTERSEG records.

The following example compares an INTERSEG record and a cINTERSEG record for the same reference:

INTERSEG	**cINTERSEG**
$E3	$F6
$03	$03
$00	$00
$00000720	$0720
$0001	
$000A	$0A
$00000345	$0345
(15 bytes)	(8 bytes)

For an explanation of each line of these records, see the preceding discussion of the INTERSEG record.

SUPER $F7

This is a supercompressed relocation-dictionary record. Each SUPER record is the equivalent of many cRELOC, cINTERSEG, and INTERSEG records. It contains a 4-byte length, a 1-byte record type, and one or more subrecords of variable size, as follows:

opcode:	$F7
length:	number of bytes in the rest of the record (4 bytes)
type:	0–37 (1 byte)
subrecords:	(variable size)

When SUPER records are used, some of the relocation information is stored in the LCONST record at the address to be patched.

The length field indicates the number of bytes in the rest of the SUPER record (that is, the number of bytes exclusive of the opcode and the length field).

The type byte indicates the type of SUPER record. There are 38 types of SUPER record:

Value	SUPER record type
0	SUPER RELOC2
1	SUPER RELOC3
2–37	SUPER INTERSEG1–SUPER INTERSEG36

SUPER RELOC2: This record can be used instead of cRELOC records that have a bit-shift count of zero and that relocate two bytes.

SUPER RELOC3: This record can be used instead of cRELOC records that have a bit-shift count of zero and that relocate three bytes.

SUPER INTERSEG1: This record can be used instead of cINTERSEG records that have a bit-shift count of zero and that relocate three bytes.

SUPER INTERSEG2 through SUPER INTERSEG12: The number in the name of the record refers to the file number of the file in which the record is used. For example, to relocate an address in file 6, use a SUPER INTERSEG6 record. These records can be used instead of INTERSEG records that meet the following criteria:

- Both offsets are less than $10000
- The segment number is less than 256
- The bit-shift count is 0
- The record relocates 3 bytes
- The file number is from 2 through 12

SUPER INTERSEG13 through SUPER INTERSEG24: These records can be used instead of cINTERSEG records that have a bit-shift count of zero, that relocate two bytes, and that have a segment number of n – 12, where n is a number from 13 to 24. For example, to replace a cINTERSEG record in segment 6, use a SUPER INTERSEG18 record.

SUPER INTERSEG25 through SUPER INTERSEG36: These records can be used instead of cINTERSEG records that have a bit-shift count of $F0 (–16), that relocate two bytes, and that have a segment number of n – 24, where n is a number from 25 to 36. For example, to replace a cINTERSEG record in segment 6, use a SUPER INTERSEG30 record.

Each subrecord consists of either a 1-byte offset count followed by a list of 1-byte offsets, or a 1-byte skip count.

Each offset count indicates how many offsets are listed in this subrecord. The offsets are one byte each. Each offset corresponds to the low byte of the first (2-byte) offset in the equivalent INTERSEG, cRELOC, or cINTERSEG record. The high byte of the offset is indicated by the location of this offset count in the SUPER record: Each subsequent offset count indicates the next 256 bytes of the load segment. Each skip count indicates the number of 256-byte pages to skip; that is, a skip count indicates that there are no offsets within a certain number of 256-byte pages of the load segment.

For example, if patches must be made at offsets 0020, 0030, 0140, and 0550 in the load segment, the subrecords would include the following fields:

2 20 30	the first 256-byte page of the load segment has two patches: one at offset 20 and one at offset 30
1 40	the second 256-byte page has one patch at offset 40
skip-3	skip the next three 256-byte pages

1 50	the sixth 256-byte page has one patch at offset 50

In the actual SUPER record, the patch count byte is the number of offsets minus one, and the skip count byte has the high bit set. A SUPER INTERSEG1 record with the offsets in the preceding example would look like this:

$F7	opcode
$00000009	number of bytes in the rest of the record
$02	INTERSEG1-type SUPER record
$01	the first 256-byte page has two patches
$20	patch the load segment at offset $0020
$30	patch the segment at $0030
$00	the second page has one patch
$40	patch the segment at $0140
$83	skip the next three 256-byte pages
$00	the sixth page has one patch
$50	patch the segment at $0550

A comparison with the RELOC record shows that a SUPER RELOC record is missing the offset of the reference. Similarly, the SUPER INTERSEG1 through SUPER INTERSEG12 records are missing the segment number and offset of the subroutine referenced. The offsets (which are two bytes long) are stored in the LCONST record at the "to be patched" location. For the SUPER INTERSEG1 through 12 records, the segment number is stored in the third byte of the "to be patched" location.

For example, if the example given in the discussion of the INTERSEG record were instead referenced through a SUPER INTERSEG1 record, the value $0345 (the offset of the subroutine referenced) would be stored at offset $0721 in the load segment (the offset of the instruction's operand.) The segment number ($0A) would be stored at offset $0723, as follows:

```
4503 0A
```

General	$FB	This record contains a 4-byte count indicating the number of bytes of data that follow. This record type is reserved for use by Apple Computer, Inc..
Experimental	$FC–$FF	These records contain a 4-byte count indicating the number of bytes of data that follow. These record types are reserved by Apple Computer for use in system development.

Expressions

Several types of OMF records contain expressions. **Expressions** form an extremely flexible reverse-Polish stack language that can be evaluated by the linker to yield numeric values such as addresses and labels. Each expression consists of a series of operators and operands together with the values on which they act.

An **operator** takes one or two values from the **evaluation stack**, performs some mathematical or logical operation on them, and places a new value onto the evaluation stack. The final value on the evaluation stack is used as if it were a single value in the record. Note that this evaluation stack is purely a programming concept and does not relate to any hardware stack in the computer. Each operation is stored in the object module file in **postfix** form; that is, the value or values come first, followed by the operator. For example, since a binary operation is stored as *Value1 Value2 Operator*, the operation *Num1 - Num2* is stored as

```
Num1Num2-
```

The operators are as follows:

- **Binary math operators:** These operators take two numbers (as two's-complement signed integers) from the top of the evaluation stack, perform the specified operation, and place the single-integer result back on the evaluation stack. The binary math operators include

$01	addition	(+)
$02	subtraction	(-)
$03	multiplication	(*)
$04	division	(/, DIV)
$05	integer remainder	(//, MOD)
$07	bit shift	(<<, >>)

The subtraction operator subtracts the second number from the first number. The division operator divides the first number by the second number. The integer-remainder operator divides the first number by the second number and returns the unsigned integer remainder to the stack. The bit-shift operator shifts the first number by the number of bit positions specified by the second number. If the second number is positive, the first number is shifted to the left, filling vacated bit positions with zeros (arithmetic shift left). If the second number is negative, the first number is shifted right, filling vacated bit positions with zeros (logical shift right).

- **Unary math operator:** A unary math operator takes a number as a two's-complement signed integer from the top of the evaluation stack, performs the operation on it, and places the integer result back on the evaluation stack. The only unary math operator currently available is

$06	negation	(–)

- **Comparison operators:** These operators take two numbers as two's-complement signed integers from the top of the evaluation stack, perform the comparison, and place the single-integer result back on the evaluation stack. Each operator compares the second number in the stack (TOS – 1) with the number at the top of the stack (TOS). If the comparison is TRUE, a 1 is placed on the stack; if FALSE, a 0 is placed on the stack. The comparison operators include

$0C	less than or equal to	(<=, ≤)
$0D	greater than or equal to	(>=, ≥)
$0E	not equal	(<>, ≠, !=)
$0F	less than	(<)
$10	greater than	(>)
$11	equal to	(= or ==)

- **Binary logical operators:** These operators take two numbers as Boolean values from the top of the evaluation stack, perform the operation, and place the single Boolean result back on the stack. Boolean values are defined as being FALSE for the number 0 and TRUE for any other number. Logical operators always return a 1 for TRUE. The binary logical operators include

$08	AND	(**, AND)
$09	OR	(++, OR, |)
$0A	EOR	(–, XOR)

- **Unary logical operator:** A unary logical operator takes a number as a Boolean value from the top of the evaluation stack, performs the operation on it, and places the Boolean result back on the stack. The only unary logical operator currently available is

$0B	NOT	(¬, NOT)

- **Binary bit operators:** These operators take two numbers as binary values from the top of the evaluation stack, perform the operation, and place the single binary result back on the stack. The operations are performed on a bit-by-bit basis. The binary bit operators include

$12	Bit AND	(logical AND)
$13	Bit OR	(inclusive OR)
$14	Bit EOR	(exclusive OR)

- **Unary bit operator:** This operator takes a number as a binary value from the top of the evaluation stack, performs the operation on it, and places the binary result back on the stack. The unary bit operator is

$15	Bit NOT	(complement)

- **Termination operator:** All expressions end with the termination operator $00.

An **operand** causes some value, such as a constant or a label, to be loaded onto the evaluation stack. The operands are as follows:

- **Location-counter operand ($80):** This operand loads the value of the current location counter onto the top of the stack. Because the location counter is loaded before the bytes from the expression are placed into the code stream, the value loaded is the value of the location counter before the expression is evaluated.
- **Constant operand ($81):** This operand is followed by a number that is loaded on the top of the stack. If the size of the number is not specified, its length is specified by the NUMLEN field in the segment header.
- **Label-reference operands ($82–$86):** Each of these operand codes is followed by the name of a label and is acted on as follows:

  $82 Weak reference (see the following note.)

  $83 The value assigned to the label is placed on the top of the stack.

  $84 The length attribute of the label is placed on the top of the stack.

  $85 The type attribute of the label is placed on the top of the stack. (Type attributes are listed in the discussion of the GLOBAL record in the section "Segment Body" earlier in this appendix).

  $86 The count attribute is placed on the top of the stack. The count attribute is 1 if the label is defined and 0 if it is not.
- **Relative offset operand ($87):** This operand is followed by a number that is treated as a displacement from the start of the segment. Its value is added to the value that the location counter had when the segment started, and the result is loaded on the top of the stack.

- *Weak reference:* The operand code $82 is referred to as the **weak reference.** The weak reference is an instruction to the linker that asks for the value of a label, if it exists. It is not an error if the linker cannot find the label. However, the linker does not load a segment from a library if only weak references to it exist. If a label does not exist, a 0 is loaded onto the top of the stack. This operand is generally used for creating jump tables to library routines that may or may not be needed in a particular program.

Example

Assume your assembly-language program contains the following line, where MSG4 and MSG3 are global labels:

```
        LDX     #MSG4-MSG3
```

This line is assembled into two OMF records:

```
CONST ($01)          A2
EXPR ($EB)           02 : MSG4MSG3-
```

In hexadecimal format, these records appear as follows:

```
01 A2   ."
EB 02 83 04 4D 53 47 34 83 04 4D 53 47 33 02 00    k...MSG4..MSG3..
```

The initial $01 is the OMF opcode for a 1-byte constant. The $A2 is the 65816 opcode for the LDX instruction. The $EB is the OMF opcode for an EXPR record, which is followed by a 1-byte count indicating the number of bytes to which the expression is to be truncated ($02 in this case). The next number, $83, is a label-reference operand for the first label in the expression, indicating that the value assigned to the label (MSG4) is to be placed on top of the evaluation stack. Next is a length byte ($04), followed by MSG4 spelled out in ASCII codes.

The next sequence of codes, starting with $83, places the value of MSG3 on the evaluation stack. Finally, the expression-operator code $02 indicates that subtraction is to be performed, and the termination operator ($00) indicates the end of the expression.

◆ *Viewing expressions:* You can use the DumpObj tool provided with some development environments to examine the contents of any OMF file. DumpObj can list the header contents of each segment and can list the body of each segment in OMF format, 65816 disassembly format, or as hexademical codes. See your development-environment manuals for instructions.

Object files

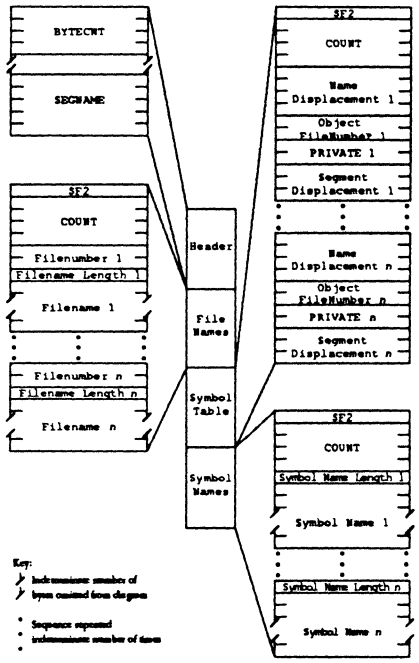
Object files (file type $B1) are created from source files by a compiler or assembler. Object files can contain any of the OMF record types except INTERSEG, cINTERSEG, RELOC, cRELOC, SUPER, and ENTRY. Object files can contain unresolved references, because all references are resolved by the linker. If you are writing a compiler for the Apple IIGS, you can use the DumpObjIIGS tool to examine the contents of a variety of object files to get an idea of their content and structure.

Library files

Library files (file type $B2) contain object segments that the linker can search for external references. Usually, these files contain general routines that can be used by more than one application. The linker extracts from the library file any object segment that contains an unresolved global definition that was referenced during the link. This segment is then added to the load segment that the linker is currently creating.

Library files differ from object files in that each library file includes a segment called the *library dictionary segment* (segment kind = $08). The library dictionary segment contains the names and locations of all segments in the library file. This information allows the linker to scan the file quickly for needed segments. Library files are created from object files by a MakeLib tool (provided with a development environment). The format of the library dictionary segment is illustrated in Figure B-3.

■ **Figure B-3** The format of a library dictionary segment



The library dictionary segment begins with a segment header, which is identical in form to other segment headers. The BYTECNT field indicates the number of bytes in the library dictionary segment, including the header. The body of the library dictionary segment consists of three LCONST records, in this order:

1. Filenames
2. Symbol table
3. Symbol names

The **filenames record** consists of one or more subrecords, each consisting of a 2-byte file number followed by a filename. The filename is in Pascal string format; that is, a length byte indicating the number of characters, followed by an ASCII string. The filenames are the full pathnames of the object files from which the segments in this library file were extracted. The file numbers are assigned by the MakeLib program and used only within the library file. These file numbers are not related to the load-file numbers in the pathname table.

The **symbol table** record consists of a cross-reference between the symbol names in the symbol-names record and the object segments in which the symbol names occur. For each global symbol in the library file, the symbol table record contains the following components:

1. A 4-byte displacement into the symbol names record, indicating the start of the symbol name.
2. The 2-byte file number of the file in which the name occurred. This is the file number assigned by the MakeLib utility and used in the filenames record of this library dictionary segment.
3. A 2-byte flag, the private flag. If this flag equals 1, the symbol name is valid only in the object file in which it occurred (that is, the symbol name was in a private segment). If this flag equals 0, the symbol name is not private.
4. A 4-byte displacement into the library file indicating the beginning of the object segment in which the symbol occurs. The displacement is to the beginning of the segment even if the symbol occurs inside the segment; the location within the segment is resolved by the linker.

The **symbol names record** consists of a series of symbol names; each symbol name consists of a length byte followed by up to 255 ASCII characters. All global symbols that appear in an object segment, including entry points and global equates, are placed in the library dictionary segment. Duplicate symbols are not allowed.

Load files

Load files (file types $B3 through $BE) contain the load segments that are moved into memory by the System Loader. They are created by a linker from object files and library files. Load files conform to the object module format but are restricted to a small subset of that format. Because the segments must be quickly relocated and loaded, they cannot contain any unresolved symbolic information.

All load files are composed of load segments. The format of each load segment is a loadable binary memory image followed by a relocation dictionary. Load files can contain any of several special segment types:

- jump-table segment
- pathname segment
- initialization segment
- direct-page/stack segment

Each of these segment types is described in the following sections.

The load segments in a load file are numbered by their relative location in the load file, where the first load segment is number 1. The segment number is used by the System Loader to find a specific segment in a load file.

Memory image and relocation dictionary

Each load segment consists of two parts, in this order:

1. A memory image comprising long-constant (LCONST) records and define-storage (DS) records. These records contain all of the code and data that do not change with load address (these records reserve space for location-dependent addresses). The DS records are inserted by the linker (in response to DS records in the object file) to reserve large blocks of space, rather than putting large blocks of zeros in the load file.
2. A relocation dictionary that provides the information necessary to patch the LCONST records at load time. The relocation dictionary contains relocation (RELOC, cRELOC, or SUPER RELOC) records and intersegment (INTERSEG, cINTERSEG, or SUPER INTERSEG) records.

When the System Loader loads the segment into memory, it loads each LCONST record or DS record in one piece; then it processes the relocation dictionary. The relocation dictionary includes only RELOC (or cRELOC or SUPER RELOC) and INTERSEG (or cINTERSEG or SUPER INTERSEG) records. The RELOC records provide the information the loader needs to recalculate the values of location-dependent local references, and the INTERSEG records provide the information it needs to transfer control to external references. For more information, see the discussions of the RELOC and INTERSEG records in the section "Segment Body," earlier in this appendix. The sequence of events that occurs when a JSL to an external dynamic segment is executed is described in general in Appendix A of this volume.

Jump-table segment

The **jump-table segment**, when used, is the segment of a load file that contains the calls to the System Loader to load dynamic segments. Each time the linker comes across a statement that references a label in a dynamic segment, it generates an entry in the jump-table segment for that label (it also creates an entry in the relocation dictionary). The entry in the jump-table segment contains the file number, segment number, and offset of the reference in the dynamic segment, plus a call to the System Loader to load the segment. The relocation dictionary entry provides the information the loader needs to patch a call to the jump-table segment into the memory image.

The segment type of the jump-table segment is KIND = $02. There is one jump-table segment per load file; it is a static segment, and it is loaded into memory at program boot time at a location determined by the Memory Manager. The System Loader maintains a list, called the jump-table directory (or just the jump table), of the jump-table segments in memory.

Each entry in the jump-table segment corresponds to a call to an external (intersegment) routine in a dynamic segment. The jump-table segment initially contains entries in the unloaded state. When the external call is encountered during program execution, a jump to the jump-table segment occurs. The code in the jump-table segment entry, in turn, jumps to the System Loader. The System Loader figures out which segment is referenced and loads it. Next, the System Loader changes the entry in the jump-table segment to the loaded state. The entry stays in the loaded state as long as the corresponding segment is in memory. If the application tells the System Loader to unload a segment, all jump-table segment entries that reference that segment are changed to their unloaded states.

Unloaded state

The unloaded state of a jump-table segment entry contains the code that calls the System Loader to load the needed segment. An entry contains the following fields:

- user ID (two bytes)
- load-file number (two bytes)
- load-segment number (two bytes)
- load-segment offset (four bytes)
- JSL to jump-table load function (four bytes)

The user ID field is reserved for the identification number assigned to the program by the UserID Manager; until initial load time, this field is 0. The load-file number, load-segment number, and load-segment offset refer to the location of the external reference. The rest of the entry is a call to the System Loader jump-table load function (an internal routine). The user ID and the address of the load function are patched by the System Loader during initial load. See Appendix A of this Volume for information about the jump-table load function. A load-file number of 0 indicates that there are no more entries in this jump-table segment. (There may be other jump-table segments for this program, however—each load file that is part of a program has its own jump-table segment.)

Loaded state

The loaded state of a jump-table segment entry is identical to the unloaded state except that the JSL to the System Loader jump-table load function is replaced by a JML to the external reference. A loaded entry contains the following fields:

- user ID (two bytes)
- load-file number (two bytes)
- load-segment number (two bytes)
- load-segment offset (four bytes)
- JML to external reference (four bytes)

◆ *Version differences:* In Versions 1.0 and 2.0 of the OMF, the jump-table segment starts with eight bytes of zeros. In future versions of the OMF, these zeros may be eliminated.

Pathname segment

The **pathname segment** is a segment in a load file that is created by the linker to help the System Loader find the load segments of run-time library files that must be loaded dynamically. It provides a cross-reference between file numbers and file pathnames. The segment type of the pathname segment is KIND = $04. When the loader processes the load file, it adds the information in the pathname segment to the pathname table that it maintains in memory. Pathname tables are described Appendix A of this volume.

The pathname segment contains one entry for each load file and for each run-time library file referenced in a load file. The format of each entry is as follows:

file number (two bytes)
file date and time (eight bytes)
file pathname (length byte and ASCII string)

The **file number** is a number assigned by the linker to a specific load file. File number 1 is reserved for the load file in which the pathname segment resides (usually the load file of the application program). A file number of 0 indicates that there are no more entries in this pathname segment.

The **file date and time** are directory items retrieved by the linker during the link process. The System Loader compares these values with the directory of the run-time library file at run time. If they are not the same, the System Loader does not load the requested load segment, thus ensuring that the run-time library file used at link time is the same as the one loaded at execution time.

The **file pathname** is the pathname of the load file. The pathname is listed as a Pascal-type string: that is, a length byte followed by an ASCII string. A pathname segment created by the linker may contain partial pathnames. A partial pathname begins with one of the prefixes supported by the operating system; these prefixes have the form n/, where n is a number from 0 to 31. The first three prefixes have fixed definitions, as follows:

0/ system prefix (initially the volume from which the operating system was booted)

1/ application subdirectory (the subdirectory out of which the application is running)

2/ system library subdirectory (initially /boot_volume/SYSTEM/LIBS/)

Initialization segment

The **initialization segment** is an optional segment in a load file. When the System Loader encounters an initialization segment during the initial loading of segments, it transfers control to the initialization segment. After the initialization segment returns control to the System Loader, the loader continues the normal initial load of the remaining segments in the load file. The segment type of the initialization segment is KIND = $10.

You might use an initialization segment, for example, to initialize the graphics environment of an application and to display a "splash screen" (such as a copyright message and company logo) for the duration of the program load.

The initialization segment does not have to be the first segment loaded, there may be more than one initialization segment, and an initialization segment can make references to other segments previously loaded.

The initialization segment must obey the following rules:

- It must not reference any segments not yet loaded.
- It must exit with an RTL instruction.
- It must not change the stack pointer.
- It must not use the current direct page. To avoid writing over a portion of the direct page being used by the loader, the initialization segment must allocate its own direct page if it needs direct-page space.

◆ *Restart:* Initialization segments are re-executed during the restart of an application from memory.

Direct-page/stack segments

The Apple IIGS stack can be located anywhere in the lower 48kb of bank $00 and can be any size up to 48kb. The direct page is the Apple IIGS equivalent of the zero page of 8-bit Apple II computers; the direct page can also be located anywhere in the lower 48kb of bank $00. Like the zero page, the direct page occupies 256 bytes of memory; on the Apple IIGS, however, a program can move its direct page while it is running. Consequently, a given program can use more than 256 bytes of memory for direct-page functions.

Each program running on the Apple IIGS reserves a portion of bank $00 as a combined direct-page/stack space. Because more than one application can be loaded in memory at one time on the Apple IIGS, more than one stack and one direct page could be in bank $00 at a given time. Furthermore, some applications may place some of their code in bank $00. A given program should therefore probably not use more than about 4kb for its direct-page/stack space.

When an instruction uses one of the direct-page addressing modes, the effective address is calculated by adding the value of the operand of the instruction to the value in the direct-page register. The stack pointer, on the other hand, is decremented each time a stack-push instruction is executed. The convention used on the Apple IIGS, therefore, is for the direct page to occupy the lower part of the direct-page/stack space, whereas the stack grows downward from the top of the space.

△ **Important** GS/OS provides no mechanism for detecting stack overflow or underflow, or collision of the stack with the direct page. Your program must be carefully designed to make sure those conditions cannot occur. △

If you do not define a direct-page/stack segment in your program, GS/OS assigns a 1024-byte direct-page/stack when the System Loader InitialLoad or Restart call is executed. *****or is it 4K now?***** To specify the size and contents of the direct-page/stack space, follow the procedures outlined in Chapter 2 ("GS/OS and Its Environment") of Volume 1.

Run-time library files

Run-time library files (file type $B4) contain dynamic load segments that the System Loader can load when these segments are referenced through the jump table. Usually, run-time library files contain general routines that can be used by more than one application.

When you include a run-time library file while linking, the file is scanned by the linker during the link process. When the linker finds a referenced segment in the run-time library file, it generates an INTERSEG reference to the segment in the relocation dictionary and adds an entry to the jump-table segment for that file. The linker also adds the pathname of the run-time library file to the pathname table if it has not already done so. It does not extract the segment from the file and place it in the file that referenced it, as it does for ordinary library files. In other words, references to segments in run-time library files are treated by the linker like references to other dynamic segments, except that the run-time library file segments are in a file other than the currently-executing load file.

The first load segment of the run-time library file contains all the information the linker needs to find referenced segments; it is not necessary for the linker to scan every subroutine in every segment each time a subroutine is referenced. The first segment contains a table of ENTRY records, each one corresponding to a segment name or global reference in the run-time library file.

Run-time library files are typically created from corresponding object files by specifying an option to a linker command.

Shell applications

Shell applications (file type $B5) are executable load files that are run under an Apple IIGS shell program, such as the APW Shell. The shell calls the System Loader's InitialLoad function and transfers control to the shell application by means of a JSL instruction, rather than launching the program through the GS/OS Quit function. Therefore, the shell does not shut down, and the program can use shell facilities during execution. The program returns control to the shell with an RTL instruction, or with a GS/OS Quit call if the shell intercepts and acts on GS/OS calls. (Development-environment shells might intercept GS/OS Quit calls.) Shell applications should use standard Text Tool Set calls for all nongraphics I/O. The shell program is responsible for initializing the Text Tool Set routines.

- *Running shell files stand-alone:* A load file of file type $B5 can be launched by GS/OS by way of the Quit call if it requires no support other than standard input from the keyboard and output to the screen. GS/OS initializes the Text Tool Set to use the Pascal I/O drivers (see the *Apple IIGS Toolbox Reference*) for the keyboard and 80-column screen. Only $B5 files that end in a GS/OS Quit call can be run in this way.

As soon as a shell application is launched, it should check the X and Y registers for a pointer to the shell-identifier string and input line. The X register holds the high word and the Y register holds the low word of this pointer. The shell program is responsible for loading this pointer into the index registers and for placing the following information in the area pointed to:

1. An 8-byte ASCII string containing an identifier for the shell. (The identifier for the APW Shell, for example, is BYTEWRKS.) The shell application should check this identifier to make sure that it has been launched by the correct shell, so that the environment it needs is in place. If the shell identifier is not correct, the shell application should write an error message to standard error output (normally the screen) and then exit with an RTL instruction (or a GS/OS Quit call if the shell intercepts GS/OS calls).
2. A null-terminated ASCII string containing the input line for the shell application. The shell program can strip any I/O redirection or pipeline commands from the input line, since those commands are intended for the shell itself, but must pass on all input parameters intended for the shell application.

The shell program must request a user ID for the shell application; the user ID is passed in the accumulator. The shell must set up a direct-page and stack area for the shell application. The shell places the address of the start of the direct-page/stack space in the direct-page (D) register and sets the stack pointer (S register) to point to the last byte of the block. If the shell application does not have a direct-page/stack segment, the shell should follow the same conventions used by GS/OS for default direct-page/stack allocation. See the section "Direct-Page/Stack Segments" earlier in this appendix, and Chapter 2 of Volume 1 for more information about direct-page and stack allocation.

◆ *GS/OS:* GS/OS does not support the identifier string or input line. If the shell application is launched by GS/OS, the X and Y registers contain zeros.

Some shell applications may launch other programs; for example, a shell nested within another shell would be a shell application. When a shell application requests a user ID for a program, the calling program is responsible for intercepting GS/OS Quit calls and system resets, so that it can remove from memory all memory buffers with that user ID before passing control to the shell.

A shell application should use the following procedure to quit:

1. If the shell application has launched any programs, it must call the System Loader's UserShutdown function to shut down those programs.
2. The shell application should release any memory buffers that it has requested and dispose of their handles.
3. The shell application must place an error code in the accumulator. If no error occurred, the error code should be $0000. The error code $FFFF is used as a general (nonspecific) error code. For a shell program you write, you can define any other error codes you want to use, and you can handle them in any way you wish.

4. The shell application should execute an RTL or a GS/OS Quit call. If the program ends in a Quit call, the shell program that launched the shell application is responsible for intercepting the Quit call, releasing all memory buffers associated with that shell application, and performing any other system tasks normally done by GS/OS in response to a Quit call.

△ **Important** When a shell launches a shell application, the address of the shell program is not pushed onto the GS/OS Quit Return stack; therefore, the shell itself must handle the shell application's Quit call, or control is not returned to the shell. To intercept the Quit call, the shell program must intercept all GS/OS calls. The shell may pass on any other operating system calls to GS/OS, but it must handle Quit calls itself. If the shell you are using does not handle GS/OS calls in this fashion, the shell application must end in an RTL instruction. △

Appendix C Generated Drivers and Firmware Drivers

This appendix provides information of use to designers of BASIC, Pascal 1.1, ProDOS, SmartPort, and extended SmartPort pertipheral 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 a driver it is.

See also Chapter 7 of this Volume 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 zero 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

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.

◆ *Addresses:* 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
- has no loaded driver
- has no ProDOS, Pascal 1.1, or SmartPort ID bytes

Dispatching

Contrary to the documented standard (see, for example, the *Apple IIc Technical Reference Manual*), 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 the input routine entry point
$C*n*00+($0036)	Add contents of $0036 to $C*n*00 to get the 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* = the slot where the driver resides)
Y Register	$*n*0 (*n* = the slot where the 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 ssumes 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* = the slot where the driver resides)
Y Register	$*n*0 (*n* = the slot where the 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 IIc Technical Reference Manual.*

Generated-driver interface

Pascal 1.1 firrmware 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 ssumes 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 the value of the byte at address $C*n*FF to $C*n*00 to get the 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
$0044-45	Buffer pointer
$0046-47	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 (Format_Device) of the GS/OS driver call Driver_Control. See Chapter 11 of this Volume.

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 ssumes 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 adresses and therefore cannot access or reside in Apple IIGS memory beyond bank $00. The **extended** version 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 the value of the byte at address $C*n*FF) to $C*n*00 to get the 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 occured, 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 do 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 D **Driver Source Code Samples**

This appendix demonstrates four different types of drivers: a block driver, a character driver, a supervisory driver, and a device driver that calls a supervisory driver. It consists of fully-commented assembly-language source-code listings for all four drivers. ■

△ **Important** These source-code examples are not executable as they stand. Use them as guides to writing your own drivers, but don't expect that the code here can be copied exactly. For one thing, there are missing parts: not all call handlers are implemented for all the drivers Furthermore, some of the drivers access fictitious firmware locations. △

The drivers in this appendix have three essential components: the driver entry point, the driver dispatch table, and the driver routines.

- **The driver entry point** is the beginning of the code section of the driver. It is the one entry for all driver calls. Code following the entry point does intitial checking and bookkeeping before using the driver dispatch table to jump to the proper driver routine.
- **The driver dispatch table** is a jump table containing offsets to all the supported driver routines.
- **The driver routines** are the code that handles all driver calls. Drivers are expected to have routines to handle all appropriate standard driver calls; they can also include routines to handle any needed device-specific calls. See Chapter 11 for descriptions of how drivers handle standard driver calls.

In addition to these components, the driver code section may include other routines, such as interrupt handlers, signal sources, and signal handlers. See Chapter 10.

Block driver

This is a typical driver for a block-oriented device such as a disk drive. It includes handlers for all standard driver calls, although in this example not all of the handlers are functional. The driver code consists of eight parts, in this order:

- Equates
- Device-driver header
- Configuration parameter lists (8 of them, for 8 supported devices)
- Format option tables (8 of them)
- Device information blocks (DIBs; 8 of them)
- Tables for dispatching calls and passing parameters
- A main entry point to the driver
- Routines that handle the driver calls

The driver has routines to handle all standard driver calls, including the standard Status and Control subcalls. Even though it is a block-device driver, for which Open and Close calls are not meaningful, handlers for them are included.

```
                65816           on
                instime         on
                gen             on
                symbol          on
                absaddr         n
                align           256
*************************************************************
*
*                       Copyright (c) 1987, 1988
*                       Apple Computer, Inc.
*                       All rights reserved.
*
*************************************************************
*
*                       Driver Core Routines Version 0.01a01
*
* NOTE:                 All driver files must be installed on the
*                       boot volume in the subdirectory "/SYSTEM/DRIVERS'.
*                       Additionally, the FileType for the driver file
*                       must be set to $00BB.  AuxType is also critical
*                       to the operating system recognizing the driver
*                       as a GS/OS device driver.  The AuxType is a long
*                       word which must have the upper word set to $0000.
*                       The most significant byte of the least significant
*                       word in the AuxType must be set to $01 to indicate
*                       an active GS/OS device driver or $81 to indicate
*                       an inactive GS/OS device driver.  The least
*                       significant byte of the least significant word
*                       of the AuxType field indicates the number of
*                       devices supported by the driver file.  This value
*                       should be analogous to the number of DIB's
*                       contained in the driver file.  GS/OS will only
*                       install the number of devices indicated in the
*                       AuxType field.
*
*                       GS/OS Device Driver:        FileType = $00BB
*                                                   AuxType  = $000001XX where:
*                                                               XX = number of devices.
*
*                       An AuxType of $00000108 indicates eight devices.  When
*                       building a device driver, the best way to set the
*                       FileType and AuxType is to use the Exerciser to get
*                       the current file info (GET_FILE_INFO), modify the
*                       FileType & AuxType and then SET_FILE_INFO.
*
```

```
****************************************************************
*
* REVISION HISTORY:
*
* DATE              Ver.        By          Description
*
* 11/16/87          0.00e01     RBM         Started initial coding.
*
* 01/10/88          0.00e02     RBM         Added new status & control calls.
* 01/11/88                                  Fixed startup for dynamic slot numbers.
*
* 02/04/88          0.00a01 RBM             General update for Alpha release.
*
* 04/11/88          0.06a01 RBM             New STARTUP.
*                                           New SHUTDOWN.
*                                           Additional control & status calls.
*                                           Removed valid access parsing performed by dispatcher.
*
* 12/21/88          1.00        RBM         Startup call now uses system service call to dynamic slot
                                            arbiter.
*
****************************************************************

                    eject
****************************************************************
*
* The following are direct page equates on the GS/OS
* direct page for driver usage.
*
****************************************************************
drvr_dev_num        gequ        $00                 ; (w)  device number
drvr_call_num       gequ        drvr_dev_num+2      ; (w)  call number
drvr_buf_ptr        gequ        drvr_call_num+2     ; (lw) buffer pointer
drvr_slist_ptr      gequ        drvr_call_num+2     ; (lw) buffer pointer
drvr_clist_ptr      gequ        drvr_call_num+2     ; (lw) buffer pointer
dev_id_ref          gequ        drvr_buf_ptr        ; (w)  indirect device ID
drvr_req_cnt        gequ        drvr_buf_ptr+4      ; (lw) request count
drvr_tran_cnt       gequ        drvr_req_cnt+4      ; (lw) transfer count
drvr_blk_num        gequ        drvr_tran_cnt+4     ; (lw) block number
drvr_blk_size       gequ        drvr_blk_num+4      ; (w)  block size
drvr_fst_num        gequ        drvr_blk_size+2     ; (w)  File System Translator Number
drvr_stat_code      gequ        drvr_fst_num        ; (w)  status code for status call
drvr_ctrl_code      gequ        drvr_fst_num        ; (w)  control code for control call
drvr_vol_id         gequ        drvr_fst_num+2      ; (w)  Driver Volume ID Number
drvr_cache          gequ        drvr_vol_id+2       ; (w)  Cache Priority
drvr_cach_ptr       gequ        drvr_cache+2        ; (lw) pointer to cached block
drvr_dib_ptr        gequ        drvr_cach_ptr+4     ; (lw) pointer to active DIB

sib_ptr             gequ        $0074               ; (lw) pointer to active SIB
sup_parm_ptr        gequ        sib_ptr+4           ; (lw) pointer to supervisor parameters

                    eject
```

```
**************************************************************
*
* The following are equates for driver command types.
*
**************************************************************
drvr_startup        gequ     $0000        ; driver startup command
drvr_open           gequ     $0001        ; driver open command
drvr_read           gequ     $0002        ; driver read command
drvr_write          gequ     $0003        ; driver write command
drvr_close          gequ     $0004        ; driver close command
drvr_status         gequ     $0005        ; driver status command
drvr_control        gequ     $0006        ; driver control command
drvr_flush          gequ     $0007        ; driver flush command
drvr_shutdn         gequ     $0008        ; driver shutdown command
max_command         gequ     $0009        ; commands $0009 - $ffff undefined

drvr_dev_stat       gequ     $0000        ; status code: return device status
drvr_conf_stat      gequ     $0001        ; status code: return configuration params
drvr_get_wait       gequ     $0002        ; status code: get wait/no wait mode
drvr_get_format     gequ     $0003        ; status code: get format options

drvr_reset          gequ     $0000        ; control code: reset device
drvr_format         gequ     $0001        ; control code: format device
drvr_eject          gequ     $0002        ; control code: eject media
drvr_set_conf       gequ     $0003        ; control code: set configuration params
drvr_set_wait       gequ     $0004        ; control code: set wait/no wait mode
drvr_set_format     gequ     $0005        ; control code: set format options
drvr_set_ptn        gequ     $0006        ; control code: set partition owner
drvr_arm            gequ     $0007        ; control code: arm interrupt signal
drvr_disarm         gequ     $0007        ; control code: arm interrupt signal

                    eject
```

```
***************************************************************
*
* The following are equates for GS/OS error codes.
*
***************************************************************
no_error            gequ      $0000        ; no error has occurred
dev_not_found       gequ      $0010        ; device not found
invalid_dev_num     gequ      $0011        ; invalid device number
drvr_bad_req        gequ      $0020        ; bad request or command
drvr_bad_code       gequ      $0021        ; bad control or status code
drvr_bad_parm       gequ      $0022        ; bad call parameter
drvr_not_open       gequ      $0023        ; character device not open
drvr_prior_open     gequ      $0024        ; character device already open
irq_table_full      gequ      $0025        ; interrupt table full
drvr_no_resrc       gequ      $0026        ; resources not available
drvr_io_error       gequ      $0027        ; I/O error
drvr_no_dev         gequ      $0028        ; device not connected
drvr_busy           gequ      $0029        ; call aborted, driver is busy
drvr_wr_prot        gequ      $002B        ; device is write protected
drvr_bad_count      gequ      $002C        ; invalid byte count
drvr_bad_block      gequ      $002D        ; invalid block address
drvr_disk_sw        gequ      $002E        ; disk has been switched
drvr_off_line       gequ      $002F        ; device off line / no media present
invalid_access      gequ      $004E        ; access not allowed
parm_range_err      gequ      $0053        ; parameter out of range
out_of_mem          gequ      $0054        ; out of memory
dup_volume          gequ      $0057        ; duplicate volume name
not_block_dev       gequ      $0058        ; not a block device
stack_overflow      gequ      $005F        ; too many applications on stack
data_unavail        gequ      $0060        ; data unavailable

                    eject
```

```
****************************************************************
*
* The following are equates for the DIB.
*
****************************************************************
link_ptr            gequ     $0000        ; (lw) pointer to next DIB
entry_ptr           gequ     $0004        ; (lw) pointer to driver
dev_char            gequ     $0008        ; (w)  device characteristics
blk_cnt             gequ     $000A        ; (lw) number of blocks
dev_name            gequ     $000E        ; (32) count and ascii name (pstring)
slot_num            gequ     $002E        ; (w)  slot number
unit_num            gequ     $0030        ; (w)  unit number
ver_num             gequ     $0032        ; (w)  version number
dev_id_num          gequ     $0034        ; (w)  device ID number (ICON ref#)
head_link           gequ     $0036        ; (w)  backward device link
forward_link        gequ     $0038        ; (w)  forward device link
ext_dib_ptr         gequ     $003A        ; (lw) dib reserved field #1
dib_dev_num         gequ     $003E        ; (w)  Device number of this device
*
* The following equate(s) are for drive specific extensions to the DIB.
* Parameters that are extended to the manditory DIB parameters are not
* accessable by GS/OS or the application but may be used within a driver
* as needed.
*
driver_unit         gequ     $0040        ; (w)  driver's internal DIB data
my_slot16           gequ     $0042        ; (w)  driver's slot * 16

                    eject
****************************************************************
*
* System Service Table Equates:
*
* NOTE: Only those system service calls that might be used
* by a device driver are listed here.  For a more complete
* list of system service calls and explanations of each call
* consult the system service call ERS.
*
****************************************************************
dev_dispatcher      gequ     $01FC00      ; dev_dispatch
cache_find_blk      gequ     $01FC04      ; cash_find
cache_add_blk       gequ     $01FC08      ; cash_add
cache_del_blk       gequ     $01FC14      ; cash_delete
cache_del_vol       gequ     $01FC18      ; cash_del_vol
set_sys_speed       gequ     $01FC50      ; set system speed
move_info           gequ     $01FC70      ; gs_move_block
set_disksw          gequ     $01FC90      ; set disksw and call swapout/delvol
sup_drvr_disp       gequ     $01FCA4      ; supervisor dispatcher
install_driver      gequ     $01FCA8      ; dynamic driver installation
dyn_slot_arbiter    gequ     $01FCBC      ; dynamic slot arbiter

                    eject
```

```
***************************************************************
*
* MOVE_INFO
*
* NOTE: The following equates are used to set the modes
* passed to the move_info call system service call.
*
***************************************************************
moveblkcmd          gequ     $0800          ; block move option
move_sinc_dinc      gequ     $0805          ; souce increment, dest. increment

move_sinc_ddec      gequ     $0809          ; souce increment, dest. decrement
move_sdec_dinc      gequ     $0806          ; souce decrement, dest. increment
move_sdec_ddec      gequ     $080A          ; souce decrement, dest. decrement

move_scon_dcon      gequ     $0800          ; souce constant, dest. constant
move_sinc_dcon      gequ     $0801          ; souce increment, dest. constant
move_sdec_dcon      gequ     $0802          ; souce decrement, dest. constant
move_scon_dinc      gequ     $0804          ; souce constant, dest. increment
move_scon_ddec      gequ     $0808          ; souce constant, dest. decrement

                    eject

; __7______6______5______4______3______2______1______0___
; |      |      |      |      |      |      |      |      |
; |slot7 |slot6 |slot5 |slot4 |      |slot2 |slot1 |      |
; |intext|intext|intext|intext|  0   |intext|intext|  0   |
; |enable|enable|enable|enable|      |enable|enable|      |
; |______|______|______|______|______|______|______|______|
;        ^^^^^  sltromsel byte ^^^^^
;
; sltromsel bits defined as follows
;     bit 7= 0 enables internal slot 7 -- 1 enables slot rom
;     bit 6= 0 enables internal slot 6 -- 1 enables slot rom
;     bit 5= 0 enables internal slot 5 -- 1 enables slot rom
;     bit 4= 0 enables internal slot 4 -- 1 enables slot rom
;     bit 3= must be 0
;     bit 2= 0 enables internal slot 2 -- 1 enables slot rom
;     bit 1= 0 enables internal slot 1 -- 1 enables slot rom
;     bit 0= must be 0

sltromsel           gequ     $00C02D        ; slot rom select
```

```
; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;|      | stop |      | stop | stop | stop | stop | stop |
;|  0   |i/o/lc|  0   |auxh-r|suprhr|hires2|hires1|txt pg|
;|      |shadow|      |shadow|shadow|shadow|shadow|shadow|
;|______|______|______|______|______|______|______|______|
;        ^^^^^  shadow byte ^^^^^
;
; shadow bits defined as follows
;    bit 7= must write 0
;    bit 6= 1 to inhibit i/o and language card operation
;    bit 5= must write 0
;    bit 4= 1 to inhibit shadowing aux hi-res page
;    bit 3= 1 to inhibit shadowing 32k video buffer
;    bit 2= 1 to inhibit shadowing hires page 2
;    bit 1= 1 to inhibit shadowing hires page 1
;    bit 0= 1 to inhibit shadowing text pages

shadow                 gequ      $00C035          ; shadow register

         eject

; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;| slow/|      |      |shadow|slot 7|slot 6|slot 5|slot 4|
;| fast |  0   |  0   |in all|motor |motor | motor| motor|
;| speed|      |      | ram  |detect|detect|detect|detect|
;|______|______|______|______|______|______|______|______|
;        ^^^^^  cyareg byte ^^^^^
;
; cyareg bits defined as follows
;    bit 7= 0=slow system speed -- 1=fast system speed
;    bit 6= must write 0
;    bit 5= must write 0
;    bit 4= shadow in all ram banks
;    bit 3= slot 7 disk motor on detect
;    bit 2= slot 6 disk motor on detect
;    bit 1= slot 5 disk motor on detect
;    bit 0= slot 4 disk motor on detect
```

```
cyareg                  gequ     $00C036         ; speed and motor on detect

; __7______6______5______4______3______2______1______0___
; |      |      |      |      |      |      |      |      |
; | alzp | page2| ramrd|ramwrt| rdrom|lcbnk2|rombnk| intcx|
; |status|status|status|status|status|status|status|status|
; |______|______|______|______|______|______|______|______|
;           ^^^^^  statereg status byte ^^^^^
;
; statereg bits defined as follows
;    bit 7= alzp status
;    bit 6= page2 status
;    bit 5= ramrd status
;    bit 4= ramwrt status
;    bit 3= rdrom status (read only ram/rom (0/1))
;
; important note:
;           do two reads to $c083 then change statereg
;           to change lcram/rom banks (0/1) and still
;           have the language card write enabled.
;
;    bit 2= lcbnk2 status 0=LC bank 0 - 1=LC bank 1
;    bit 1= rombank status
;    bit 0= intcxrom status

statereg                gequ     $00C068         ; state register

clrrom                  gequ     $00CFFF         ; switch out $c8 roms

                        eject
```

```
*************************************************************
*
* EQUATES for the IWM require index of (n*16)
*
*************************************************************
phaseoff             gequ      $00C080          ; stepper phase off
phaseon              gequ      $00C081          ; stepper phase on

ph0off               gequ      $00C080          ; phase 0 off
ph0on                gequ      $00C081          ; phase 0 on
ph1off               gequ      $00C082          ; phase 1 off
ph1on                gequ      $00C083          ; phase 1 on
ph2off               gequ      $00C084          ; phase 2 off
ph2on                gequ      $00C085          ; phase 2 on
ph3off               gequ      $00C086          ; phase 3 off
ph3on                gequ      $00C087          ; phase 3 on

motoroff             gequ      $00C088          ; disk motor off
motoron              gequ      $00C089          ; disk motor on

drv0en               gequ      $00C08A          ; select drive 0
drv1en               gequ      $00C08B          ; select drive 1

q6l                  gequ      $00C08C          ; Q6 low
q6h                  gequ      $00C08D          ; Q6 high
q7l                  gequ      $00C08E          ; Q7 low
q7h                  gequ      $00C08F          ; Q7 high

emulstack            gequ      $010100          ; emulation mode stack pointer

                     eject
```

```
****************************************************************
*
* The following equates are used to implement our hypothetical
* device driver.  They in no way reflect softswitches associated
* with any real hardware device.
*
****************************************************************

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |        BLOCK_RDY
* |___|___|___|___|___|___|___|___|
*   |   |___|___|___|___|___|___|__ Reserved
*   |______________________________ 1 = Device is ready
*
block_rdy            gequ      $00C084

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |        BLOCK_DATA
* |___|___|___|___|___|___|___|___|
*   |___|___|___|___|___|___|___|__ Block device data register
*
block_data           gequ      $00C085

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |        BLOCK_STATUS
* |___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |__ 1 = Disk switched
*   |   |   |   |   |   |   |______ 1 = Interrupt in process
*   |   |   |   |   |   |__________ 1 = Write protected
*   |   |   |   |   |______________ 0
*   |   |   |   |__________________ 1 = Online
*   |   |   |______________________ 0
*   |   |__________________________ 0
*   |______________________________ 0
*
block_status         gequ      $00C086

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |        BLOCK_CONTROL
* |___|___|___|___|___|___|___|___|
*   |___|___|___|___|___|___|___|__ Block device control register
*
block_control        gequ      $00C087

         eject
```

```
***************************************************************
*
* The following table is the header required for all loaded
* drivers which consists of the following:
*
*        Word     Offset from start to 1st DIB
*        Word     Number of DIBs
*        Word     Offset from start to 1st configuration list
*        Word     Offset from start to 2nd configuration list
*        etc.
*
***************************************************************
driver_data       data
here              entry
                  dc       i2'dib_1-here'            ; offset to 1st DIB
                  dc       i2'8'                     ; number of devices
                  dc       i2'conf1-here'            ; offset to 1st configuration list
                  dc       i2'conf2-here'            ; offset to 2nd configuration list
                  dc       i2'conf3-here'            ; offset to 3rd configuration list
                  dc       i2'conf4-here'            ; offset to 4th configuration list
                  dc       i2'conf5-here'            ; offset to 5th configuration list
                  dc       i2'conf6-here'            ; offset to 6th configuration list
                  dc       i2'conf7-here'            ; offset to 7th configuration list
                  dc       i2'conf8-here'            ; offset to 8th configuration list
*
* The following are the driver configuration parameter lists.
*
conf1             dc       i2'0'                     ; 0 bytes in parameter list
default1          dc       i2'0'                     ; 0 bytes in default list

conf2             dc       i2'0'                     ; 0 bytes in parameter list
default2          dc       i2'0'                     ; 0 bytes in default list

conf3             dc       i2'0'                     ; 0 bytes in parameter list
default3          dc       i2'0'                     ; 0 bytes in default list

conf4             dc       i2'0'                     ; 0 bytes in parameter list
default4          dc       i2'0'                     ; 0 bytes in default list

conf5             dc       i2'0'                     ; 0 bytes in parameter list
default5          dc       i2'0'                     ; 0 bytes in default list

conf6             dc       i2'0'                     ; 0 bytes in parameter list
default6          dc       i2'0'                     ; 0 bytes in default list

conf7             dc       i2'0'                     ; 0 bytes in parameter list
default7          dc       i2'0'                     ; 0 bytes in default list

conf8             dc       i2'0'                     ; 0 bytes in parameter list
default8          dc       i2'0'                     ; 0 bytes in default list

                  eject
***************************************************************
```

```
*
* The following are tables of format options for each device.
* The format option tables have the following structure:
*
*       Word    Number of entries in list
*       Word    Display count (number of head links)
*       Word    Recommended default option
*       Word    Option that current online media is formatted with
*       Entries 16 bytes per entry in the format list
*
* The twenty byte structure for each entry in the format list
* is as follows:
*
*       Word    Media variables reference number
*       Word    Link to reference number n.
*       Word    Flags / Format environment
*       Long    Number of blocks supported by device
*       Word    Block size
*       Word    Interleave factor
*       Long    Number of bytes defined by flag
*
* Bit definition within the flags word is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |___|__ Format
*   |   |   |   |   |   |   |   |   |   |   |   |   |___|_________ Flags
*   |___|___|___|___|___|___|___|___|___|___|___|_________________ Reserved
*
* Format Bit Definition: 00   Universal format
*       01   Apple Format
*       02   NonApple Format
*       11   Not Valid
*
* Flag Bit Definition:     00   Size is in bytes
*       01   Size is in Kb
*       02   Size is in Mb
*       11   Size is in Gb
*
*****************************************************************
format_tbl          entry
                    dc      i2'format1'         ; pointer to format option list #1
                    dc      i2'format2'         ; pointer to format option list #2
                    dc      i2'format3'         ; pointer to format option list #3
                    dc      i2'format4'         ; pointer to format option list #4
                    dc      i2'format5'         ; pointer to format option list #5
                    dc      i2'format6'         ; pointer to format option list #6
                    dc      i2'format7'         ; pointer to format option list #7
                    dc      i2'format8'         ; pointer to format option list #8

                    eject
format1             anop
```

```
                dc      i2'3'                   ; number of entries
                dc      i2'2'                   ; number of displayed entries
                dc      i2'1'                   ; recommended option is 1
                dc      i2'1'                   ; current media formatted w/option 1
format1_entry1  anop
                dc      i2'1'                   ; RefNum
                dc      i2'2'                   ; LinkRef
                dc      i2'4'                   ; univeral format / size in kb
                dc      i4'1600'                ; block count
                dc      i2'512'                 ; block size
                dc      i2'4'                   ; interleave factor
                dc      i2'800'                 ; media size is 800kb
format1_entry2  anop
                dc      i2'2'                   ; reference number 1
                dc      i2'0'                   ; LinkRef
                dc      i2'4'                   ; univeral format / size in kb
                dc      i4'1600'                ; block count
                dc      i2'512'                 ; block size
                dc      i2'2'                   ; interleave factor
                dc      i2'800'                 ; media size is 800kb
format1_entry3  anop
                dc      i2'3'                   ; reference number 1
                dc      i2'0'                   ; LinkRef
                dc      i2'4'                   ; univeral format / size in kb
                dc      i4'800'                 ; block count
                dc      i2'524'                 ; block size
                dc      i2'4'                   ; interleave factor
                dc      i2'400'                 ; media size is 400kb

                eject
format2         anop
                dc      i2'1'                   ; Number of entries
                dc      i2'1'                   ; number of displayed entries
                dc      i2'1'                   ; recommended option is 1
                dc      i2'1'                   ; current media formatted w/option 1
format2_entry1  anop
                dc      i2'1'                   ; RefNum
                dc      i2'0'                   ; LinkRef
                dc      i2'4'                   ; univeral format / size in kb
                dc      i4'280'                 ; block count
                dc      i2'512'                 ; block size
                dc      i2'0'                   ; interleave factor
                dc      i2'143'                 ; media size is 143 kb

format3         anop
                dc      i2'1'                   ; Number of entries
                dc      i2'1'                   ; number of displayed entries
                dc      i2'1'                   ; recommended option is 1
                dc      i2'1'                   ; current media formatted w/option 1
format3_entry1  anop
                dc      i2'1'                   ; RefNum
                dc      i2'0'                   ; LinkRef
                dc      i2'4'                   ; univeral format / size in kb
```

```
                dc      i4'280'             ; block count
                dc      i2'512'             ; block size
                dc      i2'0'               ; interleave factor
                dc      i2'143'             ; media size is 143 kb

format4         anop
                dc      i2'1'               ; Number of entries
                dc      i2'1'               ; number of displayed entries
                dc      i2'1'               ; recommended option is 1
                dc      i2'1'               ; current media formatted w/option 1
format4_entry1  anop
                dc      i2'1'               ; RefNum
                dc      i2'0'               ; LinkRef
                dc      i2'4'               ; univeral format / size in kb
                dc      i4'280'             ; block count
                dc      i2'512'             ; block size
                dc      i2'0'               ; interleave factor
                dc      i2'143'             ; media size is 143 kb

                eject
format5         anop
                dc      i2'1'               ; Number of entries
                dc      i2'1'               ; number of displayed entries
                dc      i2'1'               ; recommended option is 1
                dc      i2'1'               ; current media formatted w/option 1
format5_entry1  anop
                dc      i2'1'               ; RefNum
                dc      i2'0'               ; LinkRef
                dc      i2'4'               ; univeral format / size in kb
                dc      i4'280'             ; block count
                dc      i2'512'             ; block size
                dc      i2'0'               ; interleave factor
                dc      i2'143'             ; media size is 143 kb

format6         anop
                dc      i2'1'               ; Number of entries
                dc      i2'1'               ; number of displayed entries
                dc      i2'1'               ; recommended option is 1
                dc      i2'1'               ; current media formatted w/option 1
format6_entry1  anop
                dc      i2'1'               ; RefNum
                dc      i2'0'               ; LinkRef
                dc      i2'4'               ; univeral format / size in kb
                dc      i4'280'             ; block count
                dc      i2'512'             ; block size
                dc      i2'0'               ; interleave factor
                dc      i2'143'             ; media size is 143 kb

format7         anop
                dc      i2'1'               ; Number of entries
                dc      i2'1'               ; number of displayed entries
                dc      i2'1'               ; recommended option is 1
                dc      i2'1'               ; current media formatted w/option 1
```

```
format7_entry1  anop
                dc      i2'1'           ; RefNum
                dc      i2'0'           ; LinkRef
                dc      i2'4'           ; univeral format / size in kb
                dc      i4'280'         ; block count
                dc      i2'512'         ; block size
                dc      i2'0'           ; interleave factor
                dc      i2'143'         ; media size is 143 kb

format8         anop
                dc      i2'1'           ; Number of entries
                dc      i2'1'           ; number of displayed entries
                dc      i2'1'           ; recommended option is 1
                dc      i2'1'           ; current media formatted w/option 1
```

```
format8_entry1      anop
                    dc      i2'1'               ; RefNum
                    dc      i2'0'               ; LinkRef
                    dc      i2'4'               ; univeral format / size in kb
                    dc      i4'280'             ; block count
                    dc      i2'512'             ; block size
                    dc      i2'0'               ; interleave factor
                    dc      i2'143'             ; media size is 143 kb

                    eject
****************************************************************
*
*       Link Pointer
*       Entry Pointer
*       Device Characteristics
*     ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |___|_ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |_________ REMOVABLE
*   |   |   |   |   |   |   |   |   |   |   |   |   |_____________ FORMAT
*   |   |   |   |   |   |   |   |   |   |   |   |_________________ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |_____________________ READ
*   |   |   |   |   |   |   |   |   |   |_________________________ WRITE
*   |   |   |   |   |   |   |   |   |_____________________________ BLOCK DEVICE
*   |   |   |   |   |   |   |___|_________________________________ SPEED GROUP
*   |   |   |   |   |___|_________________________________________ RESERVED
*   |   |   |   |_________________________________________________ BUSY
*   |   |   |_____________________________________________________ LINKED
*   |   |_________________________________________________________ GENERATED
*   |_____________________________________________________________ RAM/ROM DEV
*
*       Block Count
*       Device Name
*       Slot Number
*       Unit Number
*       Device ID Number
*       Head Device Link
*       Forward Device Link
*       Reserved Word
*       Reserved Word
*       DIB device number
*
```

```
*****************************************************************
dib_1           entry
                dc      i4'dib_2'               ; link pointer to second DIB
                dc      i4'dispatch'            ; entry pointer
                dc      h'EC 00'                ; characteristics
                dc      i4'280'                 ; block count
                dc      i1'6'                   ; device name (length & 32 bytes ascii)
                dc      c'BLOCK0                          '
                dc      i2'0'                   ; slot # (valid only after startup)
                dc      i2'0'                   ; unit # (valid only after startup)
                dc      i2'current_ver'         ; version # 0001
                dc      i2'0'                   ; device ID # (valid only after startup)
                dc      i2'0'                   ; head device link
                dc      i2'0'                   ; forward device link
                dc      i4'0'                   ; extended DIB pointer
                dc      i2'0'                   ; dib device number
                dc      i2'0'                   ; drivers internal device number
                dc      i2'0'                   ; slot * 16

dib_2           entry
                dc      i4'dib_3'               ; link pointer
                dc      i4'dispatch'            ; entry pointer
                dc      h'EC 00'                ; characteristics
                dc      i4'280'                 ; block count
                dc      i1'6'                   ; device name (length & 32 bytes ascii)
                dc      c'BLOCK1                          '
                dc      i2'0'                   ; slot # (valid only after startup)
                dc      i2'0'                   ; unit # (valid only after startup)
                dc      i2'current_ver'         ; version # 0001
                dc      i2'0'                   ; device ID # (valid only after startup)
                dc      i2'0'                   ; head device link
                dc      i2'0'                   ; forward device link
                dc      i4'0'                   ; extended DIB pointer
                dc      i2'0'                   ; dib device number
                dc      i2'1'                   ; drivers internal device number
                dc      i2'0'                   ; slot * 16

                eject
```

```
dib_3           entry
                dc      i4'dib_4'           ; link pointer
                dc      i4'dispatch'        ; entry pointer
                dc      h'EC 00'            ; characteristics
                dc      i4'280'             ; block count
                dc      i1'6'               ; device name (length & 32 bytes ascii)
                dc      c'BLOCK2                      '
                dc      i2'0'               ; slot # (valid only after startup)
                dc      i2'0'               ; unit # (valid only after startup)
                dc      i2'current_ver'     ; version # 0001
                dc      i2'0'               ; device ID # (valid only after startup)
                dc      i2'0'               ; head device link
                dc      i2'0'               ; forward device link
                dc      i4'0'               ; extended DIB pointer
                dc      i2'0'               ; dib device number
                dc      i2'2'               ; drivers internal device number
                dc      i2'0'               ; slot * 16

dib_4           entry
                dc      i4'dib_5'           ; link pointer
                dc      i4'dispatch'        ; entry pointer
                dc      h'EC 00'            ; characteristics
                dc      i4'280'             ; block count
                dc      i1'6'               ; device name (length & 32 bytes ascii)
                dc      c'BLOCK3                      '
                dc      i2'0'               ; slot # (valid only after startup)
                dc      i2'0'               ; unit # (valid only after startup)
                dc      i2'current_ver'     ; version # 0001
                dc      i2'0'               ; device ID # (valid only after startup)
                dc      i2'0'               ; head device link
                dc      i2'0'               ; forward device link
                dc      i4'0'               ; extended DIB pointer
                dc      i2'0'               ; dib device number
                dc      i2'3'               ; drivers internal device number
                dc      i2'0'               ; slot * 16

dib_5           entry
                dc      i4'dib_6'           ; link pointer
                dc      i4'dispatch'        ; entry pointer
                dc      h'EC 00'            ; characteristics
                dc      i4'280'             ; block count
                dc      i1'6'               ; device name (length & 32 bytes ascii)
                dc      c'BLOCK4                      '
                dc      i2'0'               ; slot # (valid only after startup)
                dc      i2'0'               ; unit # (valid only after startup)
                dc      i2'current_ver'     ; version # 0001
                dc      i2'0'               ; device ID # (valid only after startup)
                dc      i2'0'               ; head device link
                dc      i2'0'               ; forward device link
                dc      i4'0'               ; extended DIB pointer
                dc      i2'0'               ; dib device number
                dc      i2'4'               ; drivers internal device number
                dc      i2'0'               ; slot * 16
```

```
                eject
dib_6           entry
                dc      i4'dib_7'           ; link pointer
                dc      i4'dispatch'        ; entry pointer
                dc      h'EC 00'            ; characteristics
                dc      i4'280'             ; block count
                dc      i1'6'               ; device name (length & 32 bytes ascii)
                dc      c'BLOCK5                    '
                dc      i2'0'               ; slot # (valid only after startup)
                dc      i2'0'               ; unit # (valid only after startup)
                dc      i2'current_ver'     ; version # 0001
                dc      i2'0'               ; device ID # (valid only after startup)
                dc      i2'0'               ; head device link
                dc      i2'0'               ; forward device link
                dc      i4'0'               ; extended DIB pointer
                dc      i2'0'               ; dib device number
                dc      i2'5'               ; drivers internal device number
                dc      i2'0'               ; slot * 16

dib_7           entry
                dc      i4'dib_8'           ; link pointer
                dc      i4'dispatch'        ; entry pointer
                dc      h'EC 00'            ; characteristics
                dc      i4'280'             ; block count
                dc      i1'6'               ; device name (length & 32 bytes ascii)
                dc      c'BLOCK6                    '
                dc      i2'0'               ; slot # (valid only after startup)
                dc      i2'0'               ; unit # (valid only after startup)
                dc      i2'current_ver'     ; version # 0001
                dc      i2'0'               ; device ID # (valid only after startup)
                dc      i2'0'               ; head device link
                dc      i2'0'               ; forward device link
                dc      i4'0'               ; extended DIB pointer
                dc      i2'0'               ; dib device number
                dc      i2'6'               ; drivers internal device number
                dc      i2'0'               ; slot * 16
```

```
dib_8           entry
                dc      i4'0'               ; link pointer
                dc      i4'dispatch'        ; entry pointer
                dc      h'EC 00'            ; characteristics
                dc      i4'280'             ; block count
                dc      i1'6'               ; device name (length & 32 bytes ascii)
                dc      c'BLOCK7                          '
                dc      i2'0'               ; slot # (valid only after startup)
                dc      i2'0'               ; unit # (valid only after startup)
                dc      i2'current_ver'     ; version # 0001
                dc      i2'0'               ; device ID # (valid only after startup)
                dc      i2'0'               ; head device link
                dc      i2'0'               ; forward device link
                dc      i4'0'               ; extended DIB pointer
                dc      i2'0'               ; dib device number
                dc      i2'7'               ; drivers internal device number
                dc      i2'0'               ; slot * 16

                eject
**************************************************************
*
* The following table is used to dispatch to GS/OS driver
* functions.
*
**************************************************************
dispatch_table  entry
                dc      i2'startup-1'
                dc      i2'open-1'
                dc      i2'read-1'
                dc      i2'write-1'
                dc      i2'close-1'
                dc      i2'status-1'
                dc      i2'control-1'
                dc      i2'flush-1'
                dc      i2'shutdn-1'
```

```
status_table        entry
                    dc      i2'dev_stat-1'
                    dc      i2'get_conf-1'
                    dc      i2'get_wait-1'
                    dc      i2'get_format-1'
                    dc      i2'get_partn_map-1'

control_table       entry
                    dc      i2'dev_reset-1'
                    dc      i2'format-1'
                    dc      i2'media_eject-1'
                    dc      i2'set_conf-1'
                    dc      i2'set_wait-1'
                    dc      i2'set_format-1'
                    dc      i2'set_partn-1'
                    dc      i2'arm_signal-1'
                    dc      i2'disarm_signal-1'
                    dc      i2'set_partn_map-1'

status_flag         entry
                    dc      i2'0'                   ; flag for unit #
                    dc      i2'0'                   ; flag for unit #
                    dc      i2'0'                   ; flag for unit #
                    dc      i2'0'                   ; flag for unit #
                    dc      i2'0'                   ; flag for unit #
                    dc      i2'0'                   ; flag for unit #
                    dc      i2'0'                   ; flag for unit #
                    dc      i2'0'                   ; flag for unit #

                    eject
*****************************************************************
*
* The following table contains the open status for each device
* supported by this driver.
*
*****************************************************************
open_table          entry
                    dc      i2'0'                   ; open state for DIB 1 device
                    dc      i2'0'                   ; open state for DIB 2 device
                    dc      i2'0'                   ; open state for DIB 3 device
                    dc      i2'0'                   ; open state for DIB 4 device
                    dc      i2'0'                   ; open state for DIB 5 device
                    dc      i2'0'                   ; open state for DIB 6 device
                    dc      i2'0'                   ; open state for DIB 7 device
                    dc      i2'0'                   ; open state for DIB 8 device

                    eject
```

```
*****************************************************************
*
* The following table contains the device status for each
* device supported by this driver.
*
* Encoding of status for a character device is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |__ OPEN
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |______ INTERRUPT
*   |   |   |   |   |   |   |   |   |   |   |   |   |___|__________ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |__________________ BUSY
*   |___|___|___|___|___|___|___|___|___|___|______________________ 0 RESERVED
*
* Encoding of status for a block device is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |__ DISK SW
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |______ INTERRUPT
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |__________ WRITE PROT
*   |   |   |   |   |   |   |   |   |   |   |   |   |______________ 0 RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |__________________ ONLINE
*   |___|___|___|___|___|___|___|___|___|___|______________________ 0 RESERVED
*
*****************************************************************
dstat_tbl       entry
                dc      i2'dstat1'          ; pointer to status for DIB 1 device
                dc      i2'dstat2'          ; pointer to status for DIB 2 device
                dc      i2'dstat3'          ; pointer to status for DIB 3 device
                dc      i2'dstat4'          ; pointer to status for DIB 4 device
                dc      i2'dstat5'          ; pointer to status for DIB 5 device
                dc      i2'dstat6'          ; pointer to status for DIB 6 device
                dc      i2'dstat7'          ; pointer to status for DIB 7 device
                dc      i2'dstat8'          ; pointer to status for DIB 8 device

dstat1          dc      h'00 00'            ; device general status word
                dc      i4'1600'            ; device block count

dstat2          dc      h'00 00'            ; device general status word
                dc      i4'280'             ; device block count

dstat3          dc      h'00 00'            ; device general status word
                dc      i4'280'             ; device block count
```

```
dstat4          dc      h'00 00'            ; device general status word
                dc      i4'280'             ; device block count

dstat5          dc      h'00 00'            ; device general status word
                dc      i4'280'             ; device block count

dstat6          dc      h'00 00'            ; device general status word
                dc      i4'280'             ; device block count

dstat7          dc      h'00 00'            ; device general status word
                dc      i4'280'             ; device block count

dstat8          dc      h'00 00'            ; device general status word
                dc      i4'280'             ; device block count

                eject
*****************************************************************
*
* The following table is used to return the configuration list for
* each device supported by this driver.
*
*****************************************************************
clist_tbl       entry
                dc      i2'conf1'           ; pointer to configuration list #1
                dc      i2'conf2'           ; pointer to configuration list #2
                dc      i2'conf3'           ; pointer to configuration list #3
                dc      i2'conf4'           ; pointer to configuration list #4
                dc      i2'conf5'           ; pointer to configuration list #5
                dc      i2'conf6'           ; pointer to configuration list #6
                dc      i2'conf7'           ; pointer to configuration list #7
                dc      i2'conf8'           ; pointer to configuration list #8

*****************************************************************
*
* The following table is used to return the wait mode for
* each device supported by this driver.
*
*****************************************************************
wait_mode_tbl   entry
                dc      i2'0'               ; unit 1 wait mode
                dc      i2'0'               ; unit 2 wait mode
                dc      i2'0'               ; unit 3 wait mode
                dc      i2'0'               ; unit 4 wait mode
                dc      i2'0'               ; unit 5 wait mode
                dc      i2'0'               ; unit 6 wait mode
                dc      i2'0'               ; unit 7 wait mode
                dc      i2'0'               ; unit 8 wait mode
```

```
****************************************************************
*
* The following table is used to set the current format
* option for each device supported by this driver.
*
****************************************************************
format_mode         entry
                    dc      i2'0'                  ; unit 1 format mode
                    dc      i2'0'                  ; unit 2 format mode
                    dc      i2'0'                  ; unit 3 format mode
                    dc      i2'0'                  ; unit 4 format mode
                    dc      i2'0'                  ; unit 5 format mode
                    dc      i2'0'                  ; unit 6 format mode
                    dc      i2'0'                  ; unit 7 format mode
                    dc      i2'0'                  ; unit 8 format mode

****************************************************************
*
* The following table is used by the startup call when setting
* parameters in the DIB.  Slot number, Unit number and Device
* ID number are valid only after startup.
*
****************************************************************
startup_slot        dc      i2'$000F'              ; initial slot to search for
startup_unit        dc      i2'1'                  ; initial unit to search for
startup_name        dc      h'31 20'               ; startup with BLOCK1

****************************************************************
*
* The following equates are general workspace used by the driver.
*
****************************************************************
retry_count         dc      i2'0'                  ; retry count
startup_count       dc      i2'0'

                    end

                    eject
```

```
***************************************************************
*
* DRIVER MAIN ENTRY POINT: DISPATCH
*
* This is the main entry point for the device driver.  The
* routine validates the call number prior to dispatching to
* the requested function.
*
* Call Number:        $0000    Function:   Startup
*                     $0001                Open
*                     $0002                Read
*                     $0003                Write
*                     $0004                Close
*                     $0005                Status
*                     $0006                Control
*                     $0007                Flush
*                     $0008                Shutdown
*                     $0009-$FFFF          Reserved
*
* ENTRY: Call via 'JSL'
*        [<drvr_dib_ptr] = Points to the DIB for the device being accessed
*        <drvr_dev_num = Device number of device being accessed
*        <drvr_call_num = Call number
*                     A Reg = Call Number
*                     X Reg = Undefined
*                     Y Reg = Undefined
*                     Dir Reg = GS/OS Direct Page
*                     B Reg = Undefined
*                     P Reg = N V M X D I Z C  E
*                             x x 0 0 0 0 x x  0
*
* EXIT:   Direct page = unchanged with the exception of <drvr_tran_cnt
*                     A Reg = Error code
*                     X Reg = Undefined
*                     Y Reg = Undefined
*                     Dir Reg = GS/OS Direct Page
*                     B Reg = Same as entry
*                     P Reg = N V M X D I Z C  E
*                             x x 0 0 0 0 x 0  0   No error occurred
*                             x x 0 0 0 0 x 1  0   Error occurred
*
```

```
*****************************************************************
dispatch        start
                using     driver_data
                longa     on
                longi     on

                phb                             ; save environment
                phk
                plb
                cmp       #max_command          ; is it a legal command?
                bge       illegal_req           ; no
                tay                             ; save command #
                ldx       #$0000
save_parms      anop
                lda       <drvr_dev_num,x       ; save GS/OS call parameters
                pha
                lda       <drvr_blk_num,x
                pha
                inx
                inx
                cpx       #$000C                ; up to but not including DRVR_TRAN_CNT
                bne       save_parms

                tya                             ; restore command #
                pea       func_ret-1            ; return address from function
                asl       a                     ; make index to dispatch table
                tax
                lda       |dispatch_table,x
                pha                             ; push function address for dispatch
                rts                             ; rts dispatches to function
func_ret        anop
                tay                             ; save error code

                ldx       #$000A                ; number of words to restore
restore_parms   anop
                pla                             ; restore GS/OS call parameters
                sta       <drvr_blk_num,x
                pla
                sta       <drvr_dev_num,x
                dex
                dex
                bpl       restore_parms
                plb
                bcs       gen_exit              ; force error code 0 if flag cleared
                ldy       #no_error
gen_exit        anop
                tya                             ; restore error code
                rtl
*
* Received an illegal request.  Return with an error.
*
```

```
illegal_req         anop
                    plb                                 ; restore environement
                    lda         #drvr_bad_req           ; set error
                    sec
                    rtl

                    end

                    eject
****************************************************************
*
* DRIVER CALL:    STARTUP
*
* This routine must prepare the driver to accept all other driver
* calls.
*
* ENTRY: Call via 'JSR'
*                   [<drvr_dib_ptr] = Points to the DIB for the device being accessed
*                   <drvr_dev_num = Device number of device being accessed
*                   <drvr_call_num = Call number
*                   <drvr_tran_cnt = $00000000
*                   A Reg = Call Number
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as program bank
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                   A Reg = Error code
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as entry
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x 0  0   No error occurred
*                           x x 0 0 0 0 x 1  0   Error occurred
*
```

```
*****************************************************************
startup          start
                 using    driver_data
                 longa    on
                 longi    on

                 lda      |startup_count          ; has slot been found?
                 beq      search_loop             ; no, go search for it
*
* Check for the device.
*
                 ; insert your code here.
                 bne      no_start_device         ; if you can't find your signatures bytes
                 inc      |startup_count
*
* Update the following DIB parameters...
*
*                Slot Number
*                Unit Number
*                Device Name                      (already unique for each DIB)
*
* Note that these DIB parameters are static after startup.
*
                 ldy      #slot_num               ; update DIB slot number
                 lda      |startup_slot
                 sta      [<drvr_dib_ptr],y
                 iny
                 iny
                 lda      |startup_count          ; update DIB unit number
                 sta      [<drvr_dib_ptr],y
                 bra      startup_done
*
* Always request the slot from the slot arbiter prior to scanning the $CnXX space
* for signature bytes when searching for hardware.  This provides a compatible
* method of requesting a slot should a method of dynamic slot switching be made
* available in the future.
*
search_loop
                 lda      |startup_slot           ; request slot from slot arbiter
                 jsl      dyn_slot_arbiter
                 bcs      next_slot               ; if slot was not granted
                 inc      |startup_count
*
* If the slot was granted then use the current slot to search for signature
* bytes identifying your hardware.
*
                 lda      |search_slot            ; create $Cn00 for signature search index
                 and      #$0007
                 ora      #$00C0
                 xba
                 tax      ; X register = $Cn00
*
* Now search for signatures.
```

```
*
                ; insert your code here.
                bne      next_slot              ; if you can't find your signatures bytes
*
* If you find your signature bytes then check for the device.
*
                ; insert your code here.
                bne      no_start_device        ; if you can't find your signatures bytes
*
* Update the following DIB parameters...
*
*               Slot Number
*               Unit Number
*               Device Name                     (already unique for each DIB)
*
* Note that these DIB parameters are static after startup.
*
                ldy      #slot_num              ; update DIB slot number
                lda      |startup_slot
                sta      [<drvr_dib_ptr],y
                iny
                iny
                lda      |startup_count         ; update DIB unit number
                sta      [<drvr_dib_ptr],y
startup_done
                lda      #no_error
                clc
                rts
next_slot
                dec      |startup_slot          ; point at next slot to check
                bne      search_loop            ; and check for hardware
no_start_device
                lda      #drvr_io_error
                sec
                rts


                eject
```

```
****************************************************************
*
* DRIVER CALL:     OPEN
*
* This call has no application with block device drivers and will
* return with no error.
*
* ENTRY: via a 'JSR'
*                <drvr_dev_num = Device Number of current device being accessed
*                <drvr_tran_cnt = $00000000
*                A Reg = Call Number
*                X Reg = Undefined
*                Y Reg = Undefined
*                Dir Reg = GS/OS Direct Page
*                B Reg = Undefined
*                P Reg = N V M X D I Z C  E
*                        x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                A Reg = Error code
*                X Reg = Undefined
*                Y Reg = Undefined
*                Dir Reg = GS/OS Direct Page
*                B Reg = Same as entry
*                P Reg = N V M X D I Z C  E
*                        x x 0 0 0 0 x 0  0   No error occurred
*                        x x 0 0 0 0 x 1  0   Error occurred
*
```

```
****************************************************************
open            start
                using   driver_data
                longa   on
                longi   on

                lda     #no_error
                clc
                rts

                end

                copy    core.blk.drvr/read
****************************************************************
*
* DRIVER CALL:    READ
*
* This call executes a read from the device.  Block devices must
* validate the initial block number and that the request count is
* an integral multiple of the block size.  In addition the block
* number of each successive block must be validated as it is
* accessed when an multiple block I/O transaction is in place.
*
* ENTRY: via a 'JSR'
*               <drvr_dev_num = Device Number of current device being accessed
*               <drvr_buf_ptr = Pointer to I/O buffer
*               <drvr_blk_num = Initial block number
*               <drvr_req_cnt = Number of bytes to be transferred
*               <drvr_blk_size = Size of block to be accessed
*               <drvr_tran_cnt = $00000000
*               A Reg = Call Number
*               X Reg = Undefined
*               Y Reg = Undefined
*               Dir Reg = GS/OS Direct Page
*               B Reg = Undefined
*               P Reg = N V M X D I Z C  E
*                       x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*               <drvr_tran_cnt = Number of bytes transferred
*               A Reg = Error code
*               X Reg = Undefined
*               Y Reg = Undefined
*               Dir Reg = GS/OS Direct Page
*               B Reg = Same as entry
*               P Reg = N V M X D I Z C  E
*                       x x 0 0 0 0 x 0  0  no error occurred
*                       x x 0 0 0 0 x 1  0  error occurred
*
```

```
****************************************************************
read            start
                using    driver_data
                longa    on
                longi    on

*
* The following routine implements the read call for a block device and
* includes cache support.
*
block_read      anop
                bit      <drvr_fst_num          ; is this a forced device access call?
                bmi      dev_access             ; yes
                clc                             ; specify cache block search
                jsl      cache_find_blk         ; is the block in the cache?
                bcs      not_cached             ; no
*
* The block is in the cache.  Use the system service call 'MOVE_INFO' to
* transfer the data from the cache to the buffer specified in the read call.
*
                pei      <drvr_cach_ptr+2
                pei      <drvr_cach_ptr
                pei      <drvr_buf_ptr+2
                pei      <drvr_buf_ptr
                pea      $0000
                pei      <drvr_blk_size
                pea      move_sinc_dinc
                jsl      move_info
                jsr      dev_stat               ; check for disk switched
                bcc      no_purge               ; if not switched
                brl      block_rd_err           ; else
no_purge        anop
*
```

```
* The block has been read.  Need to adjust the block address
* and buffer pointer in preperation for the next read if
* a multiple block transaction is in process.  If the
* request count has been satisfied, no further action is required.
*
                jsr     adj_buf_ptr             ; adjust new buffer address
                jsr     adjust_block            ; prepare for next block
                bcc     block_read              ; if more blocks to read
                lda     #no_error               ; else all done
                clc
                rts
*
* If block is not in cache or fst ID is negative then must force
* access to the device to read the block.
*
dev_access      anop
                ldy     #my_slot16              ; get slot index for hardware access
                lda     [<drvr_dib_ptr],y
                tax
                ldy     #0                      ; init buffer pointer

                sep     #$20                    ; 8 bit 'm'
                longa   off

read_loop0      anop
                lda     #100                    ; retry counter
                sta     |retry_count

retry_loop0     anop
                lda     >block_rdy              ; is data ready?
                bmi     read_ready0             ; yes
                dec     |retry_count
                bpl     retry_loop0             ; if retry count not exhausted
                sec                             ; else return offline error
                lda     #drvr_off_line
                rep     #$20                    ; 16 bit 'm'
                longa   on

                bra     block_rd_err

read_ready0     anop
                lda     >block_data             ; get data
                sta     [<drvr_buf_ptr],y
                iny
                cpy     <drvr_blk_size          ; read whole block?
                bne     read_loop0              ; no
                clc
```

```
read_loop_exit0 anop
                rep     #$20                ; 16 bit 'm'
                longa   on

                bcc     no_purge            ; if no I/O error
                lda     #drvr_io_error      ; else exit with error
                bcs     block_rd_err
*
* If block is not cached, action depends on the cache enable in the
* current call.  If the cache enable is zero then must force access
* to the device.  If cache enable is nonzero, a request for a block
* for the cache must be made of the cache manager.  If no block is
* granted then must force access to the device while reading only
* to the buffer.  If the block si granted, the block is read from the
* device to the buffer and the cache simultaneously.
*
not_cached      anop
                lda     <drvr_cache         ; is caching requested?
                beq     dev_access          ; no
                jsl     cache_add_blk       ; request a cached block from cache mgr
                bcs     dev_access          ; if block not granted
*
* The block has been granted from the cache manager.  Optimize I/O
* by writing data to both the cache and buffer simultaneously.
*
                ldy     #my_slot16          ; get slot index for hardware access
                lda     [<drvr_dib_ptr],y
                tax
                ldy     #0                  ; init buffer pointer

                sep     #$20                ; 8 bit 'm'
                longa   off

read_loop1      anop
                lda     #100                ; retry counter
                sta     |retry_count

retry_loop1     anop
                lda     >block_rdy          ; is data ready?
                bmi     read_ready1         ; yes
                dec     |retry_count
                bpl     retry_loop1         ; if retry count not exhausted
                sec                         ; else return I/O error
                bra     read_loop_exit1

read_ready1     anop
                lda     >block_data         ; get data
                sta     [<drvr_buf_ptr],y
                sta     [<drvr_cach_ptr],y
                iny
                cpy     <drvr_blk_size      ; read whole block?
                bne     read_loop1          ; no
                clc
```

```
read_loop_exitl anop
                rep     #$20                    ; 16 bit 'm'
                longa   on

                bcc     no_purge                ; if no I/O error
                lda     #drvr_io_error          ; else exit with error
block_rd_err    anop
                rts

                eject
****************************************************************
*
* ADJUST BUFFER POINTER
*
* This call adjusts the buffer pointer by the block size
*
* ENTRY: via a 'JSR'
*               <drvr_buf_ptr = Pointer to I/O buffer
*               <drvr_blk_size = Size of block to be accessed
*               A Reg = Call Number
*               X Reg = Undefined
*               Y Reg = Undefined
*               Dir Reg = GS/OS Direct Page
*               B Reg = Undefined
*               P Reg = N V M X D I Z C  E
*                       x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*               <drvr_tran_cnt = Number of bytes transferred
*               A Reg = Error code
*               X Reg = Undefined
*               Y Reg = Undefined
*               Dir Reg = GS/OS Direct Page
*               B Reg = Same as entry
*               P Reg = N V M X D I Z C  E
*                       x x 0 0 0 0 x x  0
*
****************************************************************
adj_buf_ptr     entry
                longa   on
                longi   on

                clc                             ; prepare for add
                lda     <drvr_blk_size
                adc     <drvr_buf_ptr
                sta     <drvr_buf_ptr
                lda     #$0000
                adc     <drvr_buf_ptr+2
                sta     <drvr_buf_ptr+2
                rts

                eject
```

```
*****************************************************************
*
* ADJUST BLOCK ADDRESS
*
* This call sets the next block address and verifies that the
* block is valid for the device being accessed.
*
* ENTRY: via a 'JSR'
*                <drvr_dib_ptr = Pointer to DIB for current device
*                <drvr_blk_num = Current block number
*                A Reg = Call Number
*                X Reg = Undefined
*                Y Reg = Undefined
*                Dir Reg = GS/OS Direct Page
*                B Reg = Undefined
*                P Reg = N V M X D I Z C  E
*                        x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                <drvr_blk_num = New block number
*                A Reg = Error code
*                X Reg = Undefined
*                Y Reg = Undefined
*                      Dir Reg = GS/OS Direct Page
*                B Reg = Same as entry
*                P Reg = N V M X D I Z C  E
*                        x x 0 0 0 0 x 0  0  if next block is valid
*                        x x 0 0 0 0 x 1  0  if next block is invalid
*
```

```
****************************************************************
adjust_block      entry
                  longa     on
                  longi     on

                  inc       <drvr_blk_num          ; set next block address
                  bne       validate_blk
                  inc       <drvr_blk_num+2
validate_blk      anop
                  ldy       #blk_cnt+2
                  lda       [<drvr_dib_ptr],y
                  cmp       <drvr_blk_num+2
                  bge       valid_blk
                  dey
                  dey
                  lda       [<drvr_dib_ptr],y
                  cmp       <drvr_blk_num
                  bge       valid_blk
                  sec
                  rts
valid_blk         anop
                  clc
                  rts

                  end

Local   Symbols

t_block
                  000000 block_read              000034 dev_access
ached             00003F read_loop0              000080 read_loop1
                  0000A1 read_loop_exit1
ready1            000044 retry_loop0             000085 retry_loop1
ate_blk


                            copy           core.blk.drvr/write
```

```
****************************************************************
*
* DRIVER CALL:     WRITE
*
* This call executes a write to the device.  Block devices must
* validate the initial block number and that the request count is
* an integral multiple of the block size.  In addition the block
* number of each successive block must be validated as it is
* accessed when an multiple block I/O transaction is in place.
*
* ENTRY: via a 'JSR'
*                 <drvr_dev_num = Device Number of current device being accessed
*                 <drvr_buf_ptr = Pointer to I/O buffer
*                 <drvr_blk_num = Initial block number
*                 <drvr_req_cnt = Number of bytes to be transferred
*                 <drvr_blk_size = Size of block to be accessed
*                 <drvr_tran_cnt = $00000000
*                 A Reg = Call Number
*                 X Reg = Undefined
*                 Y Reg = Undefined
*                 Dir Reg = GS/OS Direct Page
*                 B Reg = Undefined
*                 P Reg = N V M X D I Z C  E
*                         x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*      <drvr_tran_cnt = Number of bytes transferred
*                 A Reg = Error code
*                 X Reg = Undefined
*                 Y Reg = Undefined
*                 Dir Reg = GS/OS Direct Page
*                 B Reg = Same as entry
*                 P Reg = N V M X D I Z C  E
*                         x x 0 0 x x x 0  0  no error occurred
*                         x x 0 0 x x x 1  0  error occurred
*
```

```
****************************************************************
write           start
                using   driver_data
                longa   on
                longi   on
*
* The following routine implements the write call for a block device and
* includes cache support.
*
block_write     anop
                clc                             ; specify cache block search
                jsl     cache_find_blk          ; is the block in the cache?
                bcc     write_cache             ; yes
                lda     <drvr_cache             ; is caching requested?
                beq     write_blk_dev           ; no
                jsl     cache_add_blk           ; else request a block from cache mgr
                bcs     write_blk_dev
write_cache     anop
                pei     <drvr_buf_ptr+2         ; source
                pei     <drvr_buf_ptr
                pei     <drvr_cach_ptr+2        ; destination
                pei     <drvr_cach_ptr
                pea     $0000                   ; block size
                pei     <drvr_blk_size
                pea     move_sinc_dinc          ; move mode
                jsl     move_info
                jsr     dev_stat                ; check for disk switched
                bcs     block_wr_err            ; if cant move data
                bit     <drvr_cache             ; is it in deferred mode?
                bmi     wr_deferred             ; yes, don't write to media
write_blk_dev   anop

                bcs     block_wr_err            ; if error writing to device
wr_deferred     anop
                jsr     adj_buf_ptr             ; set next buffer address
                jsr     adjust_block            ; set next block address
                bcc     block_write             ; if more blocks to write
                lda     #no_error               ; else exit with no error
                clc
block_wr_err    anop
                rts

                end


Local  Symbols

                000000 block_write  000030 wr_deferred
                000011 write_cache


                copy    core.blk.drvr/close
```

```
****************************************************************
*
* DRIVER CALL:     CLOSE
*
* This call has no application with block devices and will
* return with no error.
*
* ENTRY: via a 'JSR'
*                  <drvr_dev_num = Device Number of current device being accessed
*                  <drvr_tran_cnt = $00000000
*                  A Reg = Call Number
*                  X Reg = Undefined
*                  Y Reg = Undefined
*                  Dir Reg = GS/OS Direct Page
*                  B Reg = Undefined
*                  P Reg = N V M X D I Z C  E
*                          x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                  A Reg = Error code
*                  X Reg = Undefined
*                  Y Reg = Undefined
*                  Dir Reg = GS/OS Direct Page
*                  B Reg = Same as entry
*                  P Reg = N V M X D I Z C  E
*                          x x 0 0 0 0 x 0  0   No error occurred
*                          x x 0 0 0 0 x 1  0   Error occurred
*
****************************************************************
close              start
                   using     driver_data
                   longa     on
                   longi     on

                   lda       #no_error
                   clc
                   rts

                   end


                   copy      core.blk.drvr/status
```

```
***********************************************************
*
* DRIVER CALL:    STATUS
*
* This routine supports all the standard device status calls.
* Any status call which is able to detect an OFFLINE or DISK
* SWITCHED condition should call the system service routine
* SET_DISKSW.  OFFLINE and DISKSW are conditions and not errors
* when detected by a status call and should only be returned as
* conditions in the status list.
*
* Status Code:        $0000      Return Device Status
*                     $0001      Return Control Parameters
*                     $0002      Return Wait/No Wait Mode
*                     $0003      Return Format Options
*                     $0004      Return Partition Map
*
* ENTRY: via a 'JSR'
*                 <drvr_dev_num = Device Number of current device being accessed
*                 <drvr_clist_ptr = Pointer to control list
*                 <drvr_ctrl_code = Control code
*                 <drvr_req_cnt = Number of bytes to be transferred
*                 <drvr_tran_cnt = $00000000
*                 A Reg = Call Number
*                 X Reg = Undefined
*                 Y Reg = Undefined
*                 Dir Reg = GS/OS Direct Page
*                 B Reg = Undefined
*                 P Reg = N V M X D I Z C  E
*                         x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                 <drvr_tran_cnt = Number of bytes transferred
*                 A Reg = Error code
*                 X Reg = Undefined
*                 Y Reg = Undefined
*                 Dir Reg = GS/OS Direct Page
*                 B Reg = Same as entry
*                 P Reg = N V M X D I Z C  E
*                         x x 0 0 0 0 x 0  0   No error occurred
*                         x x 0 0 0 0 x 1  0   Error occurred
*
```

```
*****************************************************************
status          start
                using     driver_data
                longa     on
                longi     on

*
* Need to verify that the status code specifies a
* legal status request.
*
                lda       <drvr_stat_code       ; is this a legal status request?
                cmp       #$0004
                blt       legal_status          ; yes
                lda       #drvr_bad_code        ; else return 'BAD CODE' error
                rts
*
* It's a legal status.  Dispatch to the appropriate status routine.
*
legal_status    anop
                asl       a
                tax
                lda       |status_table,x
                pha
                rts                             ; dispatch is via an 'RTS'

                eject
```

```
****************************************************************
*
* The DEVICE STATUS call returns a status list that indicates
* specific status information regarding a character or block
* device and the total number of blocks supported by a block device.
*
* Status List Pointer:     Word             General status word
*                          Longword         Total number of blocks
*
* Encoding of status for a block device is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |__ DISK SW
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |_____ INTERRUPT
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |_________ WRITE PROT
*   |   |   |   |   |   |   |   |   |   |   |   |   |_____________ 0 RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |_________________ ONLINE
*   |   |   |___|___|___|___|___|___|___|___|___|_________________ 0 RESERVED
*   |   |_________________________________________________________ LINKED DEV
*   |_____________________________________________________________ UNVALIDATED
*
* Valid request counts versus returned status list for this status
* call are as follows:
*
* Request Count: $0002     Status List: General status word
* Request Count: $0006     Status List: General status word and block count
*
```

```
*****************************************************************
dev_stat        entry
                longa   on
                longi   on

                lda     #drvr_bad_parm          ; assume invalid request count
                ldx     <drvr_req_cnt+2         ; and validate request count
                bne     bad_dev_stat
                ldx     <drvr_req_cnt
                cpx     #$0002
                blt     bad_dev_stat
                cpx     #$0007
                blt     ok_dev_stat
bad_dev_stat    anop
                sec
                rts
*
* Request count is valid.  Deterime device status and if appropriate, the
* total number of blocks for the device and return them in the device
* status list.     You insert the code required for this operation.
*
ok_dev_stat     anop
                ldy     #driver_unit            ; internal device #
                lda     [<drvr_dib_ptr],y
                asl     a
                tax
                lda     |dstat_tbl,x            ; get pointer to device status list
                tax
                ldy     #$0000                  ; status list pointer
                sep     #$20                    ; 8 bit 'm'
                longa   off
copy_dstat      anop
                lda     |0,x                    ; copy device status list to slist_ptr
                sta     [<drvr_slist_ptr],y
                inx
                iny
                cpy     <drvr_req_cnt           ; copy = request count size
                bne     copy_dstat
                rep     #$20                    ; 16 bit 'm'
                longa   on
*
```

dstat1 dc ih'0000' general status
dc i4'9728' blocks

```
* After returning the device status list check for an OFFLINE
* or DISKSW state.  If either of these conditions exist then
* the driver must call SET_DISKSW via the system service call
* table.
*
                lda     [<drvr_slist_ptr]
                eor     #$0010                  ; convert online to offline
                and     #$0011                  ; offline or disk switched?
                beq     not_blk_stat            ; no
                jsl     set_disksw              ; else call system service
not_blk_stat    anop
                brl     set_xfer_cnt            ; update xfer count & exit

                eject
****************************************************************
*
* This call returns a byte count as the first word in the status
* list followed by the data from the control parameter list.
* The request count specifies how much data is to be returned
* from the list.  If the byte count is smaller than the request
* count then only the number of bytes specified by the byte
* count will be returned and the transfer count will indicate
* this.
*
* Status List:  Word    Number of bytes in control list (including byte count).
*               Data    Data from the control list (device specific).
*
* This call requires a minimum request count of $00000002 and
* a maximum request count of $0000FFFF.
*
```

```
*****************************************************************
get_ctrl        entry
                longa   on
                longi   on

                lda     #drvr_bad_parm      ; assume invalid request count
                ldx     <drvr_req_cnt+2     ; and validate request count
                bne     bad_get_ctrl
                ldx     <drvr_req_cnt
                cpx     #$0002
                bge     ok_get_ctrl
bad_get_ctrl    anop
                sec
                rts
*
* Request count is valid.  Return control list.
*
ok_get_ctrl     anop
                ldy     #driver_unit        ; internal device #
                lda     [<drvr_dib_ptr],y
                asl     a
                tax
                lda     |clist_tbl,x        ; get pointer to control list
                tax
                lda     |0,x                ; get length of control list
                beq     no_clist            ; if list has no content
                cmp     <drvr_req_cnt       ; is list shorter than request?
                bge     req_cnt_ok          ; no
                sta     <drvr_req_cnt       ; else modify request count
req_cnt_ok      anop
                ldy     #$0000              ; status list index
                sep     #$20                ; 8 bit 'm'
                longa   off
copy_clist      anop
                lda     |0,x                ; copy control list to slist_ptr
                sta     [<drvr_slist_ptr],y
                inx
                iny
                cpy     <drvr_req_cnt       ; copy = request count size
                bne     copy_clist
                rep     #$20                ; 16 bit 'm'
                longa   on
                brl     set_xfer_cnt

no_clist        anop
                sta     [<drvr_slist_ptr]
                lda     #$0002
                brl     set_xfer_cnt

                eject
```

```
*****************************************************************
*
* This routine returns the Wait/No Wait mode that the driver
* is currently operating in.  This is returned in a word
* parameter which indicates a request count of $0002.
*
*****************************************************************
get_wait          entry
                  longa    on
                  longi    on

                  lda      #drvr_bad_parm         ; assume invalid request count
                  ldx      <drvr_req_cnt+2        ; and validate request count
                  bne      bad_get_wait
                  ldx      <drvr_req_cnt
                  cpx      #$0002
                  beq      ok_get_wait
bad_get_wait      anop
                  sec
                  rts
*
* Request count is valid. Return the wait mode for this device.
*
ok_get_wait       anop
                  ldy      #unit_num
                  lda      [<drvr_dib_ptr],y
                  tax
                  dex
                  lda      |wait_mode_tbl,x
                  sta      [<drvr_slist_ptr]
                  brl      set_xfer_cnt

                  eject
```

```
******************************************************************
*
* This routine returns the format options for the device.
* Consult the driver specification for the format option list.
* This call requires a minimum request count of $00000002.  The
* maximum request count may exceed the size of the format list
* in which case the request count returned will indicate the
* size of the format list.
*
******************************************************************
get_format         entry
                   longa     on
                   longi     on

                   lda       #drvr_bad_parm          ; assume invalid request count
                   ldx       <drvr_req_cnt+2         ; and validate request count
                   bne       bad_get_format
                   ldx       <drvr_req_cnt
                   cpx       #$0002
                   bge       ok_get_format
bad_get_format     anop
                   sec
                   rts
*
```

```
* Request count is valid. Return the format options for this device.
*
ok_get_format   anop
                ldy     #driver_unit          ; internal device #
                lda     [<drvr_dib_ptr],y
                asl     a
                tax
                lda     |format_tbl,x         ; get pointer to format option list
                tax
                lda     |0,x                  ; get # entries in option list
                asl     a                     ; list length = (n * 16) + 8
                asl     a
                asl     a
                asl     a
                clc
                adc     #$0008                ; now have option list length
                cmp     <drvr_req_cnt         ; is request longer then list length?
                bge     req_count_ok
                sta     <drvr_req_cnt
req_count_ok    anop
                ldy     #$0000                ; status list index
                sep     #$20                  ; 8 bit 'm'
                longa   off
copy_format     anop
                lda     |0,x                  ; copy format option list to slist_ptr
                sta     [<drvr_slist_ptr],y
                inx
                iny
                cpy     <drvr_req_cnt         ; copy = request count size
                bne     copy_format
                rep     #$20                  ; 16 bit 'm'
                longa   on
                brl     set_xfer_cnt

****************************************************************
*
* Get Partion Map.
*
* Normally this call would return a partition map for the device.
* Our example does not support partitioning and will return with
* no error and a transfer count of NIL.
*
****************************************************************
get_partn_map   entry
                longa   on
                longi   on

                lda     #no_error
                clc
                rts

                eject
```

```
**************************************************************
*
* This is a common exit routine for successful status calls.
* The transfer count is set to the same value as the request
* count prior to returning with no error.
*
**************************************************************
set_xfer_cnt        entry
                    lda       <drvr_req_cnt         ; set transfer count
                    sta       <drvr_tran_cnt
                    lda       <drvr_req_cnt+2
                    sta       <drvr_tran_cnt+2
                    lda       #no_error
                    clc
                    rts

                    end

                    copy      core.blk.drvr/control
**************************************************************
*
* DRIVER CALL:      CONTROL
*
* This routine supports all the standard device control calls.
*
* Control Code:         $0000       Reset Device
*                       $0001       Format Device
*                       $0002       Eject Media
*                       $0003       Set Control Parameters
*                       $0004       Set Wait/No Wait Mode
*                       $0005       Set Format Options
*                       $0006       Assign Partition Owner
*                       $0007       Arm Signal
*                       $0008       Disarm Signal
*                       $0009       Set Partition Map
*
```

```
* ENTRY: via a 'JSR'
*                <drvr_dev_num = Device Number of current device being accessed
*                <drvr_clist_ptr = Pointer to control list
*                <drvr_ctrl_code = Control code
*                <drvr_req_cnt = Number of bytes to be transferred
*                <drvr_tran_cnt = $00000000
*                A Reg = Call Number
*                X Reg = Undefined
*                Y Reg = Undefined
*                Dir Reg = GS/OS Direct Page
*                B Reg = Undefined
*                P Reg = N V M X D I Z C  E
*                        x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                <drvr_tran_cnt = Number of bytes transferred
*                A Reg = Error code
*                X Reg = Undefined
*                Y Reg = Undefined
*                Dir Reg = GS/OS Direct Page
*                B Reg = Same as entry
*                P Reg = N V M X D I Z C  E
*                        x x 0 0 0 0 x 0  0   No error occurred
*                        x x 0 0 0 0 x 1  0   Error occurred
*
*****************************************************************
control          start
                 using    driver_data
                 longa    on
                 longi    on

*
* Need to verify that the control code specifies a
* legal control request.
*
                 lda      <drvr_ctrl_code       ; is this a legal control request?
                 cmp      #$0009
                 blt      legal_control         ; yes
                 lda      #drvr_bad_code        ; else return 'BAD CODE' error
                 rts
*
* It's a legal control.  Dispatch to the appropriate control routine.
*
legal_control    anop
                 asl      a
                 tax
                 lda      |control_table,x
                 pha
                 rts                            ; dispatch is via an 'RTS'

                 eject
```

```
****************************************************************
*
* This routine will reset the device to it's default conditions
* as specified by the default control parameter list.  The
* control list contents will be updated to reflect the parameter
* changes that have taken effect.
*
* CONTROL LIST: None
*
****************************************************************
dev_reset           entry
                    lda        #no_error
                    clc
                    rts

                    eject
****************************************************************
*
* This routine will physically format the media.  No additional
* information associated with any particular file system will
* be written to the media. Check task count for disk switch
* prior to  execution of read.
*
* CONTROL LIST: None
*
****************************************************************
format              entry
                    lda        #no_error
                    clc
                    rts

                    eject
****************************************************************
*
* This routine will physically eject media from the device.
* Character devices will not perform any action as a result of
* this call.
*
* CONTROL LIST: None
*
****************************************************************
media_eject         entry
                    lda        #no_error
                    clc
                    rts

                    eject
****************************************************************
```

```
*
* This routine will set the configuration parameter list as specified
* by the contents of the configuration list.  Note that the first
* word of the configuration list must have the same value as the
* current configuration parameter list.
*
* CONTROL LIST:   Word      Size of configuration parameter list
*                 Data      Configuration parameter list
*
*****************************************************************
set_ctrl          entry
                  longa     on
                  longi     on

                  lda       #drvr_bad_parm          ; assume invalid request count
                  ldx       <drvr_req_cnt+2 ; and validate request count
                  bne       bad_set_ctrl
                  ldx       <drvr_req_cnt
                  cpx       #$0002
                  bge       ok_set_ctrl
bad_set_ctrl      anop
                  sec
                  rts
*
* Request count is valid.  Set configuration list.
*
ok_set_ctrl       anop
                  ldy       #driver_unit            ; internal device #
                  lda       [<drvr_dib_ptr],y
                  asl       a
                  tax
                  lda       |clist_tbl,x            ; get pointer to configuration list
                  tax
                  lda       |0,x                    ; are lengths the same?
                  cmp       [<drvr_clist_ptr]
                  beq       req_cnt_ok              ; yes
                  lda       #drvr_bad_parm          ; else return an error
                  sec
                  rts
req_cnt_ok        anop
                  ldy       #$0000                  ; status list index
                  sep       #$20                    ; 8 bit 'm'
                  longa     off
```

```
copy_clist      anop                            ; set new configuration list
                lda     [<drvr_clist_ptr],y
                sta     |0,x
                inx
                iny
                tya
                cmp     [<drvr_clist_ptr]
                bne     copy_clist
                rep     #$20                    ; 16 bit 'm'
                longa   on
                brl     set_xfer_cnt

                eject
***************************************************************
*
* This routine will set the WAIT/NO WAIT mode as specified
* by the contents of the control list.          Note that a device
* may not support no wait mode and should return a bad parameter
* error if this support is not provided.
*
* CONTROL LIST: Word        Wait / No Wait Mode
*
***************************************************************
set_wait        entry
                longa   on
                longi   on

                lda     #drvr_bad_parm          ; assume invalid request count
                ldx     <drvr_req_cnt+2         ; and validate request count
                bne     bad_set_wait
                ldx     <drvr_req_cnt
                cpx     #$0002
                beq     ok_set_wait
bad_set_wait    anop
                sec
                rts
*
* Request count is valid. Set the wait mode for this device.
*
ok_set_wait     anop
                ldy     #driver_unit
                lda     [<drvr_dib_ptr],y
                tax
                lda     [<drvr_slist_ptr]
                sta     |wait_mode_tbl,x
                brl     set_xfer_cnt

                eject
```

```
****************************************************************
*
* This routine will set the format option as specified
* by the contents of the control list.
*
* CONTROL LIST: Word        Format Option Referenc Number
*
****************************************************************
set_format      entry
                longa     on
                longi     on

                lda       #drvr_bad_parm        ; assume invalid request count
                ldx       <drvr_req_cnt+2       ; and validate request count
                bne       bad_fmt_opt
                ldx       <drvr_req_cnt
                cpx       #$0002
                beq       ok_set_format
bad_fmt_opt     anop
                sec
                rts
*
* Request count is valid. Set the format option for this device.
*
ok_set_format   anop
                ldy       #driver_unit
                lda       [<drvr_dib_ptr],y
                tax
                lda       [<drvr_slist_ptr]
                sta       |format_mode,x
                brl       set_xfer_cnt

                eject
****************************************************************
*
* This routine will set the partition owner as specified
* by the contents of the control list. Note that this call
* is only supported by partitioned devices such as CD ROM.
* Non partitioned devices should perform no action and return
* with no error.
*
* CONTROL LIST:   Word      String length
*                 Name      Name of partition owner
*
```

```
****************************************************************
set_partn          entry
                   longa     on
                   longi     on

                   lda       #no_error
                   clc
                   rts

                   eject
****************************************************************
*
* This routine is envoked by an application to install a signal
* into the event mechanism.
*
* CONTROL LIST:    Word      Signal Code
*                  Word      Signal Priority
*                  Long      Signal Handler Address
*
****************************************************************
arm_signal         entry
                   longa     on
                   longi     on

                   lda       #no_error
                   clc
                   rts

****************************************************************
*
* This routine is remove a signal from the event mechanism that
* was previously installed with the arm_signal call.
*
* CONTROL LIST:    Word      Signal Code
*
****************************************************************
disarm_signal        entry
                     longa     on
                     longi     on

                     lda       #no_error
                     clc
                     rts

****************************************************************
*
* Set Partion Map.
*
* Normally this call would set a partition map for the device.
* Our example does not support partitioning and will return with
* no error and a transfer count of NIL.
*
****************************************************************
```

```
set_partn_map       entry
                    longa      on
                    longi      on

                    lda        #no_error
                    clc
                    rts

                    end


Local  Symbols

mt_opt             00002F bad_set_ctrl
                   00004D copy_clist   000012 dev_reset
                   000017 format
                   00001C media_eject  000031 ok_set_ctrl
                   00006E ok_set_wait  000048 req_cnt_ok
ormat              00009A set_partn
                   00005E set_wait


                   copy       core.blk.drvr/flush
```

```
*****************************************************************
*
* DRIVER CALL:    FLUSH
*
* This call writes any data in the devices internal buffer to
* the device.  It should be noted that this is a WAIT MODE call
* which is only supported by devices which maintain their own
* internal I/O buffer.devices that cannot write in NO WAIT mode
* do not support this call and will return with no error.
*
* ENTRY: via a 'JSR'
*                 <drvr_dev_num = Device Number of current device being accessed
*                 <drvr_tran_cnt = $00000000
*                 A Reg = Call Number
*                 X Reg = Undefined
*                 Y Reg = Undefined
*                 Dir Reg = GS/OS Direct Page
*                 B Reg = Undefined
*                 P Reg = N V M X D I Z C  E
*                         x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                 <drvr_tran_cnt = Number of bytes transferred
*                 A Reg = Error code
*                 X Reg = Undefined
*                 Y Reg = Undefined
*                 Dir Reg = GS/OS Direct Page
*                 B Reg = Same as entry
*                 P Reg = N V M X D I Z C  E
*                         x x 0 0 0 0 x 0  0   No error occurred
*                         x x 0 0 0 0 x 1  0   Error occurred
*
*****************************************************************
flush           start
                using     driver_data
                longa     on
                longi     on

                lda       #no_error
                clc
                rts

                end


                copy      core.blk.drvr/shutdown
*****************************************************************
```

```
*
* DRIVER CALL:    SHUTDOWN
*
* This call prepares the driver for shutdown.  This may include
* closing a character device as well as releasing any and all
* system resources that may have been aquired by either a
* STARTUP or OPEN call.  Many devices may share a common code segment.
* If this is the case, an error should be returned on shutdown from all
* but the last code segment.  The device dispatcher will free up the
* memory occupied by the driver when no error is returned on shutdown.
*
* ENTRY: via a 'JSR'
*                 <drvr_dev_num = Device Number of current device being accessed
*                 <drvr_tran_cnt = $00000000
*                 A Reg = Call Number
*                 X Reg = Undefined
*                 Y Reg = Undefined
*                 Dir Reg = GS/OS Direct Page
*                 B Reg = Undefined
*                 P Reg = N V M X D I Z C  E
*                         x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                 <drvr_tran_cnt = Number of bytes transferred
*                 A Reg = Error code
*                 X Reg = Undefined
*                 Y Reg = Undefined
*                 Dir Reg = GS/OS Direct Page
*                 B Reg = Same as entry
*                 P Reg = N V M X D I Z C  E
*                         x x 0 0 0 0 x 0  0   No error occurred
*                         x x 0 0 0 0 x 1  0   Error occurred
*
*****************************************************************
shutdn          start
                using   driver_data
                longa   on
                longi   on

                dec     |startup_count          ; is this the last device shutdown?
                bne     not_last                ; no
                lda     #no_error               ; else return no error on last device
                clc
                rts

not_last        anop
                lda     #drvr_busy              ; return an error if not last
                sec
                rts

                end
```

Character driver

This is a typical driver for a character device such as a serial printer. It includes handlers for all standard driver calls, although in this example not all of the handlers are functional. The driver code consists of seven parts, in this order:

- Equates
- Device-driver header
- Configuration parameter lists (3 of them, for 3 supported devices)
- Device information blocks (DIBs; 3 of them)
- Tables for dispatching calls and passing parameters
- A main entry point to the driver
- Routines that handle the driver calls

Like the block device driver listed earlier in this appendix, this driver has routines to handle all standard driver calls, including the standard Status and Control subcalls. Even though it is a character-device driver, for which several Control subcalls are not meaningful, handlers for all subcalls are included.

```
                65816           on
                instime         on
                gen   on
                symbol          on
                absaddr         on
                align           256
**************************************************************
*
*               Copyright (c) 1987, 1988
*               Apple Computer, Inc.
*               All rights reserved.
*
**************************************************************
*
*               Driver Core Routines Version 0.01a01
*
* NOTE:         All driver files must be installed on the
*               boot volume in the subdirectory "/SYSTEM/DRIVERS'.
*               Additionally, the FileType for the driver file
*               must be set to $00BB.  AuxType is also critical
*               to the operating system recognizing the driver
*               as a GS/OS device driver.  The AuxType is a long
*               word which must have the upper word set to $0000.
*               The most significant byte of the least significant
*               word in the AuxType must be set to $01 to indicate
*               an active GS/OS device driver or $81 to indicate
*               an inactive GS/OS device driver.  The least
*               significant byte of the least significant word
*               of the AuxType field indicates the number of
*               devices supported by the driver file.  This value
*               should be analogous to the number of DIB's
*               contained in the driver file.  GS/OS will only
*               install the number of devices indicated in the
*               AuxType field.
*
*               GS/OS Device Driver:        FileType = $00BB
*                                           AuxType  = $000001XX where:
*                                                   XX = number of devices.
*
*               An AuxType of $00000108 indicates eight devices.  When
*               building a device driver, the best way to set the
*               FileType and AuxType is to use the Exerciser to get
*               the current file info (GET_FILE_INFO), modify the
*               FileType & AuxType and then SET_FILE_INFO.
*
**************************************************************
*
* REVISION HISTORY:
*
* DATE      Ver.   By     Description
*
* 11/16/87 0.00e01 RBM    Started initial coding.
*
```

```
* 01/10/88 0.00e02 RBM      Added new status & control calls.
* 01/11/88                  Fixed startup for dynamic slot numbers.
*
* 02/04/88 0.00a01 RBM      General update for Alpha release.
*
* 02/12/88 0.01a01 RBM      Modified for character device support only.
*
* 04/11/88 0.06a01 RBM      Removed valid access checking done by dispatcher.
*                           Added status and control call support.
*
*****************************************************************


        eject
*****************************************************************
*
* The following are direct page equates on the GS/OS
* direct page for driver usage.
*
*****************************************************************

drvr_dev_num        gequ      $00               ; (w)  device number
drvr_call_num       gequ      drvr_dev_num+2    ; (w)  call number
drvr_buf_ptr        gequ      drvr_call_num+2   ; (lw) buffer pointer
drvr_slist_ptr      gequ      drvr_call_num+2   ; (lw) buffer pointer
drvr_clist_ptr      gequ      drvr_call_num+2   ; (lw) buffer pointer
dev_id_ref          gequ      drvr_buf_ptr      ; (w)  indirect device ID
drvr_req_cnt        gequ      drvr_buf_ptr+4    ; (lw) request count
drvr_tran_cnt       gequ      drvr_req_cnt+4    ; (lw) transfer count
drvr_blk_num        gequ      drvr_tran_cnt+4   ; (lw) block number
drvr_blk_size       gequ      drvr_blk_num+4    ; (w)  block size
drvr_fst_num        gequ      drvr_blk_size+2   ; (w)  File System Translator Number
drvr_stat_code      gequ      drvr_fst_num      ; (w)  status code for status call
drvr_ctrl_code      gequ      drvr_fst_num      ; (w)  control code for control call
drvr_vol_id         gequ      drvr_fst_num+2    ; (w)  Driver Volume ID Number
drvr_cache          gequ      drvr_vol_id+2     ; (w)  Cache Priority
drvr_cach_ptr       gequ      drvr_cache+2      ; (lw) pointer to cached block
drvr_dib_ptr        gequ      drvr_cach_ptr+4   ; (lw) pointer to active DIB

sib_ptr             gequ      $0074             ; (lw) pointer to active SIB
sup_parm_ptr        gequ      sib_ptr+4         ; (lw) pointer to supervisor parameters


        eject
*****************************************************************
*
* The following are equates for driver command types.
*
*****************************************************************

drvr_startup    gequ    $0000    ; driver startup command
drvr_open       gequ    $0001    ; driver open command
drvr_read       gequ    $0002    ; driver read command
```

```
drvr_write        gequ     $0003     ; driver write command
drvr_close        gequ     $0004     ; driver close command
drvr_status       gequ     $0005     ; driver status command
drvr_control      gequ     $0006     ; driver control command
drvr_flush        gequ     $0007     ; driver flush command
drvr_shutdn       gequ     $0008     ; driver shutdown command
max_command       gequ     $0009     ; commands $0009 - $ffff undefined

drvr_dev_stat     gequ     $0000     ; status code: return device status
drvr_conf_stat    gequ     $0001     ; status code: return configuration params
drvr_get_wait     gequ     $0002     ; status code: get wait/no wait mode
drvr_get_format   gequ     $0003     ; status code: get format options

drvr_reset        gequ     $0000     ; control code: reset device
drvr_format       gequ     $0001     ; control code: format device
drvr_eject        gequ     $0002     ; control code: eject media
drvr_set_conf     gequ     $0003     ; control code: set configuration params
drvr_set_wait     gequ     $0004     ; control code: set wait/no wait mode
drvr_set_format   gequ     $0005     ; control code: set format options
drvr_set_ptn      gequ     $0006     ; control code: set partition owner
drvr_arm          gequ     $0007     ; control code: arm interrupt signal
drvr_disarm       gequ     $0007     ; control code: arm interrupt signal

         eject
***************************************************************
*
* The following are equates for GS/OS error codes.
*
***************************************************************
no_error          gequ     $0000     ; no error has occurred
dev_not_found     gequ     $0010     ; device not found
invalid_dev_num   gequ     $0011     ; invalid device number
drvr_bad_req      gequ     $0020     ; bad request or command
drvr_bad_code     gequ     $0021     ; bad control or status code
drvr_bad_parm     gequ     $0022     ; bad call parameter
drvr_not_open     gequ     $0023     ; character device not open
drvr_prior_open   gequ     $0024     ; character device already open
irq_table_full    gequ     $0025     ; interrupt table full
drvr_no_resrc     gequ     $0026     ; resources not available
drvr_io_error     gequ     $0027     ; I/O error
drvr_no_dev       gequ     $0028     ; device not connected
drvr_busy         gequ     $0029     ; call aborted, driver is busy
drvr_wr_prot      gequ     $002B     ; device is write protected
drvr_bad_count    gequ     $002C     ; invalid byte count
drvr_bad_block    gequ     $002D     ; invalid block address
drvr_disk_sw      gequ     $002E     ; disk has been switched
drvr_off_line     gequ     $002F     ; device off line / no media present
invalid_access    gequ     $004E     ; access not allowed
parm_range_err    gequ     $0053     ; parameter out of range
out_of_mem        gequ     $0054     ; out of memory
dup_volume        gequ     $0057     ; duplicate volume name
not_block_dev     gequ     $0058     ; not a block device
stack_overflow    gequ     $005F     ; too many applications on stack
```

```
data_unavail     gequ    $0060    ; data unavailable

        eject
*****************************************************************
*
* The following are equates for the DIB.
*
*****************************************************************
link_ptr         gequ    $0000    ; (lw) pointer to next DIB
entry_ptr        gequ    $0004    ; (lw) pointer to driver
dev_char         gequ    $0008    ; (w)  device characteristics
blk_cnt          gequ    $000A    ; (lw) number of blocks
dev_name         gequ    $000E    ; (32) count and ascii name (pstring)
slot_num         gequ    $002E    ; (w)  slot number
unit_num         gequ    $0030    ; (w)  unit number
ver_num          gequ    $0032    ; (w)  version number
dev_id_num       gequ    $0034    ; (w)  device ID number (ICON ref#)
head_link        gequ    $0036    ; (w)  backward device link
forward_link     gequ    $0038    ; (w)  forward device link
link_dib_ptr     gequ    $003A    ; (lw) dib reserved field #1
dib_dev_num      gequ    $003E    ; (w)  Device number of this device
*
* The following equate(s) are for drive specific extensions to the DIB.
* Parameters that are extended to the manditory DIB parameters are not
* accessable by GS/OS or the application but may be used within a driver
* as needed.
*
driver_unit      gequ    $0040    ; (w)  driver's internal DIB data
my_slot16        gequ    $0042    ; (w)  driver's slot * 16


        eject
*****************************************************************
*
* System Service Table Equates:
*
* NOTE: Only those system service calls that might be used
* by a device driver are listed here.  For a more complete
* list of system service calls and explanations of each call
* consult the system service call ERS.
*
*****************************************************************
dev_dispatcher   gequ    $01FC00          ; dev_dispatch
cache_find_blk   gequ    $01FC04          ; cash_find
cache_add_blk    gequ    $01FC08          ; cash_add
cache_del_blk    gequ    $01FC14          ; cash_delete
cache_del_vol    gequ    $01FC18          ; cash_del_vol
set_sys_speed    gequ    $01FC50          ; set system speed
move_info        gequ    $01FC70          ; gs_move_block
set_disksw       gequ    $01FC90          ; set disksw and call swapout/delvol
sup_drvr_disp    gequ    $01FCA4          ; supervisor dispatcher
install_driver   gequ    $01FCA8          ; dynamic driver installation
```

```
dyn_slot_arbiter  gequ     $01FCBC          ; dynamic slot arbiter

         eject
***************************************************************
*
* MOVE_INFO
*
* NOTE: The following equates are used to set the modes
* passed to the move_info call system service call.
*
***************************************************************
moveblkcmd         gequ     $0800    ; block move option
move_sinc_dinc     gequ     $0805    ; source increment, dest. increment

move_sinc_ddec     gequ     $0809    ; source increment, dest. decrement
move_sdec_dinc     gequ     $0806    ; source decrement, dest. increment
move_sdec_ddec     gequ     $080A    ; source decrement, dest. decrement

move_scon_dcon     gequ     $0800    ; source constant, dest. constant
move_sinc_dcon     gequ     $0801    ; source increment, dest. constant
move_sdec_dcon     gequ     $0802    ; source decrement, dest. constant
move_scon_dinc     gequ     $0804    ; source constant, dest. increment
move_scon_ddec     gequ     $0808    ; source constant, dest. decrement

         eject

; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;|slot7 |slot6 |slot5 |slot4 |      |slot2 |slot1 |      |
;|intext|intext|intext|intext|  0   |intext|intext|  0   |
;|enable|enable|enable|enable|      |enable|enable|      |
;|______|______|______|______|______|______|______|______|
;        ^^^^^  sltromsel byte ^^^^^
;
; sltromsel bits defined as follows
;          bit 7= 0 enables internal slot 7 -- 1 enables slot rom
;          bit 6= 0 enables internal slot 6 -- 1 enables slot rom
;          bit 5= 0 enables internal slot 5 -- 1 enables slot rom
;          bit 4= 0 enables internal slot 4 -- 1 enables slot rom
;          bit 3= must be 0
;          bit 2= 0 enables internal slot 2 -- 1 enables slot rom
;          bit 1= 0 enables internal slot 1 -- 1 enables slot rom
;          bit 0= must be 0

sltromsel          gequ     $00C02D          ;slot rom select
```

```
; __7_____6_____5_____4_____3_____2_____1_____0___
;|      |      |      |      |      |      |      |      |
;|      | stop |      | stop | stop | stop | stop | stop |
;|  0   |i/o/lc|  0   |auxh-r|suprhr|hires2|hires1|txt pg|
;|      |shadow|      |shadow|shadow|shadow|shadow|shadow|
;|______|______|______|______|______|______|______|______|
;        ^^^^^  shadow byte ^^^^^
;
; shadow bits defined as follows
;         bit 7= must write 0
;         bit 6= 1 to inhibit i/o and language card operation
;         bit 5= must write 0
;         bit 4= 1 to inhibit shadowing aux hi-res page
;         bit 3= 1 to inhibit shadowing 32k video buffer
;         bit 2= 1 to inhibit shadowing hires page 2
;         bit 1= 1 to inhibit shadowing hires page 1
;         bit 0= 1 to inhibit shadowing text pages

shadow   gequ     $00C035          ;shadow register

         eject

; __7_____6_____5_____4_____3_____2_____1_____0___
;|      |      |      |      |      |      |      |      |
;| slow/|      |      |shadow|slot 7|slot 6|slot 5|slot 4|
;| fast |  0   |  0   |in all|motor |motor | motor| motor|
;| speed|      |      | ram  |detect|detect|detect|detect|
;|______|______|______|______|______|______|______|______|
;        ^^^^^  cyareg byte ^^^^^
;
; cyareg bits defined as follows
;         bit 7= 0=slow system speed -- 1=fast system speed
;         bit 6= must write 0
;         bit 5= must write 0
;         bit 4= shadow in all ram banks
;         bit 3= slot 7 disk motor on detect
;         bit 2= slot 6 disk motor on detect
;         bit 1= slot 5 disk motor on detect
;         bit 0= slot 4 disk motor on detect

cyareg   gequ     $00C036          ;speed and motor on detect
```

```
; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;| alzp | page2| ramrd|ramwrt| rdrom|lcbnk2|rombnk| intcx|
;|status|status|status|status|status|status|status|status|
;|______|______|______|______|______|______|______|______|
;        ^^^^^  statereg status byte ^^^^^
;
; statereg bits defined as follows
;         bit 7= alzp status
;         bit 6= page2 status
;         bit 5= ramrd status
;         bit 4= ramwrt status
;         bit 3= rdrom status (read only ram/rom (0/1))
;
; important note:
;        do two reads to $c083 then change statereg
;        to change lcram/rom banks (0/1) and still
;        have the language card write enabled.
;
;         bit 2= lcbnk2 status 0=LC bank 0 - 1=LC bank 1
;         bit 1= rombank status
;         bit 0= intcxrom status

statereg          gequ     $00C068          ; state register

clrrom            gequ     $00CFFF          ; switch out $c8 roms


        eject
```

```
******************************************************************
*
* EQUATES for the IWM require index of (n*16)
*
******************************************************************
phaseoff        gequ    $00C080         ; stepper phase off.
phaseon         gequ    $00C081         ; stepper phase on.

ph0off          gequ    $00C080         ; phase 0 off
ph0on           gequ    $00C081         ; phase 0 on
ph1off          gequ    $00C082         ; phase 1 off
ph1on           gequ    $00C083         ; phase 1 on
ph2off          gequ    $00C084         ; phase 2 off
ph2on           gequ    $00C085         ; phase 2 on
ph3off          gequ    $00C086         ; phase 3 off
ph3on           gequ    $00C087         ; phase 3 on

motoroff        gequ    $00C088         ; disk motor off
motoron         gequ    $00C089         ; disk motor on

drv0en          gequ    $00C08A         ; select drive 0
drv1en          gequ    $00C08B         ; select drive 1

q6l             gequ    $00C08C         ; Q6 low
q6h             gequ    $00C08D         ; Q6 high
q7l             gequ    $00C08E         ; Q7 low
q7h             gequ    $00C08F         ; Q7 high

emulstack       gequ    $010100         ; emulation mode stack pointer

                eject
******************************************************************
*
* The following equates are used to implement our hypothetical
* device driver.  They in no way reflect softswitches associated
* with any real hardware device.
*
******************************************************************

*   ___ ___ ___ ___ ___ ___ ___ ___
*  |   |   |   |   |   |   |   |   |
*  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |          READY
*  |___|___|___|___|___|___|___|___|
*    |   |___|___|___|___|___|___|__ Reserved
*    |______________________________ 1 = Device is ready
*
ready    gequ     $00C080
```

```
*    ___ ___ ___ ___ ___ ___ ___ ___
*   |   |   |   |   |   |   |   |   |
*   | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |          CHAR
*   |___|___|___|___|___|___|___|___|
*     |___|___|___|___|___|___|___|__ Character device data register
*
char     gequ     $00C081

*    ___ ___ ___ ___ ___ ___ ___ ___
*   |   |   |   |   |   |   |   |   |
*   | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |          CHAR_STATUS
*   |___|___|___|___|___|___|___|___|
*     |   |   |   |   |   |   |   |__ 0
*     |   |   |   |   |   |   |______ 1 = Interrupt in process
*     |   |   |   |   |   |__________ 0
*     |   |   |   |   |______________ 0
*     |   |   |   |__________________ 1 = Online
*     |   |   |______________________ 0
*     |   |__________________________ 0
*     |______________________________ 0
*
char_status     gequ     $00C082

*    ___ ___ ___ ___ ___ ___ ___ ___
*   |   |   |   |   |   |   |   |   |
*   | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |          CHAR_CONTROL
*   |___|___|___|___|___|___|___|___|
*     |___|___|___|___|___|___|___|__ Character device control register
*
char_control     gequ     $00C083

         eject
*****************************************************************
*
* The following table is the header required for all loaded
* drivers which consists of the following:
*
*        Word     Offset from start to 1st DIB
*        Word     Number of DIBs
*        Word     Offset from start to 1st configuration list
*        Word     Offset from start to 2nd configuration list
*        etc.
*
```

```
****************************************************************
driver_data         data
here                entry
                    dc          i2'dib_1-here'      ; offset to 1st DIB
                    dc          i2'3'               ; number of devices
                    dc          i2'conf1-here'      ; offset to 1st configuration list
                    dc          i2'conf2-here'      ; offset to 2nd configuration list
                    dc          i2'conf3-here'      ; offset to 3rd configuration list
*
* The following are the driver configuration parameter lists.
*
conf1               dc          i2'0'               ; 0 bytes in parameter list
default1            dc          i2'0'               ; 0 bytes in default list

conf2               dc          i2'0'               ; 0 bytes in parameter list
default2            dc          i2'0'               ; 0 bytes in default list

conf3               dc          i2'0'               ; 0 bytes in parameter list
default3            dc          i2'0'               ; 0 bytes in default list

                    eject
****************************************************************
*
*       Link Pointer
*       Entry Pointer
*       Device Characteristics
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |_ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |_____ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |_________ REMOVABLE
*   |   |   |   |   |   |   |   |   |   |   |   |   |_____________ FORMAT
*   |   |   |   |   |   |   |   |   |   |   |   |_________________ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |_____________________ READ
*   |   |   |   |   |   |   |   |   |   |_________________________ WRITE
*   |   |   |   |   |   |   |   |   |_____________________________ BLOCK DEVICE
*   |   |   |   |   |   |   |   |_________________________________ SPEED (LSB)
*   |   |   |   |   |   |   |_____________________________________ SPEED (MSB)
*   |   |   |   |   |   |_________________________________________ RESERVED
*   |   |   |   |   |_____________________________________________ RESERVED
*   |   |   |   |_________________________________________________ BUSY
*   |   |   |_____________________________________________________ LINKED
*   |   |_________________________________________________________ GENERATED
*   |_____________________________________________________________ RAM/ROM DEV
*
*       Block Count
*       Device Name
*       Slot Number
*       Unit Number
*       Device ID Number
*       Head Device Link
```

```
*       Forward Device Link
*       Reserved Word
*       Reserved Word
*       DIB device number
*
*******************************************************************
dib_1   entry
        dc      i4'dib_2'          ; link pointer to second DIB
        dc      i4'dispatch'       ; entry pointer
        dc      h'60 03'           ; characteristics
        dc      i4'0'              ; block count
        dc      i1'10'             ; device name (length & 32 bytes ascii)
        dc      c'CHARACTER1                    '
        dc      i2'7'              ; slot # (valid only after startup)
        dc      i2'1'              ; unit # (valid only after startup)
        dc      i2'1'              ; version # 0001
        dc      h'16 00'           ; device ID # (valid only after startup)
        dc      i2'0'              ; head device link
        dc      i2'0'              ; forward device link
        dc      i2'0'              ; Reserved
        dc      i2'0'              ; Reserved
        dc      i2'0'              ; dib device number
        dc      i2'0'              ; drivers internal device number
        dc      i2'0'              ; slot * 16

dib_2   entry
        dc      i4'dib_3'          ; link pointer
        dc      i4'dispatch'       ; entry pointer
        dc      h'60 03'           ; characteristics
        dc      i4'0'              ; block count
        dc      i1'10'             ; device name (length & 32 bytes ascii)
        dc      c'CHARACTER2                    '
        dc      i2'2'              ; slot # (valid only after startup)
        dc      i2'1'              ; unit # (valid only after startup)
        dc      i2'1'              ; version # 0001
        dc      h'16 00'           ; device ID # (valid only after startup)
        dc      i2'0'              ; head device link
        dc      i2'0'              ; forward device link
        dc      i2'0'              ; Reserved
        dc      i2'0'              ; Reserved
        dc      i2'0'              ; dib device number
        dc      i2'1'              ; drivers internal device number
        dc      i2'0'              ; slot * 16

        eject
```

```
dib_3   entry
        dc      i4'0'            ; link pointer
        dc      i4'dispatch'     ; entry pointer
        dc      h'60 03'         ; characteristics
        dc      i4'0'            ; block count
        dc      i1'10'           ; device name (length & 32 bytes ascii)
        dc      c'CHARACTER3                    '
        dc      i2'1'            ; slot # (valid only after startup)
        dc      i2'1'            ; unit # (valid only after startup)
        dc      i2'1'            ; version # 0001
        dc      h'16 00'         ; device ID # (valid only after startup)
        dc      i2'0'            ; head device link
        dc      i2'0'            ; forward device link
        dc      i2'0'            ; Reserved
        dc      i2'0'            ; Reserved
        dc      i2'0'            ; dib device number
        dc      i2'2'            ; drivers internal device number
        dc      i2'0'            ; slot * 16

        eject
****************************************************************
*
* The following table is used to dispatch to GS/OS driver
* functions.
*
****************************************************************
dispatch_table  entry
        dc      i2'startup-1'
        dc      i2'open-1'
        dc      i2'read-1'
        dc      i2'write-1'
        dc      i2'close-1'
        dc      i2'status-1'
        dc      i2'control-1'
        dc      i2'flush-1'
        dc      i2'shutdn-1'

status_table    entry
        dc      i2'dev_stat-1'
        dc      i2'get_conf-1'
        dc      i2'get_wait-1'
        dc      i2'get_format-1'
        dc      i2'get_partn_map-1'
```

```
control_table    entry
        dc       i2'dev_reset-1'
        dc       i2'format-1'
        dc       i2'media_eject-1'
        dc       i2'set_conf-1'
        dc       i2'set_wait-1'
        dc       i2'set_format-1'
        dc       i2'set_partn-1'
        dc       i2'arm_signal-1'
        dc       i2'disarm_signal-1'
        dc       i2'set_partn_map-1'

status_flag      entry
        dc       i2'0'             ; flag for unit #
        dc       i2'0'             ; flag for unit #
        dc       i2'0'             ; flag for unit #

        eject
****************************************************************
*
* The following table contains the open status for each device
* supported by this driver.
*
****************************************************************
open_table       entry
        dc       i2'0'    ; open state for DIB 1 device
        dc       i2'0'    ; open state for DIB 2 device
        dc       i2'0'    ; open state for DIB 3 device

        eject
```

```
****************************************************************
*
* The following table contains the device status for each
* device supported by this driver.
*
* Encoding of status for a character device is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |__ OPEN
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |______ INTERRUPT
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |__________ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |   |______________ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |__________________ BUSY
*   |   |   |   |   |   |   |   |   |   |   |______________________ 0 RESERVED
*   |   |   |   |   |   |   |   |   |   |__________________________ 0 RESERVED
*   |   |   |   |   |   |   |   |   |______________________________ 0 RESERVED
*   |   |   |   |   |   |   |   |__________________________________ 0 RESERVED
*   |   |   |   |   |   |   |______________________________________ 0 RESERVED
*   |   |   |   |   |   |__________________________________________ 0 RESERVED
*   |   |   |   |   |______________________________________________ 0 RESERVED
*   |   |   |   |__________________________________________________ 0 RESERVED
*   |   |   |______________________________________________________ 0 RESERVED
*   |   |__________________________________________________________ LINKED
*   |______________________________________________________________ 0 RESERVED
*
****************************************************************
dstat_tbl      entry
               dc          i2'dstat1'        ; pointer to status for DIB 1 device
               dc          i2'dstat2'        ; pointer to status for DIB 2 device
               dc          i2'dstat3'        ; pointer to status for DIB 3 device

dstat1         dc          i2'0'             ; device general status word
               dc          i4'0'             ; device block count

dstat2         dc          i2'0'             ; device general status word
               dc          i4'0'             ; device block count

dstat3         dc          i2'0'             ; device general status word
               dc          i4'0'             ; device block count

               eject
```

```
*************************************************************
*
* The following table is used to return the configuration list for
* each device supported by this driver.
*
*************************************************************
clist_tbl       entry
                dc          i2'conf1'          ; pointer to configuration list #1
                dc          i2'conf2'          ; pointer to configuration list #2
                dc          i2'conf3'          ; pointer to configuration list #3

*************************************************************
*
* The following table is used to return the wait mode for
* each device supported by this driver.
*
*************************************************************
wait_mode_tbl   entry
                dc          i2'0'              ; unit 1 wait mode
                dc          i2'0'              ; unit 2 wait mode
                dc          i2'0'              ; unit 3 wait mode

*************************************************************
*
* The following table is used to set the current format
* option for each device supported by this driver.
*
*************************************************************
format_mode     entry
                dc          i2'0'              ; unit 1 format mode
                dc          i2'0'              ; unit 2 format mode
                dc          i2'0'              ; unit 3 format mode

*************************************************************
*
* The following equates are general workspace used by the driver.
*
*************************************************************
retry_count       dc      i2'0'    ; retry count
startup_count     dc      i2'0'

        end

        eject
```

```
****************************************************************
*
* DRIVER MAIN ENTRY POINT: DISPATCH
*
* This is the main entry point for the device driver.  The
* routine validates the call number prior to dispatching to
* the requested function.
*
* Call Number:      $0000       Function:    Startup
*                   $0001                    Open
*                   $0002                    Read
*                   $0003                    Write
*                   $0004                    Close
*                   $0005                    Status
*                   $0006                    Control
*                   $0007                    Flush
*                   $0008                    Shutdown
*         .         $0009-$FFFF              Reserved
*
* ENTRY: Call via 'JSL'
*                  [<drvr_dib_ptr] = Points to the DIB for the device being accessed
*                  <drvr_dev_num = Device number of device being accessed
*                     <drvr_call_num = Call number
*                  A Reg = Call Number
*                  X Reg = Undefined
*                  Y Reg = Undefined
*                  Dir Reg = GS/OS Direct Page
*                  B Reg = Undefined
*                  P Reg = N V M X D I Z C  E
*                          x x 0 0 0 0 x x  0
*
* EXIT:   Direct page = unchanged with the exception of <drvr_tran_cnt
*                  A Reg = Error code
*                  X Reg = Undefined
*                  Y Reg = Undefined
*                  Dir Reg = GS/OS Direct Page
*                  B Reg = Same as entry
*                  P Reg = N V M X D I Z C  E
*                          x x 0 0 0 0 x 0  0          No error occurred
*                          x x 0 0 0 0 x 1  0          Error occurred
*
```

```
****************************************************************
dispatch            start
                    using   driver_data
                    longa   on
                    longi   on

                    phb                         ; save environment
                    phk
                    plb
                    cmp     #max_command        ; is it a legal command?
                    bge     illegal_req         ; no
                    tay                         ; save command #
                    ldx     #$0000
save_parms          anop
                    lda     <drvr_dev_num,x     ; save GS/OS call parameters
                    pha
                    lda     <drvr_blk_num,x
                    pha
                    inx
                    inx
                    cpx     #$000C          ; up to but not including DRVR_TRAN_CNT
                    bne     save_parms

                    tya                         ; restore command #
                    pea     func_ret-1          ; return address from function
                    asl     a                   ; make index to dispatch table
                    tax
                    lda     |dispatch_table,x
                    pha                         ; push function address for dispatch
                    rts                         ; rts dispatches to function
func_ret            anop
                    tay                         ; save error code

                    ldx     #$000A              ; number of words to restore
restore_parms       anop
                    pla                         ; restore GS/OS call parameters
                    sta     <drvr_blk_num,x
                    pla
                    sta     <drvr_dev_num,x
                    dex
                    dex
                    bpl     restore_parms
                    plb
                    bcs     gen_exit            ; force error code 0 if flag cleared
                    ldy     #no_error
gen_exit            anop
                    tya                         ; restore error code
                    rtl
*
```

```
* Received an illegal request.  Return with an error.
*
illegal_req          anop
                     plb                              ; restore environement
                     lda      #drvr_bad_req           ; set error
                     sec
                     rtl

                     end

                     eject
*****************************************************************
*
* DRIVER CALL:    STARTUP
*
* This routine must prepare the driver to accept all other driver
* calls.
*
* ENTRY: Call via 'JSR'
*                    [<drvr_dib_ptr] = Points to DIB for device being accessed
*                    <drvr_dev_num = Device number of device being accessed
*                    <drvr_call_num = Call number
*                    <drvr_tran_cnt = $00000000
*                    A Reg = Call Number
*                    X Reg = Undefined
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Same as program bank
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                    A Reg = Error code
*                    X Reg = Undefined
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Same as entry
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x 0  0       No error occurred
*                            x x 0 0 0 0 x 1  0       Error occurred
*
```

```
*************************************************************
startup             start
                    using    driver_data
                    longa    on
                    longi    on

                    lda      |startup_count          ; has slot been found?
                    beq      search_loop             ; no, go search for it
*
* Check for the device.
*
                    ; insert your code here.
                    bne      no_start_device         ; if you can't find your signature bytes
                    inc      |startup_count
*
* Update the following DIB parameters...
*
*                   Slot Number
*                   Unit Number
*                   Device Name                      (already unique for each DIB)
*
* Note that these DIB parameters are static after startup.
*
                    ldy      #slot_num               ; update DIB slot number
                    lda      |startup_slot
                    sta      [<drvr_dib_ptr],y
                    iny
                    iny
                    lda      |startup_count          ; update DIB unit number
                    sta      [<drvr_dib_ptr],y
                    bra      startup_done
*
* Always request the slot from the slot arbiter prior to scanning the $CnXX space
* for signature bytes when searching for hardware.   This provides a compatible
* method of requesting a slot should a method of dynamic slot switching be made
* available in the future.
*
search_loop
                    lda      |startup_slot           ; request slot from slot arbiter
                    jsl      dyn_slot_arbiter
                    bcs      next_slot               ; if slot was not granted
                    inc      |startup_count
*
* If the slot was granted then use the current slot to search for signature
* bytes identifying your hardware.
*
                    lda      |search_slot            ; create $Cn00 for signature search index
                    and      #$0007
                    ora      #$00C0
                    xba
                    tax      ; X register = $Cn00
*
```

```
* Now search for signatures.
*
                ; insert your code here.
                bne     next_slot               ; if you can't find your signatures bytes
*
* If you find your signature bytes then check for the device.
*
                ; insert your code here.
                bne     no_start_device         ; if you can't find your signatures bytes
*
* Update the following DIB parameters...
*
*               Slot Number
*               Unit Number
*               Device Name                     (already unique for each DIB)
*
* Note that these DIB parameters are static after startup.
*
                ldy     #slot_num               ; update DIB slot number
                lda     |startup_slot
                sta     [<drvr_dib_ptr],y
                iny
                iny
                lda     |startup_count          ; update DIB unit number
                sta     [<drvr_dib_ptr],y
startup_done
                lda     #no_error
                clc
                rts
next_slot
                dec     |startup_slot           ; point at next slot to check
                bne     search_loop             ; and check for hardware
no_start_device
                lda     #drvr_io_error
                sec
                rts

                end

                eject
```

```
*****************************************************************
*
* DRIVER CALL:    OPEN
*
* This call opens a device in preperation for subsequent read
* from or writes to the device.
*
* ENTRY: via a 'JSR'
*                   <drvr_dev_num = Device Number of current device being accessed
*                   <drvr_tran_cnt = $00000000
*                   A Reg = Call Number
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Undefined
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                   A Reg = Error code
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as entry
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x 0  0          No error occurred
*                           x x 0 0 0 0 x 1  0          Error occurred
*
```

```
****************************************************************
open            start
                using   driver_data
                longa   on
                longi   on

                ldy     #driver_unit            ; get internal device reference number
                lda     [<drvr_dib_ptr],y
                dec     a
                asl     a
                tax
                sec                             ; assume a device error
                lda     |open_stat,x            ; can device be opened?
                bne     exit                    ; no, exit w/error
                lda     |open_table,x           ; else get current open state
                bne     already_open            ; if device is already open
                dec     |open_table,x           ; set device open
*
* At this point, your driver may wish to allocate system resources such
* as memory from the memory manager for use in buffering, etc.
*

                lda     #no_error               ; and exit w/o error
                clc
                rts

already_open    anop
                lda     #drvr_prior_open
                sec
exit            anop
                rts

open_stat       anop
                dc      i2'no_error'            ; status for dib 1 device
                dc      i2'no_error'            ; status for dib 2 device
                dc      i2'no_error'            ; status for dib 3 device
                dc      i2'no_error'            ; status for dib 4 device
                dc      i2'no_error'            ; status for dib 5 device
                dc      i2'no_error'            ; status for dib 6 device
                dc      i2'no_error'            ; status for dib 7 device
                dc      i2'no_error'            ; status for dib 8 device

                end

                eject
```

```
****************************************************************
*
* DRIVER CALL:    READ
*
* This call executes a read from the device.  For block devices
* the call must validate the initial block number and that the
* request count is an integral multiple of the block size.    In
* addition the block number of each successive block must be
* validated as it is accessed when an multiple block I/O
* transaction is in place.
*
* ENTRY: via a 'JSR'
*       <drvr_dev_num = Device Number of current device being accessed
*       <drvr_buf_ptr = Pointer to I/O buffer
*       <drvr_blk_num = Initial block number
*       <drvr_req_cnt = Number of bytes to be transferred
*       <drvr_blk_size = Size of block to be accessed
*       <drvr_tran_cnt = $00000000
*       A Reg = Call Number
*       X Reg = Undefined
*       Y Reg = Undefined
*       Dir Reg = GS/OS Direct Page
*       B Reg = Undefined
*       P Reg = N V M X D I Z C  E
*               x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*       <drvr_tran_cnt = Number of bytes transferred
*       A Reg = Error code
*       X Reg = Undefined
*       Y Reg = Undefined
*       Dir Reg = GS/OS Direct Page
*       B Reg = Same as entry
*       P Reg = N V M X D I Z C  E
*               x x 0 0 0 0 x 0  0          if no error
*               x x 0 0 0 0 x 1  0          if error
*
```

```
*****************************************************************
read            start
                using       driver_data
                longa       on
                longi       on

*
* If the call seems valid then all that remains to be done prior to executing
* the I/O transaction is to check that the device is open.    If the device is
* not open then a 'driver not open' error will be returned with no I/O
* transaction executed.
*
                ldy         #driver_unit        ; driver's internal device list ref
                lda         [<drvr_dib_ptr],y
                tax
                lda         |open_table,x       ; is device open?
                bne         character_read      ; yes
                lda         #drvr_not_open      ; else return error
                sec
                rts
*
* The character device is open.  At this point two types of I/O transaction
* are possible.  The transaction can be executed in WAIT or NO WAIT mode.
* In wait mode the driver will poll the I/O device for each byte read until
* either an I/O error occurs or the request count is satisfied.  In NO WAIT
* mode the driver will return the number of bytes currently held in the
* driver's own I/O buffer.  NO WAIT mode implies that the driver is running
* with an interrupt handler which manages the buffering of I/O for the device.
* Our driver is set up to run in either mode but no interrupt handler has
* been installed.  Since no hardware exists for this driver many hardware
* specific routines such as interrupt handlers have been deleted although
* the general environment for interrupt handlers has been provided.
*
```

```
character_read      anop
                    ldy       #my_slot16          ; need slot * 16 (from DIB extension)
                    lda       [<drvr_dib_ptr],y
                    tax
wait_char_ready     anop
                    lda       >ready,x            ; is character ready?
                    bmi       read_char           ; yes
                    ldy       #driver_unit        ; is driver in wait mode?
                    lda       [<drvr_dib_ptr],y
                    tax
                    bit       |wait_mode_tbl,x
                    bpl       wait_char_ready ; yes
                    bmi       char_rd_exit        ; else exit
read_char           anop
                    lda       >char,x             ; then read character into buffer
                    sta       [<drvr_buf_ptr]
                    inc       <drvr_buf_ptr       ; adjust buffer for next character
                    bne       incr_tran_cnt
                    inc       <drvr_buf_ptr+2
incr_tran_cnt       anop
                    inc       <drvr_tran_cnt      ; adjust transfer count
                    bne       check_req_cnt
                    inc       <drvr_tran_cnt+2
check_req_cnt       anop
                    lda       <drvr_tran_cnt      ; has request count been satisfied?
                    cmp       <drvr_req_cnt
                    bne       wait_char_ready ; no
                    lda       <drvr_tran_cnt+2
                    cmp       <drvr_req_cnt+2
                    bne       wait_char_ready ; no
char_rd_exit        anop
                    lda       #no_error           ; yes, then return with no error
                    clc
                    rts

                    end

                    eject
```

```
****************************************************************
*
* DRIVER CALL:    WRITE
*
* This call executes a write to the device.  For block devices
* the call must validate the initial block number and that the
* request count is an integral multiple of the block size.     In
* addition the block number of each successive block must be
* validated as it is accessed when an multiple block I/O
* transaction is in place.
*
* ENTRY: via a 'JSR'
*       <drvr_dev_num = Device Number of current device being accessed
*       <drvr_buf_ptr = Pointer to I/O buffer
*       <drvr_blk_num = Initial block number
*       <drvr_req_cnt = Number of bytes to be transferred
*       <drvr_blk_size = Size of block to be accessed
*       <drvr_tran_cnt = $00000000
*       A Reg = Call Number
*       X Reg = Undefined
*       Y Reg = Undefined
*       Dir Reg = GS/OS Direct Page
*       B Reg = Undefined
*       P Reg = N V M X D I Z C  E
*               x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*       <drvr_tran_cnt = Number of bytes transferred
*       A Reg = Error code
*       X Reg = Undefined
*       Y Reg = Undefined
*       Dir Reg = GS/OS Direct Page
*       B Reg = Same as entry
*       P Reg = 0=m=x=e
*       Carry = 1 if error occurred
*       Carry = 0 if no error occurred
*
```

```
***************************************************************
write    start
         using    driver_data
         longa    on
         longi    on
*
* If the call seems valid then all that remains to be done prior to executing
* the I/O transaction is to check that the device is open.    If the device is
* not open then a 'driver not open' error will be returned with no I/O
* transaction executed.
*
                   ldy      #driver_unit         ; driver's internal device list ref
                   lda      [<drvr_dib_ptr],y
                   tax
                   lda      |open_table,x        ; is device open?
                   bne      character_write      ; yes
                   lda      #drvr_not_open       ; else return error
                   sec
                   rts
*
* The character device is open.  Go ahead and write data to the device.
*
character_write    anop
                   ldy      #my_slot16           ; need slot * 16 (from DIB extension)
                   lda      [<drvr_dib_ptr],y
                   tax
wait_char_ready    anop
                   lda      >ready,x             ; is device ready for character?
                   bpl      wait_char_ready      ; no
                   lda      [<drvr_buf_ptr]
                   sta      >char,x              ; then write character into buffer
                   inc      <drvr_buf_ptr        ; adjust buffer for next character
                   bne      incr_tran_cnt
                   inc      <drvr_buf_ptr+2
incr_tran_cnt      anop
                   inc      <drvr_tran_cnt       ; adjust transfer count
                   bne      check_req_cnt
                   inc      <drvr_tran_cnt+2
check_req_cnt      anop
                   lda      <drvr_tran_cnt       ; has request count been satisfied?
                   cmp      <drvr_req_cnt
                   bne      wait_char_ready      ; no
                   lda      <drvr_tran_cnt+2
                   cmp      <drvr_req_cnt+2
                   bne      wait_char_ready      ; no
                   lda      #no_error            ; yes, then return with no error
                   clc
                   rts

                   end

                   eject
```

```
**************************************************************
*
* DRIVER CALL:    CLOSE
*
* This call closes a device and returns the device to the
* same state that existed prior to an open call.
*
* ENTRY: via a 'JSR'
*       <drvr_dev_num = Device Number of current device being accessed
*       <drvr_tran_cnt = $00000000
*       A Reg = Call Number
*       X Reg = Undefined
*       Y Reg = Undefined
*       Dir Reg = GS/OS Direct Page
*       B Reg = Undefined
*       P Reg = N V M X D I Z C  E
*               x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*       A Reg = Error code
*       X Reg = Undefined
*       Y Reg = Undefined
*       Dir Reg = GS/OS Direct Page
*       B Reg = Same as entry
*       P Reg = N V M X D I Z C  E
*               x x 0 0 0 0 x 0  0         No error occurred
*               x x 0 0 0 0 x 1  0         Error occurred
*
```

```
************************************************************
close             start
                  using      driver_data
                  longa      on
                  longi      on

                  ldy        #driver_unit         ; get internal device reference number
                  lda        [<drvr_dib_ptr],y
                  dec        a
                  asl        a
                  tax
                  sec                             ; assume a device error
                  lda        |close_stat,x        ; can device be closed?
                  bne        exit                 ; no, exit w/error
                  lda        |open_table,x        ; else get current open state
                  beq        already_closed       ; if device is already closed
                  inc        |open_table,x        ; set device open
*
* At this point, your driver should release any system resources such
* as memory, that were aquired during the open call.
*

                  lda        #no_error            ; and exit w/o error
                  clc
                  rts

already_closed    anop
                  lda        #drvr_not_open
                  sec
exit              anop
                  rts

close_stat        anop
                  dc         i2'no_error'         ; status for dib 1 device
                  dc         i2'no_error'         ; status for dib 2 device
                  dc         i2'no_error'         ; status for dib 3 device
                  dc         i2'no_error'         ; status for dib 4 device
                  dc         i2'no_error'         ; status for dib 5 device
                  dc         i2'no_error'         ; status for dib 6 device
                  dc         i2'no_error'         ; status for dib 7 device
                  dc         i2'no_error'         ; status for dib 8 device

                  end

                  eject
```

```
*****************************************************************
*
* DRIVER CALL:    STATUS
*
* This routine supports all the standard device status calls.
* Any status call which is able to detect an OFFLINE or DISK
* SWITCHED condition should call the system service routine
* SET_DISKSW.  OFFLINE and DISKSW are conditions and not errors
* when detected by a status call and should only be returned as
* conditions in the status list.
*
* Status Code:    $0000    Return Device Status
*                 $0001    Return Configuration Parameters
*                 $0002    Return Wait/No Wait Mode
*                 $0003    Return Format Options
*                 $0004    Return Partition Map
*
* ENTRY: via a 'JSR'
*        <drvr_dev_num = Device Number of current device being accessed
*        <drvr_clist_ptr = Pointer to control list
*        <drvr_ctrl_code = Control code
*        <drvr_req_cnt = Number of bytes to be transferred
*        <drvr_tran_cnt = $00000000
*        A Reg = Call Number
*        X Reg = Undefined
*        Y Reg = Undefined
*        Dir Reg = GS/OS Direct Page
*        B Reg = Undefined
*        P Reg = N V M X D I Z C  E
*                x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*           <drvr_tran_cnt = Number of bytes transferred
*        A Reg = Error code
*        X Reg = Undefined
*        Y Reg = Undefined
*        Dir Reg = GS/OS Direct Page
*        B Reg = Same as entry
*        P Reg = N V M X D I Z C  E
*                x x 0 0 0 0 x 0  0          No error occurred
*                x x 0 0 0 0 x 1  0          Error occurred
*
*****************************************************************
status            start
                  using      driver_data
                  longa      on
                  longi      on

*
```

```
* Need to verify that the status code specifies a
* legal status request.
*
                lda         <drvr_stat_code         ; is this a legal status request?
                cmp         #$0004
                blt         legal_status            ; yes
                lda         #drvr_bad_code          ; else return 'BAD CODE' error
                rts
*
* It's a legal status.  Dispatch to the appropriate status routine.
*
legal_status    anop
                asl         a
                tax
                lda         |status_table,x
                pha
                rts                                 ; dispatch is via an 'RTS'

                eject
*****************************************************************
*
* The DEVICE STATUS call returns a status list that indicates
* specific status information regarding a character device.
*
* Status List Pointer:    Word            General status word
*                         Longword        Total number of blocks
*
* Character devices should indicate $00000000 as the block count.
* Status conditions are bit encoded in the status word.
* Encoding of status for a character device is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |__ OPEN
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |______ INTERRUPT
*   |   |   |   |   |   |   |   |   |   |   |   |   |___|__________ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |__________________ BUSY
*   |   |   |___|___|___|___|___|___|___|___|___|__________________ 0 RESERVED
*   |   |__________________________________________________________ LINKED DEV
*   |______________________________________________________________ 0 RESERVED
*
* Valid request counts versus returned status list for this status
* call are as follows:
*
* Request Count: $0002    Status List: General status word
* Request Count: $0006    Status List: General status word and block count
*
```

```
******************************************************************
dev_stat      entry
              longa     on
              longi     on

              lda       #drvr_bad_parm      ; assume invalid request count
              ldx       <drvr_req_cnt+2     ; and validate request count
              bne       bad_dev_stat
              ldx       <drvr_req_cnt
              cpx       #$0002
              blt       bad_dev_stat
              cpx       #$0007
              blt       ok_dev_stat
bad_dev_stat  anop
              sec
              rts
*
* Request count is valid.  Deterime device status and if appropriate, the
* total number of blocks for the device and return them in the device
* status list.  You insert the code required for this operation.
*
ok_dev_stat             anop
                        ldy       #driver_unit              ; internal device #
                        lda       [<drvr_dib_ptr],y
                        asl       a
                        tax
                        lda       |dstat_tbl,x              ; get pointer to device status list
                        tax
                        ldy       #$0000                    ; status list pointer
                        sep       #$20                      ; 8 bit 'm'
                        longa     off
copy_dstat              anop
                        lda       |0,x                      ; copy device status list to slist_ptr
                        sta       [<drvr_slist_ptr],y
                        inx
                        iny
                        cpy       <drvr_req_cnt             ; copy = request count size
                        bne       copy_dstat
                        rep       #$20                      ; 16 bit 'm'
                        longa     on
                        brl       set_xfer_cnt              ; update xfer count & exit

                        eject
```

```
****************************************************************
*
* This call returns a byte count as the first word in the status
* list followed by the data from the configuration parameter list.
* The request count specifies how much data is to be returned
* from the list.  If the byte count is smaller than the request
* count then only the number of bytes specified by the byte
* count will be returned and the transfer count will indicate
* this.
*
* Status List:    Word     Length of configuration list (including count).
*                 Data     Data from the configuration list (device specific).
*
* This call requires a minimum request count of $00000002 and
* a maximum request count of $0000FFFF.
*
****************************************************************
get_conf      entry
              longa       on
              longi       on

              lda         #drvr_bad_parm      ; assume invalid request count
              ldx         <drvr_req_cnt+2     ; and validate request count
              bne         bad_get_ctrl
              ldx         <drvr_req_cnt
              cpx         #$0002
              bge         ok_get_ctrl
bad_get_ctrl              anop
              sec
              rts
*
```

```
* Request count is valid.  Return configuration list.
*
ok_get_ctrl         anop
                    ldy      #driver_unit          ; internal device #
                    lda      [<drvr_dib_ptr],y
                    asl      a
                    tax
                    lda      |clist_tbl,x          ; get pointer to configuration list
                    tax
                    lda      |0,x                  ; get length of configuration list
                    beq      no_clist              ; if list has no content
                    cmp      <drvr_req_cnt         ; is list shorter than request?
                    bge      req_cnt_ok            ; no
                    sta      <drvr_req_cnt         ; else modify request count
req_cnt_ok          anop
                    ldy      #$0000                ; status list index
                    sep      #$20                  ; 8 bit 'm'
                    longa    off
copy_clist          anop
                    lda      |0,x                  ; copy configuration list to slist_ptr
                    sta      [<drvr_slist_ptr],y
                    inx
                    iny
                    cpy      <drvr_req_cnt         ; copy = request count size
                    bne      copy_clist
                    rep      #$20                  ; 16 bit 'm'
                    longa    on
                    brl      set_xfer_cnt

no_clist            anop
                    sta      [<drvr_slist_ptr]
                    lda      #$0002
                    brl      set_xfer_cnt

                    eject
```

```
***************************************************************
*
* This routine returns the Wait/No Wait mode that the driver
* is currently operating in.  This is returned in a word
* parameter which indicates a request count of $0002.
*
***************************************************************
get_wait             entry
                     longa     on
                     longi     on

                     lda       #drvr_bad_parm          ; assume invalid request count
                     ldx       <drvr_req_cnt+2         ; and validate request count
                     bne       bad_get_wait            ≠ Ø => bad
                     ldx       <drvr_req_cnt
                     cpx       #$0002
                     beq       ok_get_wait
bad_get_wait         anop
                     sec
                     rts
*
* Request count is valid. Return the wait mode for this device.
*
ok_get_wait          anop
                     ldy       #unit_num
                     lda       [<drvr_dib_ptr],y
                     tax
                     dex
                     lda       |wait_mode_tbl,x
                     sta       [<drvr_slist_ptr]
                     brl       set_xfer_cnt

                     eject
***************************************************************
*
* This routine returns the format options for the device.
* Character devices do not support formatting and will return
* with no error and a transfer count of NIL.
*
***************************************************************
get_format           entry
                     longa     on
                     longi     on

                     lda       #no_error               ; no action this call
                     clc
                     rts
```

```
****************************************************************
*
* This is a common exit routine for successful status calls.
* The transfer count is set to the same value as the request
* count prior to returning with no error.
*
****************************************************************
set_xfer_cnt        entry
                    lda       <drvr_req_cnt           ; set transfer count
                    sta       <drvr_tran_cnt
                    lda       <drvr_req_cnt+2
                    sta       <drvr_tran_cnt+2
                    lda       #no_error
                    clc
                    rts

                    eject
****************************************************************
*
* This routine returns the partition map for the device.
* Character devices do not support partitioning and will return
* with no error and a transfer count of NIL.
*
****************************************************************
get_partn_map       entry
                    longa     on
                    longi     on

                    lda       #no_error               ; no action this call
                    clc
                    rts

                    end

                    eject
```

```
*****************************************************************
*
* DRIVER CALL:          CONTROL
*
* This routine supports all the standard device control calls.
*
* Control Code: $0000   Reset Device
*                       $0001       Format Device
*                       $0002       Eject Media
*                       $0003       Set Configuration Parameters
*                       $0004       Set Wait/No Wait Mode
*                       $0005       Set Format Options
*                       $0006       Assign Partition Owner
*                       $0007       Arm Signal
*                       $0008       Disarm Signal
*                       $0009       Set Partition Map
*
* ENTRY: via a 'JSR'
*        <drvr_dev_num = Device Number of current device being accessed
*        <drvr_clist_ptr = Pointer to control list
*        <drvr_ctrl_code = Control code
*        <drvr_req_cnt = Number of bytes to be transferred
*        <drvr_tran_cnt = $00000000
*        A Reg = Call Number
*        X Reg = Undefined
*        Y Reg = Undefined
*        Dir Reg = GS/OS Direct Page
*        B Reg = Undefined
*        P Reg = N V M X D I Z C  E
*                x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*        <drvr_tran_cnt = Number of bytes transferred
*        A Reg = Error code
*        X Reg = Undefined
*        Y Reg = Undefined
*        Dir Reg = GS/OS Direct Page
*        B Reg = Same as entry
*        P Reg = N V M X D I Z C  E
*                x x 0 0 0 0 x 0  0          No error occurred
*                x x 0 0 0 0 x 1  0          Error occurred
*
```

```
***************************************************************
control          start
                 using     driver_data
                 longa     on
                 longi     on

*
* Need to verify that the control code specifies a
* legal control request.
*
                 lda       <drvr_ctrl_code          ; is this a legal control request?
                 cmp       #$0009
                 blt       legal_control            ; yes
                 lda       #drvr_bad_code           ; else return 'BAD CODE' error
                 rts
*
* It's a legal control.  Dispatch to the appropriate control routine.
*
legal_control    anop
                 asl       a
                 tax
                 lda       |control_table,x
                 pha
                 rts       ; dispatch is via an 'RTS'

                 eject
***************************************************************
*
* This routine will reset the device to it's default conditions
* as specified by the default configuration parameter list.  The
* configuration list contents will be updated to reflect the parameter
* changes that have taken effect.  Since our driver has a configuration
* parameter list of NIL, no action is taken.
*
* CONTROL LIST:  None
*
***************************************************************
dev_reset        entry
         lda     #no_error
         clc
         rts

         eject
```

```
***************************************************************
*
* Character devices do not support the FORMAT function and will
* return with no error and a transfer count of NIL.
*
* CONTROL LIST:   None
*
***************************************************************
format          entry
                lda       #no_error
                clc
                rts

                eject
***************************************************************
*
* Character devices do not support the media eject command and
* will return with no error and a transfer count of NIL.
*
* CONTROL LIST:   None
*
***************************************************************
media_eject     entry
                lda       #no_error
                clc
                rts

                eject
***************************************************************
*
* This routine will set the configuration parameter list as specified
* by the contents of the configuration list.  Note that the first
* word of the configuration list must have the same value as the
* current configuration parameter list.
*
* CONTROL LIST:   Word      Size of configuration parameter list
*                 Data      Configuration parameter list
*
***************************************************************
set_conf        entry
                longa     on
                longi     on

                lda       #drvr_bad_parm          ; assume invalid request count
                ldx       <drvr_req_cnt+2         ; and validate request count
                bne       bad_set_ctrl
                ldx       <drvr_req_cnt
                cpx       #$0002
                bge       ok_set_ctrl
bad_set_ctrl    anop
                sec
                rts
*
```

```
* Request count is valid.  Set configuration list.
*
ok_set_ctrl     anop
                ldy       #driver_unit          ; internal device #
                lda       [<drvr_dib_ptr],y
                asl       a
                tax
                lda       |clist_tbl,x          ; get pointer to configuration list
                tax
                lda       |0,x                  ; are lengths the same?
                cmp       [<drvr_clist_ptr]
                beq       req_cnt_ok            ; yes
                lda       #drvr_bad_parm        ; else return an error
                sec
                rts
req_cnt_ok      anop
                ldy       #$0000                ; status list index
                sep       #$20                  ; 8 bit 'm'
                longa     off
copy_clist      anop                            ; set new configuration list
                lda       [<drvr_clist_ptr],y
                sta       |0,x
                inx
                iny
                tya
                cmp       [<drvr_clist_ptr]
                bne       copy_clist
                rep       #$20                  ; 16 bit 'm'
                longa     on
*
* Prior to exiting with the proper transfer count, your driver would
* have to put the new configuration parameters into effect.  The driver
* should reconfigure itself based on the values passed in the new
* configuration parameter list.
*

*
* After the new parameters have been put into effect, exit with the
* proper transfer count.
*
                brl       set_xfer_cnt

                eject
****************************************************************
*
* This routine will set the WAIT/NO WAIT mode as specified
* by the contents of the control list.
*
* CONTROL LIST:   Word     Wait / No Wait Mode
*
****************************************************************
set_wait        entry
                longa     on
```

```
                longi   on

                lda     #drvr_bad_parm          ; assume invalid request count
                ldx     <drvr_req_cnt+2         ; and validate request count
                bne     bad_set_wait
                ldx     <drvr_req_cnt
                cpx     #$0002
                beq     ok_set_wait
bad_set_wait    anop
                sec
                rts
*
* Request count is valid. Set the wait mode for this device.
*
ok_set_wait     anop
                ldy     #driver_unit
                lda     [<drvr_dib_ptr],y
                tax
                lda     [<drvr_slist_ptr]
                sta     |wait_mode_tbl,x
                brl     set_xfer_cnt

                eject
***************************************************************
*
* Format options are not supported by character devices and will
* return with no error and a transfer count of NIL.
*
* CONTROL LIST:  Word     Format Option Reference Number
*
***************************************************************
set_format      entry
                longa   on
                longi   on

                lda     #no_error       ; exit without action
                clc
                rts

                eject
***************************************************************
*
* Character devices do not support partitions and will return
* with no error and a transfer count of NIL.
*
* CONTROL LIST:  Word     String length
*                Name     Name of partition owner
*
***************************************************************
set_partn       entry
                lda     #no_error
                clc
                rts
```

```
                eject
***************************************************************
*
* This routine is envoked by an application to install a signal
* into the event mechanism.
*
* CONTROL LIST:  Word     Signal Code
*                Word     Signal Priority
*                Long     Signal Handler Address
*
***************************************************************
arm_signal      entry
                lda      #no_error
                clc
                rts

                eject
***************************************************************
*
* This routine is remove a signal from the event mechanism that
* was previously installed with the arm_signal call.
*
* CONTROL LIST:  Word     Signal Code
*
***************************************************************
disarm_signal   entry
                lda      #no_error
                clc
                rts

                eject
***************************************************************
*
* This routine is used to set the partition map for the device.
* Character devices do not support partitioning and will return
* with no error and a transfer count of NIL.
*
***************************************************************
set_partn_map   entry
                lda      #no_error
                clc
                rts

                end

                eject
```

```
****************************************************************
*
* DRIVER CALL:    FLUSH
*
* This call writes any data in the devices internal buffer to
* the device.  It should be noted that this is a WAIT MODE call
* which is only supported by devices which maintain their own
* internal I/O buffer.     Devices that cannot write in NO WAIT mode
* do not support this call and will return with no error.
*
* ENTRY: via a 'JSR'
*       <drvr_dev_num = Device Number of current device being accessed
*       <drvr_tran_cnt = $00000000
*       A Reg = Call Number
*       X Reg = Undefined
*       Y Reg = Undefined
*       Dir Reg = GS/OS Direct Page
*       B Reg = Undefined
*       P Reg = N V M X D I Z C  E
*               x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*       <drvr_tran_cnt = Number of bytes transferred
*       A Reg = Error code
*       X Reg = Undefined
*       Y Reg = Undefined
*       Dir Reg = GS/OS Direct Page
*       B Reg = Same as entry
*       P Reg = N V M X D I Z C  E
*               x x 0 0 0 0 x 0  0         No error occurred
*               x x 0 0 0 0 x 1  0         Error occurred
*
```

```
*****************************************************************
flush           start
                using   driver_data
                longa   on
                longi   on

                ldy     #driver_unit        ; get internal device reference number
                lda     [<drvr_dib_ptr],y
                asl     a
                tax
                sec                         ; assume device will return an error
                lda     |flush_stat,x       ; get status for this device
                bne     exit
                clc
exit            anop
                rts

flush_stat      anop
                dc      i2'no_error'        ; status for dib 1 device
                dc      i2'no_error'        ; status for dib 2 device
                dc      i2'no_error'        ; status for dib 3 device
                dc      i2'no_error'        ; status for dib 4 device
                dc      i2'no_error'        ; status for dib 5 device
                dc      i2'no_error'        ; status for dib 6 device
                dc      i2'no_error'        ; status for dib 7 device
                dc      i2'no_error'        ; status for dib 8 device

                end

                eject
```

```
****************************************************************
*
* DRIVER CALL:    SHUTDOWN
*
* This call prepares the driver for shutdown.  This may include
* closing a character device as well as releasing any and all
* system resources that may have been aquired by either a
* STARTUP or OPEN call.  Many devices may share a common code segment.
* If this is the case, an error should be returned on shutdown from all
* but the last code segment.  The device dispatcher will free up the
* memory occupied by the driver when no error is returned on shutdown.
*
* ENTRY: via a 'JSR'
*        <drvr_dev_num = Device Number of current device being accessed
*        <drvr_tran_cnt = $00000000
*        A Reg = Call Number
*        X Reg = Undefined
*        Y Reg = Undefined
*        Dir Reg = GS/OS Direct Page
*        B Reg = Undefined
*        P Reg = N V M X D I Z C  E
*                x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*        <drvr_tran_cnt = Number of bytes transferred
*        A Reg = Error code
*        X Reg = Undefined
*        Y Reg = Undefined
*        Dir Reg = GS/OS Direct Page
*        B Reg = Same as entry
*        P Reg = N V M X D I Z C  E
*                x x 0 0 0 0 x 0  0          No error occurred
*                x x 0 0 0 0 x 1  0          Error occurred
*
****************************************************************
shutdn       start
             using      driver_data
             longa      on
             longi      on

             dec        |startup_count      ; is this the last device shutdown?
             bne        not_last            ; no
             lda        #no_error           ; else return no error on last device
             clc
             rts

not_last     anop
             lda        #drvr_busy          ; return an error if not last
             sec
             rts

             end
```

Supervisory driver

This is the shell of a typical supervisory driver, whose job is to mediate among several device drivers that access several hardware device through the same hardware controller. See Chapter 8 of this volume. The driver code consists of five parts, in this order:

- Equates
- Supervisor-driver header
- Tables for dispatching calls and passing parameters
- A main entry point to the driver
- Routines that handle the driver calls

The driver has handlers for three calls (Supervisor_Startup, Supervisor_Specific, and Supervisor_Shutdown), although the Supervisor_Specific call is a nonfunctional skeleton in this example.

```
                    65816      on
                    instime    on
                    gen        on
                    symbol     on
                    absaddr    on
                    align      256
*****************************************************************
*
*                   Copyright (c) 1988
*                   Apple Computer, Inc.
*                   All rights reserved.
*
*****************************************************************
*
*                    Supervisory Driver Core Routines Version 0.01a01
*
* NOTE:             All supervisor driver files must be installed on the
*                    boot volume in the subdirectory "/SYSTEM/DRIVERS'.
*                    Additionally, the FileType for the driver file
*                    must be set to $00BB.          AuxType is also critical
*                    to the operating system recognizing the driver
*                    as a GS/OS supervisory driver.  The AuxType is a long
*                    word which must have the upper word set to $0000.
*                    The least significant word of the AuxType field should
*                    be set to $0140 for supervisory driver files. The
*                    supervisory driver file should be compacted to OMF2.
*
*                    GS/OS Supervisory Driver:      FileType = $00BB
*                                                   AuxType  = $00000140
*
*****************************************************************
*
* REVISION HISTORY:
*
* DATE              Ver.      By                    Description
*
* 02/26/88          0.00e01   RBM                   Started initial coding.
*
*****************************************************************

                    eject
```

```
*****************************************************************
*
* The following are direct page equates on the GS/OS
* direct page for driver usage.
*
*****************************************************************

drvr_dev_num        gequ     $00                 ; (w)  device number
drvr_call_num       gequ     drvr_dev_num+2      ; (w)  call number
drvr_buf_ptr        gequ     drvr_call_num+2     ; (lw) buffer pointer
drvr_slist_ptr      gequ     drvr_call_num+2     ; (lw) buffer pointer
drvr_clist_ptr      gequ     drvr_call_num+2     ; (lw) buffer pointer
dev_id_ref          gequ     drvr_buf_ptr        ; (w)  indirect device ID
drvr_req_cnt        gequ     drvr_buf_ptr+4      ; (lw) request count
drvr_tran_cnt       gequ     drvr_req_cnt+4      ; (lw) transfer count
drvr_blk_num        gequ     drvr_tran_cnt+4     ; (lw) block number
drvr_blk_size       gequ     drvr_blk_num+4      ; (w)  block size
drvr_fst_num        gequ     drvr_blk_size+2     ; (w)  File System Translator Number
drvr_stat_code      gequ     drvr_fst_num        ; (w)  status code for status call
drvr_ctrl_code      gequ     drvr_fst_num        ; (w)  control code for control call
drvr_vol_id         gequ     drvr_fst_num+2      ; (w)  Driver Volume ID Number
drvr_cache          gequ     drvr_vol_id+2       ; (w)  Cache Priority
drvr_cach_ptr       gequ     drvr_cache+2        ; (lw) pointer to cached block
drvr_dib_ptr        gequ     drvr_cach_ptr+4     ; (lw) pointer to active DIB

sib_ptr             gequ     $0074               ; (lw) pointer to active SIB
sup_parm_ptr        gequ     sib_ptr+4           ; (lw) pointer to supervisor parameters

                    eject
```

```
****************************************************************
*
* The following are equates for driver command types.
*
****************************************************************

drvr_startup        gequ     $0000                 ; driver startup command
drvr_open           gequ     $0001                 ; driver open command
drvr_read           gequ     $0002                 ; driver read command
drvr_write          gequ     $0003                 ; driver write command
drvr_close          gequ     $0004                 ; driver close command
drvr_status         gequ     $0005                 ; driver status command
drvr_control        gequ     $0006                 ; driver control command
drvr_flush          gequ     $0007                 ; driver flush command
drvr_shutdn         gequ     $0008                 ; driver shutdown command
max_command         gequ     $0009                 ; commands $0009 - $ffff undefined

drvr_dev_stat       gequ     $0000                 ; status code: return device status
drvr_ctrl_stat      gequ     $0001                 ; status code: return control params
drvr_get_wait       gequ     $0002                 ; status code: get wait/no wait mode
drvr_get_format     gequ     $0003                 ; status code: get format options

drvr_reset          gequ     $0000                 ; control code: reset device
drvr_format         gequ     $0001                 ; control code: format device
drvr_eject          gequ     $0002                 ; control code: eject media
drvr_set_ctrl       gequ     $0003                 ; control code: set control params
drvr_set_wait       gequ     $0004                 ; control code: set wait/no wait mode
drvr_set_format     gequ     $0005                 ; control code: set format options
drvr_set_ptn        gequ     $0006                 ; control code: set partition owner
drvr_arm            gequ     $0007                 ; control code: arm interrupt signal
drvr_disarm         gequ     $0007                 ; control code: arm interrupt signal

                    eject
```

```
*****************************************************************
*
* The following are equates for GS/OS error codes.
*
*****************************************************************

no_error          gequ     $0000            ; no error has occurred
dev_not_found     gequ     $0010            ; device not found
invalid_dev_num   gequ     $0011            ; invalid device number
drvr_bad_req      gequ     $0020            ; bad request or command
drvr_bad_code     gequ     $0021            ; bad control or status code
drvr_bad_parm     gequ     $0022            ; bad call parameter
drvr_not_open     gequ     $0023            ; character device not open
drvr_prior_open   gequ     $0024            ; character device already open
irq_table_full    gequ     $0025            ; interrupt table full
drvr_no_resrc     gequ     $0026            ; resources not available
drvr_io_error     gequ     $0027            ; I/O error
drvr_no_dev       gequ     $0028            ; device not connected
drvr_busy         gequ     $0029            ; call aborted, driver is busy
drvr_wr_prot      gequ     $002B            ; device is write protected
drvr_bad_count    gequ     $002C            ; invalid byte count
drvr_bad_block    gequ     $002D            ; invalid block address
drvr_disk_sw      gequ     $002E            ; disk has been switched
drvr_off_line     gequ     $002F            ; device off line / no media present
invalid_access    gequ     $004E            ; access not allowed
parm_range_err    gequ     $0053            ; parameter out of range
out_of_mem        gequ     $0054            ; out of memory
dup_volume        gequ     $0057            ; duplicate volume name
not_block_dev     gequ     $0058            ; not a block device
stack_overflow    gequ     $005F            ; too many applications on stack
data_unavail      gequ     $0060            ; data unavailable


                  eject
*****************************************************************
*
* The following are equates for the DIB.
*
*****************************************************************
link_ptr          gequ     $0000            ; (lw) pointer to next DIB
entry_ptr         gequ     $0004            ; (lw) pointer to driver
dev_char          gequ     $0008            ; (w)  device characteristics
blk_cnt           gequ     $000A            ; (lw) number of blocks
dev_name          gequ     $000E            ; (32) count and ascii name (pstring)
slot_num          gequ     $002E            ; (w)  slot number
unit_num          gequ     $0030            ; (w)  unit number
ver_num           gequ     $0032            ; (w)  version number
dev_id_num        gequ     $0034            ; (w)  device ID number (ICON ref#)
head_link         gequ     $0036            ; (w)  backward device link
forward_link      gequ     $0038            ; (w)  forward device link
link_dib_ptr      gequ     $003A            ; (lw) dib reserved field #1
dib_dev_num       gequ     $003E            ; (w)  Device number of this device
*
```

```
* The following equate(s) are for drive specific extensions to the DIB.
* Parameters that are extended to the manditory DIB parameters are not
* accessable by GS/OS or the application but may be used within a driver
* as needed.
*
driver_unit         gequ      $0040               ; (w)  driver's internal DIB data
my_slot16           gequ      $0042               ; (w)  driver's slot * 16


                    eject
****************************************************************
*
* System Service Table Equates:
*
* NOTE: Only those system service calls that might be used
* by a device driver are listed here.  For a more complete
* list of system service calls and explanations of each call
* consult the system service call ERS.
*
****************************************************************
dev_dispatcher      gequ      $01FC00             ; dev_dispatch
cache_find_blk      gequ      $01FC04             ; cash_find
cache_add_blk       gequ      $01FC08             ; cash_add
cache_del_blk       gequ      $01FC14             ; cash_delete
cache_del_vol       gequ      $01FC18             ; cash_del_vol
set_sys_speed       gequ      $01FC50             ; set system speed
move_info           gequ      $01FC70             ; gs_move_block
set_disksw          gequ      $01FC90             ; set disksw and call swapout/delvol
sup_drvr_disp       gequ      $01FCA4             ; supervisor dispatcher
install_driver      gequ      $01FCA8             ; dynamic driver installation
dyn_slot_arbiter    gequ      $01FCBC             ; dynamic slot arbiter

                    eject
```

```
***************************************************************
*
* MOVE_INFO
*
* NOTE: The following equates are used to set the modes
* passed to the move_info call system service call.
*
***************************************************************
moveblkcmd          gequ     $0800             ; block move option
move_sinc_dinc      gequ     $0805             ; source increment, dest. increment

move_sinc_ddec      gequ     $0809             ; source increment, dest. decrement
move_sdec_dinc      gequ     $0806             ; source decrement, dest. increment
move_sdec_ddec      gequ     $080A             ; source decrement, dest. decrement

move_scon_dcon      gequ     $0800             ; source constant, dest. constant
move_sinc_dcon      gequ     $0801             ; source increment, dest. constant
move_sdec_dcon      gequ     $0802             ; source decrement, dest. constant
move_scon_dinc      gequ     $0804             ; source constant, dest. increment
move_scon_ddec      gequ     $0808             ; source constant, dest. decrement


                    eject

; __7_____6_____5_____4_____3_____2_____1_____0___
;|     |     |     |     |     |     |     |     |
;|slot7 |slot6 |slot5 |slot4 |     |slot2 |slot1 |     |
;|intext|intext|intext|intext|  0  |intext|intext|  0  |
;|enable|enable|enable|enable|     |enable|enable|     |
;|_____|_____|_____|_____|_____|_____|_____|_____|
;                   ^^^^^  sltromsel byte ^^^^^
;
; sltromsel bits defined as follows
;         bit 7= 0 enables internal slot 7 -- 1 enables slot rom
;         bit 6= 0 enables internal slot 6 -- 1 enables slot rom
;         bit 5= 0 enables internal slot 5 -- 1 enables slot rom
;         bit 4= 0 enables internal slot 4 -- 1 enables slot rom
;         bit 3= must be 0
;         bit 2= 0 enables internal slot 2 -- 1 enables slot rom
;         bit 1= 0 enables internal slot 1 -- 1 enables slot rom
;         bit 0= must be 0 `

sltromsel           gequ     $00C02D           ;slot rom select
```

```
; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;|      | stop |      | stop | stop | stop | stop | stop |
;|  0   |i/o/lc|  0   |auxh-r|suprhr|hires2|hires1|txt pg|
;|      |shadow|      |shadow|shadow|shadow|shadow|shadow|
;|______|______|______|______|______|______|______|______|
;                     ^^^^^  shadow byte ^^^^^
;
; shadow bits defined as follows
;          bit 7= must write 0
;          bit 6= 1 to inhibit i/o and language card operation
;          bit 5= must write 0
;          bit 4= 1 to inhibit shadowing aux hi-res page
;          bit 3= 1 to inhibit shadowing 32k video buffer
;          bit 2= 1 to inhibit shadowing hires page 2
;          bit 1= 1 to inhibit shadowing hires page 1
;          bit 0= 1 to inhibit shadowing text pages

shadow                gequ     $00C035               ;shadow register

                      eject
```

```
; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;| slow/|      |      |shadow|slot 7|slot 6|slot 5|slot 4|
;| fast |  0   |  0   |in all|motor |motor | motor| motor|
;| speed|      |      | ram  |detect|detect|detect|detect|
;|______|______|______|______|______|______|______|______|
;                     ^^^^^  cyareg byte ^^^^^
;
; cyareg bits defined as follows
;        bit 7= 0=slow system speed -- 1=fast system speed
;        bit 6= must write 0
;        bit 5= must write 0
;        bit 4= shadow in all ram banks
;        bit 3= slot 7 disk motor on detect
;        bit 2= slot 6 disk motor on detect
;        bit 1= slot 5 disk motor on detect
;        bit 0= slot 4 disk motor on detect

cyareg              gequ     $00C036            ; speed and motor on detect

; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;| alzp | page2| ramrd|ramwrt| rdrom|lcbnk2|rombnk| intcx|
;|status|status|status|status|status|status|status|status|
;|______|______|______|______|______|______|______|______|
;                     ^^^^^  statereg status byte ^^^^^
;
; statereg bits defined as follows
;        bit 7= alzp status
;        bit 6= page2 status
;        bit 5= ramrd status
;        bit 4= ramwrt status
;        bit 3= rdrom status (read only ram/rom (0/1))
;
; important note:
;       do two reads to $c083 then change statereg
;       to change lcram/rom banks (0/1) and still
;       have the language card write enabled.
;
;        bit 2= lcbnk2 status 0=LC bank 0 - 1=LC bank 1
;        bit 1= rombank status
;        bit 0= intcxrom status

statereg            gequ     $00C068            ; state register

clrrom              gequ     $00CFFF            ; switch out $c8 roms

                    eject
```

```
****************************************************************
*
* EQUATES for the IWM require index of (n*16)
*
****************************************************************
phaseoff            gequ      $00C080             ; stepper phase off
phaseon             gequ      $00C081             ; stepper phase on

ph0off              gequ      $00C080             ; phase 0 off
ph0on               gequ      $00C081             ; phase 0 on
ph1off              gequ      $00C082             ; phase 1 off
ph1on               gequ      $00C083             ; phase 1 on
ph2off              gequ      $00C084             ; phase 2 off
ph2on               gequ      $00C085             ; phase 2 on
ph3off              gequ      $00C086             ; phase 3 off
ph3on               gequ      $00C087             ; phase 3 on

motoroff            gequ      $00C088             ; disk motor off
motoron             gequ      $00C089             ; disk motor on

drv0en              gequ      $00C08A             ; select drive 0
drv1en              gequ      $00C08B             ; select drive 1

q6l                 gequ      $00C08C             ; Q6 low
q6h                 gequ      $00C08D             ; Q6 high
q7l                 gequ      $00C08E             ; Q7 low
q7h                 gequ      $00C08F             ; Q7 high

emulstack           gequ      $010100             ; emulation mode stack pointer

                    eject
```

```
*****************************************************************
*
* The following equates are used to implement our hypothetical
* device driver.  They in no way reflect softswitches associated
* with any real hardware device.
*
*****************************************************************

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                 READY
* |___|___|___|___|___|___|___|___|
*   |   |___|___|___|___|___|___|__ Reserved
*   |______________________________ 1 = Device is ready
*
ready                gequ     $00C080

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                 CHAR
* |___|___|___|___|___|___|___|___|
*   |___|___|___|___|___|___|___|__ Character device data register
*
char                 gequ     $00C081

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                 CHAR_STATUS
* |___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |__ 0
*   |   |   |   |   |   |   |______ 1 = Interrupt in process
*   |   |   |   |   |   |__________ 0
*   |   |   |   |   |______________ 0
*   |   |   |   |__________________ 1 = Online
*   |   |   |______________________ 0
*   |   |__________________________ 0
*   |______________________________ 0
*
char_status          gequ     $00C082
```

```
*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                  CHAR_CONTROL
* |___|___|___|___|___|___|___|___|
*   |___|___|___|___|___|___|___|__ Character device control register
*
char_control         gequ     $00C083

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                  BLOCK_RDY
* |___|___|___|___|___|___|___|___|
*   |   |___|___|___|___|___|___|__ Reserved
*   |______________________________ 1 = Device is ready
*
block_rdy            gequ     $00C084

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                  BLOCK_DATA
* |___|___|___|___|___|___|___|___|
*   |___|___|___|___|___|___|___|__ Block device data register
*
block_data           gequ     $00C085

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                  BLOCK_STATUS
* |___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |__ 1 = Disk switched
*   |   |   |   |   |   |   |______ 1 = Interrupt in process
*   |   |   |   |   |   |__________ 1 = Write protected
*   |   |   |   |   |______________ 0
*   |   |   |   |__________________ 1 = Online
*   |   |   |______________________ 0
*   |   |__________________________ 0
*   |______________________________ 0
*
block_status         gequ     $00C086

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                  BLOCK_CONTROL
* |___|___|___|___|___|___|___|___|
*   |___|___|___|___|___|___|___|__ Block device control register
*
block_control        gequ     $00C087

                     eject
```

```
**************************************************************
*
* The following table is the header required for all supervisory
* drivers which consists of the following:
*
*                   Long     Entry pointer to supervisory driver
*                   Word     Supervisory ID Number
*                   Word     Supervisory Driver Version Number
*                   Word     Reserved
*                   Word     Reserved
*                   Word     Reserved
*                   Word     Reserved
*
**************************************************************
driver_data         data
sib                 entry
                    dc       i4'dispatch'        ; Supervisory driver entry pointer
                    dc       h'$A5C3'            ; Supervisory ID Number
                    dc       h'$010e'            ; Supervisory Version Number 0.01e
                    dc       i4'sib_name'        ; SIB Name Pointer
                    dc       i2'0'               ; SIB Reserved #1
                    dc       i2'0'               ; SIB Reserved #1
*
* Supervisory specific extensions to the SIB may be required by
* certain implementations of the supervisory driver.  These
* extensions are allowed.  The reserved fields in the current
* SIB structure are for Apple's internal use.  If your implementation
* of a supervisory driver requires additional fields in the SIB then
* you should extend the SIB beyond it's current definition.
*
* The SIB name string that follows is not an extension to the SIB, rather
* an optional name string that describes the SIB.
*
sib_name            dc       i2'5'
                    dc       c'MYSIB'


**************************************************************
*
* The following table is used to dispatch to functions withing
* the supervisory driver.
*
**************************************************************
dispatch_tbl        entry
                    dc       i2'startup-1'
                    dc       i2'shutdn-1'
                    dc       i2'sup_call-1'

                    end

                    eject
```

```
****************************************************************
*
* SUPERVISORY DRIVER MAIN ENTRY POINT: DISPATCH
*
* This is the main entry point for the supervisory driver.    The
* routine validates the call number prior to dispatching to the
* requested function.
*
* Call Number:       $0000         Function:     Startup
*                    $0001                       Shutdown
*                    $0002-$FFFF                 Supervisor Specific
*
* ENTRY: Call via 'JSL'
*               [<drvr_sib_ptr] = Points to SIB for supervisor being accessed
*               [<sup_parm_ptr] = Supervisory parameter list pointer
*               A Reg = Supervisory Driver Number
*               X Reg = Supervisory Call Number
*               Y Reg = Undefined
*               Dir Reg = GS/OS Direct Page
*               B Reg = Undefined
*               P Reg = N V M X D I Z C  E
*                       x x 0 0 0 0 x x  0
*
* EXIT:   Direct page = unchanged with the exception of <drvr_tran_cnt
*               A Reg = Error code
*               X Reg = Undefined
*               Y Reg = Undefined
*               Dir Reg = GS/OS Direct Page
*               B Reg = Same as entry
*               P Reg = N V M X D I Z C  E
*                       x x 0 0 0 0 x 0  0          No error occurred
*                       x x 0 0 0 0 x 1  0          Error occurred
*
```

```
******************************************************************
dispatch        start
                using   driver_data
                longa   on
                longi   on

                phb                     ; save environment
                phk
                plb
                txa
                cmp     #$0002          ; startup or shutdown?
                blt     command_ok      ; yes
                lda     #$0002          ; else all specific through one entry
command_ok      anop
                pea     func_ret-1      ; return address from function
                asl     a               ; make index to dispatch table
                tax
                lda     |dispatch_tbl,x
                pha                     ; push function address for dispatch
                rts                     ; rts dispatches to function
func_ret        anop
                plb
                rtl

                end

                eject
```

```
*****************************************************************
*
* SUPERVISORY DRIVER CALL:     STARTUP
*
* This routine must prepare the driver to accept all other driver
* calls.
*
* ENTRY: Call via 'JSR'
*                    [<drvr_sib_ptr] = Points to SIB for supervisor being accessed
*                    [<sup_parm_ptr] = Supervisory parameter list pointer
*                    A Reg = Supervisory Driver Number
*                    X Reg = Supervisory Call Number
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Undefined
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                    A Reg = Error code
*                    X Reg = Undefined
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Same as entry
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x 0  0       No error occurred
*                            x x 0 0 0 0 x 1  0       Error occurred
*
```

```
****************************************************************
startup          start
                 using    driver_data
                 longa    on
                 longi    on


                 lda      |device_count        ; has slot been found?
                 beq      search_loop          ; no, go search for it
*
* Check for the device.
*
device_loop
                 ; insert your code here.
                 bne      search_loop          ; if you can't find a device
                 inc      |device_count
*
* The supervisor may want to construct a list of devices by slot and
* unit number so that the devices may be claimed by a device driver
* that uses the supervisor driver.
*
                 ; insert your code here to build a device list.
                 bra      device_loop          ; loop to check for next device
*
* Always request the slot from the slot arbiter prior to scanning the $CnXX
* space for signature bytes when searching for hardware.  This provides a
* compatible method of requesting a slot should a method of dynamic slot
* switching be made available in the future.
*
search_loop
               lda       |startup_slot       ; request slot from slot arbiter
               jsl       dyn_slot_arbiter
               bcs       next_slot           ; if slot was not granted
               inc       |device_count
*
```

```
* If the slot was granted then use the current slot to search for signature
* bytes identifying your hardware.
*
                lda         |search_slot        ; create $Cn00 for signature search index
                and         #$0007
                ora         #$00C0
                xba
                tax                             ; X register = $Cn00
*
* Now search for signatures.
*
                ; insert your code here.
                beq         device_loop         ; if you find a device
next_slot
                dec         |startup_slot       ; point at next slot to check
                bne         search_loop         ; and check for hardware
                lda         |device_count       ; any devices?
                beq         no_start_device     ; if not, don't need supervisor in system
                lda         #no_error
                clc
                rts

no_start_device
                lda         #drvr_io_error      ; an error on startup forces supervisor
                sec                             ; to be purged.
                rts

                end

                eject
```

```
****************************************************************
*
* SUPERVISOR DRIVER CALL:  Supervisory Specific
*
* This entry point is dispatched to for all supervisory specific
* calls.  Our skeleton driver does not implement any functional
* calls.  Your own implementation may require seperate entries
* for each supervisory specific call or a single entry may be
* used where the supervisory function is defined by the parameters
* passed in the supervisory parameter list.
*
* ENTRY: Call via 'JSR'
*                   [<drvr_sib_ptr] = Points to SIB for supervisor being accessed
*                   [<sup_parm_ptr] = Supervisory parameter list pointer
*                   A Reg = Supervisory Driver Number
*                   X Reg = Supervisory Call Number
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Undefined
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                   A Reg = Error code
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as entry
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x 0  0        No error occurred
*                           x x 0 0 0 0 x 1  0        Error occurred
*
****************************************************************
sup_call            start
                    using     driver_data
                    longa     on
                    longi     on

                    lda       #no_error             ; and exit w/o error
                    clc
                    rts

                    end

                    eject
```

```
****************************************************************
*
* SUPERVISORY DRIVER CALL:     SHUTDOWN
*
* This call prepares the driver for shutdown.  This may include
* releasing any and all system resources that may have been
* aquired by a STARTUP call.
*
* ENTRY: Call via 'JSR'
*                   [<drvr_sib_ptr] = Points to the SIB for the supervisor being accessed
*                   [<sup_parm_ptr] = Supervisory parameter list pointer
*                   A Reg = Supervisory Driver Number
*                   X Reg = Supervisory Call Number
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Undefined
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                   <drvr_tran_cnt = Number of bytes transferred
*                   A Reg = Error code
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as entry
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x 0  0       No error occurred
*                           x x 0 0 0 0 x 1  0       Error occurred
*
****************************************************************
shutdn              start
                    using     driver_data
                    longa     on
                    longi     on

                    lda       #no_error
                    clc
                    rts

                    end
```

Device driver that calls a supervisory driver

This could be either a block driver or character driver But it does not access its device(s) directly; instead, it goes through a supervisory driver like the one just listed The driver code consists of eight parts, in this order:

- Equates
- Device-driver header
- Format option tables (3 of them, for 3 supported formatting options)
- Device information blocks (DIBs; 2 of them, for 2 supported devices)
- Tables for dispatching calls and passing parameters
- A main entry point to the driver
- Routines that handle the driver calls

The driver has routines to handle all standard driver calls, including the standard Status and Control subcalls. The main difference between this driver and the previously listed device drivers is that, for each call that access a device, the driver simply passes the information on to the supervisory driver; the supervisory driver actually handles the call. In general, a supervisory driver and its individual device drivers can allocate among themselves the tasks of handling device-access calls in any way they see fit; GS/OS imposes no restrictions.

```
                    65816     on
                    instime on
                    gen       on
                    symbol    on
                    absaddr on
                    align     256
*************************************************************
*
*                   Copyright (c) 1987, 1988
*                   Apple Computer, Inc.
*                   All rights reserved.
*
*************************************************************
*
*                   Driver Core Routines Version 0.06a01
*
* NOTE:             All driver files must be installed on the
*                   boot volume in the subdirectory "/SYSTEM/DRIVERS'.
*                   Additionally, the FileType for the driver file
*                   must be set to $00BB.  AuxType is also critical
*                   to the operating system recognizing the driver
*                   as a GS/OS device driver.  The AuxType is a long
*                   word which must have the upper word set to $0000.
*                   The most significant byte of the least significant
*                   word in the AuxType must be set to $01 to indicate
*                   an active GS/OS device driver or $81 to indicate
*                   an inactive GS/OS device driver.  The least
*                   significant byte of the least significant word
*                   of the AuxType field indicates the number of
*                   devices supported by the driver file.  This value
*                   should be analogous to the number of DIB's
*                   contained in the driver file.  GS/OS will only
*                   install the number of devices indicated in the
*                   AuxType field.
*
*                   GS/OS Device Driver:          FileType = $00BB
*                                                 AuxType  = $000001XX where:
*                                                             XX = number of devices.
*
*                   An AuxType of $00000108 indicates eight devices.  When
*                   building a device driver, the best way to set the
*                   FileType and AuxType is to use the Exerciser to get
*                   the current file info (GET_FILE_INFO), modify the
*                   FileType & AuxType and then SET_FILE_INFO.
*
*                   Note that this driver requires the presence of
*                   a supervisory driver with a supervisory ID of $A5C3.
*
```

```
**************************************************************
*
* REVISION HISTORY:
*
* mm/dd/yy Version By          Revision description
*
* 02/26/88 0.00e01 RBM         Started initial coding.
*
* 04/11/88 0.06a01 RBM         New startup.
*                              New shutdown.
*                              Additional control and status calls.
*                              Removed valid access parsing performed by dispatcher.
*
**************************************************************


                    eject
**************************************************************
*
* The following are direct page equates on the GS/OS
* direct page for driver usage.
*
**************************************************************

drvr_dev_num        gequ     $00                 ; (w)  device number
drvr_call_num       gequ     drvr_dev_num+2      ; (w)  call number
drvr_buf_ptr        gequ     drvr_call_num+2     ; (lw) buffer pointer
drvr_slist_ptr      gequ     drvr_call_num+2     ; (lw) buffer pointer
drvr_clist_ptr      gequ     drvr_call_num+2     ; (lw) buffer pointer
dev_id_ref          gequ     drvr_buf_ptr        ; (w)  indirect device ID
drvr_req_cnt        gequ     drvr_buf_ptr+4      ; (lw) request count
drvr_tran_cnt       gequ     drvr_req_cnt+4      ; (lw) transfer count
drvr_blk_num        gequ     drvr_tran_cnt+4     ; (lw) block number
drvr_blk_size       gequ     drvr_blk_num+4      ; (w)  block size
drvr_fst_num        gequ     drvr_blk_size+2     ; (w)  File System Translator Number
drvr_stat_code      gequ     drvr_fst_num        ; (w)  status code for status call
drvr_ctrl_code      gequ     drvr_fst_num        ; (w)  control code for control call
drvr_vol_id         gequ     drvr_fst_num+2      ; (w)  Driver Volume ID Number
drvr_cache          gequ     drvr_vol_id+2       ; (w)  Cache Priority
drvr_cach_ptr       gequ     drvr_cache+2        ; (lw) pointer to cached block
drvr_dib_ptr        gequ     drvr_cach_ptr+4     ; (lw) pointer to active DIB

sib_ptr             gequ     $0074               ; (lw) pointer to active SIB
sup_parm_ptr        gequ     sib_ptr+4           ; (lw) pointer to supervisor parameters


                    eject
```

```
***************************************************************
*
* The following are equates for driver command types.
*
***************************************************************

drvr_startup        gequ      $0000               ; driver startup command
drvr_open           gequ      $0001               ; driver open command
drvr_read           gequ      $0002               ; driver read command
drvr_write          gequ      $0003               ; driver write command
drvr_close          gequ      $0004               ; driver close command
drvr_status         gequ      $0005               ; driver status command
drvr_control        gequ      $0006               ; driver control command
drvr_flush          gequ      $0007               ; driver flush command
drvr_shutdn         gequ      $0008               ; driver shutdown command
max_command         gequ      $0009               ; commands $0009 - $ffff undefined

drvr_dev_stat       gequ      $0000               ; status code: return device status
drvr_ctrl_stat      gequ      $0001               ; status code: return control params
drvr_get_wait       gequ      $0002               ; status code: get wait/no wait mode
drvr_get_format     gequ      $0003               ; status code: get format options

drvr_reset          gequ      $0000               ; control code: reset device
drvr_format         gequ      $0001               ; control code: format device
drvr_eject          gequ      $0002               ; control code: eject media
drvr_set_ctrl       gequ      $0003               ; control code: set control params
drvr_set_wait       gequ      $0004               ; control code: set wait/no wait mode
drvr_set_format     gequ      $0005               ; control code: set format options
drvr_set_ptn        gequ      $0006               ; control code: set partition owner
drvr_arm            gequ      $0007               ; control code: arm interrupt signal
drvr_disarm         gequ      $0007               ; control code: arm interrupt signal

                    eject
```

```
**************************************************************
*
* The following are equates for GS/OS error codes.
*
*************************************************************

no_error            gequ      $0000               ; no error has occurred
dev_not_found       gequ      $0010               ; device not found
invalid_dev_num     gequ      $0011               ; invalid device number
drvr_bad_req        gequ      $0020               ; bad request or command
drvr_bad_code       gequ      $0021               ; bad control or status code
drvr_bad_parm       gequ      $0022               ; bad call parameter
drvr_not_open       gequ      $0023               ; character device not open
drvr_prior_open     gequ      $0024               ; character device already open
irq_table_full      gequ      $0025               ; interrupt table full
drvr_no_resrc       gequ      $0026               ; resources not available
drvr_io_error       gequ      $0027               ; I/O error
drvr_no_dev         gequ      $0028               ; device not connected
drvr_busy           gequ      $0029               ; call aborted, driver is busy
drvr_wr_prot        gequ      $002B               ; device is write protected
drvr_bad_count      gequ      $002C               ; invalid byte count
drvr_bad_block      gequ      $002D               ; invalid block address
drvr_disk_sw        gequ      $002E               ; disk has been switched
drvr_off_line       gequ      $002F               ; device off line / no media present
invalid_access      gequ      $004E               ; access not allowed
parm_range_err      gequ      $0053               ; parameter out of range
out_of_mem          gequ      $0054               ; out of memory
dup_volume          gequ      $0057               ; duplicate volume name
not_block_dev       gequ      $0058               ; not a block device
stack_overflow      gequ      $005F               ; too many applications on stack
data_unavail        gequ      $0060               ; data unavailable

                    eject
```

```
***************************************************************
*
* The following are equates for the DIB.
*
***************************************************************
link_ptr            gequ     $0000                ; (lw) pointer to next DIB
entry_ptr           gequ     $0004                ; (lw) pointer to driver
dev_char            gequ     $0008                ; (w)  device characteristics
blk_cnt             gequ     $000A                ; (lw) number of blocks
dev_name            gequ     $000E                ; (32) count and ascii name (pstring)
slot_num            gequ     $002E                ; (w)  slot number
unit_num            gequ     $0030                ; (w)  unit number
ver_num             gequ     $0032                ; (w)  version number
dev_id_num          gequ     $0034                ; (w)  device ID number (ICON ref#)
head_link           gequ     $0036                ; (w)  backward device link
forward_link        gequ     $0038                ; (w)  forward device link
link_dib_ptr        gequ     $003A                ; (lw) dib reserved field #1
dib_dev_num         gequ     $003E                ; (w)  Device number of this device
*
* The following equate(s) are for drive specific extensions to the DIB.
* Parameters that are extended to the manditory DIB parameters are not
* accessable by GS/OS or the application but may be used within a driver
* as needed.
*
driver_unit         gequ     $0040                ; (w)  driver's internal DIB data
my_slot16           gequ     $0042                ; (w)  driver's slot * 16

                    eject
***************************************************************
*
* System Service Table Equates:
*
* NOTE: Only those system service calls that might be used
* by a device driver are listed here.  For a more complete
* list of system service calls and explanations of each call
* consult the system service call ERS.
*
***************************************************************
dev_dispatcher      gequ     $01FC00              ; dev_dispatch
cache_find_blk      gequ     $01FC04              ; cash_find
cache_add_blk       gequ     $01FC08              ; cash_add
cache_del_blk       gequ     $01FC14              ; cash_delete
cache_del_vol       gequ     $01FC18              ; cash_del_vol
set_sys_speed       gequ     $01FC50              ; set system speed
move_info           gequ     $01FC70              ; gs_move_block
set_disksw          gequ     $01FC90              ; set disksw and call swapout/delvol
sup_drvr_disp       gequ     $01FCA4              ; supervisor dispatcher
install_driver      gequ     $01FCA8              ; dynamic driver installation
dyn_slot_arbiter    gequ     $01FCBC              ; dynamic slot arbiter

                    eject
```

```
****************************************************************
*
* MOVE_INFO
*
* NOTE: The following equates are used to set the modes
* passed to the move_info call system service call.
*
****************************************************************
moveblkcmd          gequ     $0800               ; block move option
move_sinc_dinc      gequ     $0805               ; souce increment, dest. increment

move_sinc_ddec      gequ     $0809               ; souce increment, dest. decrement
move_sdec_dinc      gequ     $0806               ; souce decrement, dest. increment
move_sdec_ddec      gequ     $080A               ; souce decrement, dest. decrement

move_scon_dcon      gequ     $0800               ; souce constant, dest. constant
move_sinc_dcon      gequ     $0801               ; souce increment, dest. constant
move_sdec_dcon      gequ     $0802               ; souce decrement, dest. constant
move_scon_dinc      gequ     $0804               ; souce constant, dest. increment
move_scon_ddec      gequ     $0808               ; souce constant, dest. decrement

                    eject

; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;|slot7 |slot6 |slot5 |slot4 |      |slot2 |slot1 |      |
;|intext|intext|intext|intext|  0   |intext|intext|  0   |
;|enable|enable|enable|enable|      |enable|enable|      |
;|______|______|______|______|______|______|______|______|
;                    ^^^^^  sltromsel byte ^^^^^
;
; sltromsel bits defined as follows
;         bit 7= 0 enables internal slot 7 -- 1 enables slot rom
;         bit 6= 0 enables internal slot 6 --   enables slot rom
;         bit 5= 0 enables internal slot 5 -    enables slot rom
;         bit 4= 0 enables internal slot 4 -- 1 enables slot rom
;         bit 3= must be 0
;         bit 2= 0 enables internal slot 2 -- 1 enables slot rom
;         bit 1= 0 enables internal slot 1 -- 1 enables slot rom
;         bit 0= must be 0

sltromsel        gequ        $00C02D             ;slot rom select

; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;|      | stop |      | stop | stop | stop | stop | stop |
;|  0   |i/o/lc|  0   |auxh-r|suprhr|hires2|hires1|txt pg|
;|      |shadow|      |shadow|shadow|shadow|shadow|shadow|
;|______|______|______|______|______|______|______|______|
;                    ^^^^^  shadow byte ^^^^^
;
; shadow bits defined as follows
;         bit 7= must write 0
```

```
;         bit 6= 1 to inhibit i/o and language card operation
;         bit 5= must write 0
;         bit 4= 1 to inhibit shadowing aux hi-res page
;         bit 3= 1 to inhibit shadowing 32k video buffer
;         bit 2= 1 to inhibit shadowing hires page 2
;         bit 1= 1 to inhibit shadowing hires page 1
;         bit 0= 1 to inhibit shadowing text pages

shadow    gequ      $00C035      ; shadow register

                    eject

; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;| slow/|      |      |shadow|slot 7|slot 6|slot 5|slot 4|
;| fast |  0   |  0   |in all|motor |motor | motor| motor|
;| speed|      |      | ram  |detect|detect|detect|detect|
;|______|______|______|______|______|______|______|______|
;                     ^^^^^  cyareg byte ^^^^^
;
; cyareg bits defined as follows
;         bit 7= 0=slow system speed -- 1=fast system speed
;         bit 6= must write 0
;         bit 5= must write 0
;         bit 4= shadow in all ram banks
;         bit 3= slot 7 disk motor on detect
;         bit 2= slot 6 disk motor on detect
;         bit 1= slot 5 disk motor on detect
;         bit 0= slot 4 disk motor on detect

cyareg    gequ      $00C036      ;speed and motor on detect

; __7______6______5______4______3______2______1______0___
;|      |      |      |      |      |      |      |      |
;| alzp | page2| ramrd|ramwrt| rdrom|lcbnk2|rombnk| intcx|
;|status|status|status|status|status|status|status|status|
;|______|______|______|______|______|______|______|______|
;                     ^^^^^  statereg status byte ^^^^^
;
; statereg bits defined as follows
;         bit 7= alzp status
;         bit 6= page2 status
;         bit 5= ramrd status
;         bit 4= ramwrt status
;         bit 3= rdrom status (read only ram/rom (0/1))
;
; important note:
;        do two reads to $c083 then change statereg
;        to change lcram/rom banks (0/1) and still
;        have the language card write enabled.
;
;         bit 2= lcbnk2 status 0=LC bank 0 - 1=LC bank 1
;         bit 1= rombank status
```

```
;          bit 0= intcxrom status

statereg            gequ      $00C068              ; state register

clrrom              gequ      $00CFFF              ; switch out $c8 roms

                    eject

*****************************************************************
*
* EQUATES for the IWM require index of (n*16)
*
*****************************************************************
phaseoff            gequ      $00C080              ; stepper phase off
phaseon             gequ      $00C081              ; stepper phase on

ph0off              gequ      $00C080              ; phase 0 off
ph0on               gequ      $00C081              ; phase 0 on
ph1off              gequ      $00C082              ; phase 1 off
ph1on               gequ      $00C083              ; phase 1 on
ph2off              gequ      $00C084              ; phase 2 off
ph2on               gequ      $00C085              ; phase 2 on
ph3off              gequ      $00C086              ; phase 3 off
ph3on               gequ      $00C087              ; phase 3 on

motoroff            gequ      $00C088              ; disk motor off
motoron             gequ      $00C089              ; disk motor on

drv0en              gequ      $00C08A              ; select drive 0
drv1en              gequ      $00C08B              ; select drive 1

q6l                 gequ      $00C08C              ; Q6 low
q6h                 gequ      $00C08D              ; Q6 high
q7l                 gequ      $00C08E              ; Q7 low
q7h                 gequ      $00C08F              ; Q7 high

emulstack           gequ      $010100              ; emulation mode stack pointer

                    eject
*****************************************************************
*
* The following are equates for the SIB.
*
*****************************************************************
sup_entry_ptr       gequ      $0000                ; (lw) pointer to driver
sup_id              gequ      $0004                ; (w)  supervisory driver ID number
sup_version         gequ      $0006                ; (w)  supervisory driver version
sib_res_1           gequ      $0008                ; (w)  sib reserved #1
sib_res_2           gequ      $000A                ; (w)  sib reserved #2
sib_res_3           gequ      $000C                ; (w)  sib reserved #3
sib_res_4           gequ      $000E                ; (w)  sib reserved #4

                    eject
```

```
****************************************************************
*
* The following equates are used to implement our hypothetical
* device driver.  They in no way reflect softswitches associated
* with any real hardware device.
*
****************************************************************

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                 READY
* |___|___|___|___|___|___|___|___|
*   |   |___|___|___|___|___|___|__ Reserved
*   |______________________________ 1 = Device is ready
*
ready               gequ      $00C080

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                 CHAR
* |___|___|___|___|___|___|___|___|
*   |___|___|___|___|___|___|___|__ Character device data register
*
char                gequ      $00C081

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                 CHAR_STATUS
* |___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |__ 0
*   |   |   |   |   |   |   |______ 1 = Interrupt in process
*   |   |   |   |   |   |__________ 0
*   |   |   |   |   |______________ 0
*   |   |   |   |__________________ 1 = Online
*   |   |   |______________________ 0
*   |   |__________________________ 0
*   |______________________________ 0
*
char_status         gequ      $00C082

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                 CHAR_CONTROL
* |___|___|___|___|___|___|___|___|
*   |___|___|___|___|___|___|___|__ Character device control register
*
char_control        gequ      $00C083

*  ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |
* | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                 BLOCK_RDY
* |___|___|___|___|___|___|___|___|
*   |   |___|___|___|___|___|___|__ Reserved
```

```
*    |_____________________________ 1 = Device is ready
*
block_rdy            gequ      $00C084

*   ___ ___ ___ ___ ___ ___ ___ ___
*  |   |   |   |   |   |   |   |   |
*  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                    BLOCK_DATA
*  |___|___|___|___|___|___|___|___|
*    |___|___|___|___|___|___|___|__ Block device data register
*
block_data           gequ      $00C085

*   ___ ___ ___ ___ ___ ___ ___ ___
*  |   |   |   |   |   |   |   |   |
*  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                    BLOCK_STATUS
*  |___|___|___|___|___|___|___|___|
*    |   |   |   |   |   |   |   |__ 1 = Disk switched
*    |   |   |   |   |   |   |______ 1 = Interrupt in process
*    |   |   |   |   |   |__________ 1 = Write protected
*    |   |   |   |   |______________ 0
*    |   |   |   |__________________ 1 = Online
*    |   |   |______________________ 0
*    |   |__________________________ 0
*    |______________________________ 0
*
block_status         gequ      $00C086

*   ___ ___ ___ ___ ___ ___ ___ ___
*  |   |   |   |   |   |   |   |   |
*  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                    BLOCK_CONTROL
*  |___|___|___|___|___|___|___|___|
*    |___|___|___|___|___|___|___|__ Block device control register
*
block_control        gequ      $00C087

                     eject
*****************************************************************
*
* The following table is the header required for all loaded
* drivers which consists of the following:
*
*                    Word      Offset from start to 1st DIB
*                    Word      Number of DIBs
*                    Word      Offset from start to 1st configuration list
*                    Word      Offset from start to 2nd configuration list
*                    etc.
*
*****************************************************************
driver_data          data
here                 entry
                     dc        i2'dib_1-here'         ; offset to 1st DIB
                     dc        i2'1'                  ; number of devices
                     dc        i2'conf1-here'         ; offset to 1st configuration list
```

```
*
* The following are the driver configuration parameter lists.
*
confl               dc        i2'0'                  ; 0 bytes in parameter list
defaultl            dc        i2'0'                  ; 0 bytes in default list

                    eject
***************************************************************
*
* The following are tables of format options for each device.
* The format option tables have the following structure:
*
*                   Word      Number of entries in list
*                   Word      Display count (number of head links)
*                   Word      Recommended default option
*                   Word      Option that current online media is formatted with
*                   Entries   16 bytes per entry in the format list
*
* The twenty byte structure for each entry in the format list
* is as follows:
*
*                   Word      Media variables reference number
*                   Word      Link to reference number n.
*                   Word      Flags / Format environment
*                   Long      Number of blocks supported by device
*                   Word      Block size
*                   Word      Interleave factor
*                   Long      Number of bytes defined by flag
*
* Bit definition within the flags word is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |___|__ Format
*   |   |   |   |   |   |   |   |   |   |   |   |   |___|_________ Flags
*   |___|___|___|___|___|___|___|___|___|___|___|_________________ Reserved
*
* Format Bit Definition: 00   Universal format
*                        01   Apple Format
*                        02   NonApple Format
*                        11   Not Valid
*
* Flag Bit Definition:          00   Size is in bytes
*                        01   Size is in Kb
*                        02   Size is in Mb
*                        11   Size is in Gb
*
***************************************************************
format_tbl          entry
                    dc        i2'3'                  ; number of entries
                    dc        i2'2'                  ; number of displayed entries
                    dc        i2'1'                  ; recommended option is 1
```

```
                    dc          i2'1'                   ; current media formatted w/option 1
format1_entry1      anop
                    dc          i2'1'                   ; RefNum
                    dc          i2'2'                   ; LinkRef
                    dc          i2'4'                   ; univeral format / size in kb
                    dc          i4'1600'                ; block count
                    dc          i2'512'                 ; block size
                    dc          i2'4'                   ; interleave factor
                    dc          i2'800'                 ; media size is 800kb
format1_entry2      anop
                    dc          i2'2'                   ; reference number 1
                    dc          i2'0'                   ; LinkRef
                    dc          i2'4'                   ; univeral format / size in kb
                    dc          i4'1600'                ; block count
                    dc          i2'512'                 ; block size
                    dc          i2'2'                   ; interleave factor
                    dc          i2'800'                 ; media size is 800kb
format1_entry3      anop
                    dc          i2'3'                   ; reference number 1
                    dc          i2'0'                   ; LinkRef
                    dc          i2'4'                   ; univeral format / size in kb
                    dc          i4'800'                 ; block count
                    dc          i2'524'                 ; block size
                    dc          i2'4'                   ; interleave factor
                    dc          i2'400'                 ; media size is 400kb

                    eject
*****************************************************************
*
*                   Link Pointer
*                   Entry Pointer
*                   Device Characteristics
*    ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
*   | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
*   |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |___|_ RESERVED
*     |   |   |   |   |   |   |   |   |   |   |   |   |   |_______ REMOVABLE
*     |   |   |   |   |   |   |   |   |   |   |   |   |___________ FORMAT
*     |   |   |   |   |   |   |   |   |   |   |   |_______________ RESERVED
*     |   |   |   |   |   |   |   |   |   |   |___________________ READ
*     |   |   |   |   |   |   |   |   |   |_______________________ WRITE
*     |   |   |   |   |   |   |   |   |___________________________ BLOCK DEVICE
*     |   |   |   |   |   |   |___|_______________________________ SPEED GROUP
*     |   |   |   |   |___|_______________________________________ RESERVED
*     |   |   |   |___________________________________________________ BUSY
*     |   |   |_______________________________________________________ LINKED
*     |   |___________________________________________________________ GENERATED
*     |_______________________________________________________________ RAM/ROM DEV
*
*                   Block Count
*                   Device Name
*                   Slot Number
```

```
*                       Unit Number
*                       Device ID Number
*                       Head Device Link
*                       Forward Device Link
*                       Reserved Word
*                       Reserved Word
*                       DIB device number
*
*****************************************************************
dib_1                   entry
                        dc      i4'dib_2'           ; link pointer to second DIB
                        dc      i4'dispatch'        ; entry pointer
                        dc      h'EC 00'            ; characteristics
                        dc      i4'280'             ; block count
                        dc      i1'11'              ; device name (length & 32 bytes ascii)
                        dc      c'SUPERVISORY'
                        dc      20h'20'
                        dc      h'0F 00'            ; slot # (valid only after startup)
                        dc      h'03 00'            ; unit # (valid only after startup)
                        dc      H'1e 00'            ; version # 0.01e
                        dc      h'FF 01'            ; device ID # (valid only after startup)
                        dc      i2'0'               ; head device link
                        dc      i2'0'               ; forward device link
                        dc      i2'0'               ; Reserved
                        dc      i2'0'               ; Reserved
                        dc      i2'0'               ; dib device number
                        dc      i2'0'               ; drivers internal device number
                        dc      i2'0'               ; slot * 16

                        eject
dib_2                   entry
                        dc      i4'0'               ; link pointer to next DIB if any
                        dc      i4'dispatch'        ; entry pointer
                        dc      h'EC 00'            ; characteristics
                        dc      i4'280'             ; block count
                        dc      i1'11'              ; device name (length & 32 bytes ascii)
                        dc      c'SUPERVISORY'
                        dc      20h'20'
                        dc      h'0F 00'            ; slot # (valid only after startup)
                        dc      h'04 00'            ; unit # (valid only after startup)
                        dc      H'1e 00'            ; version # 0.01e
                        dc      h'FF 01'            ; device ID # (valid only after startup)
                        dc      i2'0'               ; head device link
                        dc      i2'0'               ; forward device link
                        dc      i2'0'               ; Reserved
                        dc      i2'0'               ; Reserved
                        dc      i2'0'               ; dib device number
                        dc      i2'0'               ; drivers internal device number
                        dc      i2'0'               ; slot * 16

                        eject
*****************************************************************
*
```

```
* The following table is used to dispatch to GS/OS driver
* functions.
*
****************************************************************
dispatch_table      entry
                    dc       i2'startup-1'
                    dc       i2'open-1'
                    dc       i2'read-1'
                    dc       i2'write-1'
                    dc       i2'close-1'
                    dc       i2'status-1'
                    dc       i2'control-1'
                    dc       i2'flush-1'
                    dc       i2'shutdn-1'

status_table        entry
                    dc       i2'dev_stat-1'
                    dc       i2'get_ctrl-1'
                    dc       i2'get_wait-1'
                    dc       i2'get_format-1'
                    dc       i2'get_partn_map-1'

control_table       entry
                    dc       i2'dev_reset-1'
                    dc       i2'format-1'
                    dc       i2'media_eject-1'
                    dc       i2'set_ctrl-1'
                    dc       i2'set_wait-1'
                    dc       i2'set_format-1'
                    dc       i2'set_partn-1'
                    dc       i2'arm_signal-1'
                    dc       i2'disarm_signal-1'
                    dc       i2'set_partn_map-1'

status_flag         entry
                    dc       i2'0'                ; flag for unit #

                    eject
****************************************************************
*
* The following table contains the open status for each device
* supported by this driver.
*
****************************************************************
open_table          entry
                    dc       i2'0'                ; open state for DIB 1 device

                    eject
****************************************************************
*
* The following table contains the device status for each
* device supported by this driver.
*
```

```
* Encoding of status for a character device is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |__ OPEN
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |______ INTERRUPT
*   |   |   |   |   |   |   |   |   |   |   |   |___|___|__________ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |__________________ BUSY
*   |___|___|___|___|___|___|___|___|___|___|____________________ 0 RESERVED
*
* Encoding of status for a block device is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |__ DISK SW
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |______ INTERRUPT
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |__________ WRITE PROT
*   |   |   |   |   |   |   |   |   |   |   |   |   |______________ 0 RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |__________________ ONLINE
*   |___|___|___|___|___|___|___|___|___|___|______________________ 0 RESERVED
*
****************************************************************
dstat_tbl           entry
                    dc        i2'0'               ; device general status word
                    dc        i4'1600'            ; device block count

                    eject
****************************************************************
*
* The following table is used to return the configuration list for
* each device supported by this driver.
*
****************************************************************
clist_tbl           entry
                    dc        i2'confl'           ; pointer to configuration list #1

****************************************************************
*
* The following table is used to return the wait mode for
* each device supported by this driver.
*
****************************************************************
wait_mode_tbl       entry
                    dc        i2'0'               ; unit 1 wait mode

****************************************************************
*
* The following table is used to set the current format
* option for each device supported by this driver.
*
****************************************************************
```

```
format_mode         entry
                    dc        i2'0'                  ; unit 1 format mode

*****************************************************************
*
* The following table is used by the startup call when setting
* parameters in the DIB.  Slot number, Unit number and Device
* ID number are valid only after startup.
*
*****************************************************************
startup_slot        dc        i2'7'                  ; initial slot to search for
startup_unit        dc        i2'1'                  ; initial unit to search for
sup_num             dc        i2'0'                  ; supervisory driver number

*****************************************************************
*
* The following equates are general workspace used by the driver.
*
*****************************************************************
retry_count         dc        i2'0'                  ; retry count
startup_count       dc        i2'0'

                    end

                    eject
*****************************************************************
*
* DRIVER MAIN ENTRY POINT: DISPATCH
*
* This is the main entry point for the device driver.  The
* routine validates the call number prior to dispatching to
* the requested function.
*
* Call Number:      $0000     Function:     Startup
*                   $0001                   Open
*                   $0002                   Read
*                   $0003                   Write
*                   $0004                   Close
*                   $0005                   Status
*                   $0006                   Control
*                   $0007                   Flush
*                   $0008                   Shutdown
*                   $0009-$FFFF             Reserved
*
* ENTRY: Call via 'JSL'
*                   [<drvr_dib_ptr] = Points to DIB for device being accessed
*                   <drvr_dev_num = Device number of device being accessed
*                   <drvr_call_num = Call number
*                   A Reg = Call Number
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Undefined
```

```
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
* EXIT:    Direct page = unchanged with the exception of <drvr_tran_cnt
*                   A Reg = Error code
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as entry
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x 0  0       No error occurred
*                           x x 0 0 0 0 x 1  0       Error occurred
*
*****************************************************************
dispatch            start
                    using     driver_data
                    longa     on
                    longi     on

                    phb                             ; save environment
                    phk
                    plb
                    cmp       #max_command          ; is it a legal command?
                    bge       illegal_req           ; no
                    tay       ; save command #
                    ldx       #$0000
save_parms          anop
                    lda       <drvr_dev_num,x       ; save GS/OS call parameters
                    pha
                    lda       <drvr_blk_num,x
                    pha
                    inx
                    inx
                    cpx       #$000C                ; up to but not including DRVR_TRAN_CNT
                    bne       save_parms

                    tya       ; restore command #
                    pea       func_ret-1            ; return address from function
                    asl       a                     ; make index to dispatch table
                    tax
                    lda       |dispatch_table,x
                    pha                             ; push function address for dispatch
                    rts                             ; rts dispatches to function
func_ret            anop
                    tay                             ; save error code

                    ldx       #$000A                ; number of words to restore
restore_parms       anop
                    pla                             ; restore GS/OS call parameters
                    sta       <drvr_blk_num,x
                    pla
                    sta       <drvr_dev_num,x
                    dex
```

```
                    dex
                    bpl       restore_parms
                    plb
                    bcs       gen_exit              ; force error code 0 if flag cleared
                    ldy       #no_error
gen_exit            anop
                    tya                             ; restore error code
                    rtl
*
* Received an illegal request. Return with an error.
*
illegal_req         anop
                    plb                             ; restore environement
                    lda       #drvr_bad_req         ; set error
                    sec
                    rtl

                    end

                    eject
****************************************************************
*
* DRIVER CALL:      STARTUP
*
* This routine must prepare the driver to accept all other driver
* calls.
*
* ENTRY: Call via 'JSR'
*                   [<drvr_dib_ptr] = Points to the DIB for the device being accessed
*                   <drvr_dev_num = Device number of device being accessed
*                   <drvr_call_num = Call number
*                   <drvr_tran_cnt = $00000000
*                   A Reg = Call Number
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as program bank
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                   A Reg = Error code
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as entry
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x 0  0      No error occurred
*                           x x 0 0 0 0 x 1  0      Error occurred
*
```

```
*****************************************************************
startup            start
                   using    driver_data
                   longa    on
                   longi    on
*
* This driver requires the use of a supervisory driver with a
* supervisory ID of $A5C3.    The startup call attempts to aquire
* the supervisory driver number for the supervisory driver with
* the supervisory ID $A5C3.  If a supervisory driver number can
* be aquired then the supervisory number is saved for subsequent
* dispatches to the supervisory driver.  If no supervisory driver
* number is returned then the driver cannot startup and will
* return an error to the device dispatcher.  This will force the
* driver to be purged from the device list and memory.
*
                   lda      #$0000             ; Supervisory dispatcher
                   ldx      #$0000             ; Supervisory call number
                   ldy      #$A5C3             ; Supervisory ID number
                   jsl      sup_drvr_disp      ; get supervisory driver number
                   bcc      found_sup          ; if found supervisory driver
                   rts
found_sup          anop
                   txa                         ; save our supervisory driver number
                   sta      >sup_num
*
* Now any device initialization that must occur should be executed
* through the supervisory driver dispatcher.
*
                inc        |startup_count      ; keep track of how many drvrs started
                lda        #no_error
                clc
                rts

                end

                eject
*****************************************************************
*
* DRIVER CALL:       OPEN
*
* ENTRY: via a 'JSR'
*                    <drvr_dev_num = Device Number of current device being accessed
*                    <drvr_tran_cnt = $00000000
*                    A Reg = Call Number
*                    X Reg = Undefined
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Undefined
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
```

```
*                    A Reg = Error code
*                    X Reg = Undefined
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Same as entry
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x 0  0        No error occurred
*                            x x 0 0 0 0 x 1  0        Error occurred
*
*****************************************************************
open                 start
                     using      driver_data
                     longa      on
                     longi      on
*
* Pass on the standard GS/OS call parameters to the supervisory driver.
*
                tdc
                sta        <sup_parm_ptr          ; set pointer to supervisor parameters
                stz        <sup_parm_ptr+2
                lda        |sup_num               ; get supervisor driver number
                ldx        #$0002                 ; supervisor specific call
                jsl        sup_drvr_disp          ; call supervisory driver
                rts

                end

                eject
*****************************************************************
*
* DRIVER CALL:       READ
*
* This call executes a read via a supervisory driver.  Caching
* support is provided within the supervisory driver.
*
* ENTRY: via a 'JSR'
*                    <drvr_dev_num = Device Number of current device being accessed
*                    <drvr_buf_ptr = Pointer to I/O buffer
*                    <drvr_blk_num = Initial block number
*                    <drvr_req_cnt = Number of bytes to be transferred
*                    <drvr_blk_size = Size of block to be accessed
*                    <drvr_tran_cnt = $00000000
*                    A Reg = Call Number
*                    X Reg = Undefined
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Undefined
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                    <drvr_tran_cnt = Number of bytes transferred
*                    A Reg = Error code
```

```
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as entry
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x 0  0      if no error
*                           x x 0 0 0 0 x 1  0      if error
*
****************************************************************
read                start
                    using    driver_data
                    longa    on
                    longi    on
*
* Pass on the standard GS/OS call parameters to the supervisory driver.
*
                tdc
                sta     <sup_parm_ptr           ; set pointer to supervisor parameters
                stz     <sup_parm_ptr+2
                lda     |sup_num                ; get supervisor driver number
                ldx     #$0002                  ; supervisor specific call
                jsl     sup_drvr_disp           ; call supervisory driver
                rts

                end

                eject
****************************************************************
*
* DRIVER CALL:      WRITE
*
* This call executes a write via the supervisory driver.  Caching
* support is provided by the supervisory driver.
*
* ENTRY: via a 'JSR'
*                   <drvr_dev_num = Device Number of current device being accessed
*                   <drvr_buf_ptr = Pointer to I/O buffer
*                   <drvr_blk_num = Initial block number
*                   <drvr_req_cnt = Number of bytes to be transferred
*                   <drvr_blk_size = Size of block to be accessed
*                   <drvr_tran_cnt = $00000000
*                   A Reg = Call Number
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Undefined
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                   <drvr_tran_cnt = Number of bytes transferred
*                   A Reg = Error code
*                   X Reg = Undefined
```

```
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Same as entry
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 x x x 0  0      No error occurred
*                            x x 0 0 x x x 1  0      Error occurred
*
*****************************************************************
write           start
                using       driver_data
                longa       on
                longi       on
*
* Pass on the standard GS/OS call parameters to the supervisory driver.
*
                tdc
                sta         <sup_parm_ptr         ; set pointer to supervisor parameters
                stz         <sup_parm_ptr+2
                lda         |sup_num              ; get supervisor driver number
                ldx         #$0002                ; supervisor specific call
                jsl         sup_drvr_disp         ; call supervisory driver
                rts

                end

                eject
*****************************************************************
*
* DRIVER CALL:       CLOSE
*
* ENTRY: via a 'JSR'
*                    <drvr_dev_num = Device Number of current device being accessed
*                    <drvr_tran_cnt = $00000000
*                    A Reg = Call Number
*                    X Reg = Undefined
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Undefined
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                    A Reg = Error code
*                    X Reg = Undefined
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Same as entry
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x 0  0      No error occurred
*                            x x 0 0 0 0 x 1  0      Error occurred
*
*****************************************************************
close                start
```

```
                using     driver_data
                longa     on
                longi     on
*
* Pass on the standard GS/OS call parameters to the supervisory driver.
*
             tdc
             sta        <sup_parm_ptr          ; set pointer to supervisor parameters
             stz        <sup_parm_ptr+2
             lda        |sup_num               ; get supervisor driver number
             ldx        #$0002                 ; supervisor specific call
             jsl        sup_drvr_disp          ; call supervisory driver
             rts

             end

             eject
****************************************************************
*
* DRIVER CALL:      STATUS
*
* This routine supports all the standard device status calls.
* Any status call which is able to detect an OFFLINE or DISK
* SWITCHED condition should call the system service routine
* SET_DISKSW.  OFFLINE and DISKSW are conditions and not errors
* when detected by a status call and should only be returned as
* conditions in the status list.
*
* Status Code:      $0000     Return Device Status
*                   $0001     Return Control Parameters
*                   $0002     Return Wait/No Wait Mode
*                   $0003     Return Format Options
*
* ENTRY: via a 'JSR'
*                   <drvr_dev_num = Device Number of current device being accessed
*                   <drvr_clist_ptr = Pointer to control list
*                   <drvr_ctrl_code = Control code
*                   <drvr_req_cnt = Number of bytes to be transferred
*                   <drvr_tran_cnt = $00000000
*                   A Reg = Call Number
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Undefined
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
```

```
* EXIT: via an 'RTS'
*                   <drvr_tran_cnt = Number of bytes transferred
*                   A Reg = Error code
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as entry
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x 0  0       No error occurred
*                           x x 0 0 0 0 x 1  0       Error occurred
*
****************************************************************
status              start
                    using     driver_data
                    longa     on
                    longi     on

*
* Need to verify that the status code specifies a
* legal status request.
*
                    lda       <drvr_stat_code       ; is this a legal status request?
                    cmp       #$0004
                    blt       legal_status          ; yes
                    lda       #drvr_bad_code        ; else return 'BAD CODE' error
                    rts
*
* It's a legal status.  Dispatch to the appropriate status routine.
*
legal_status        anop
                    asl       a
                    tax
                    lda       |status_table,x
                    pha
                    rts       ; dispatch is via an 'RTS'

                    eject
```

```
***************************************************************
*
* The DEVICE STATUS call returns a status list that indicates
* specific status information regarding a character of block
* device and the total number of blocks supported by a block device.
*
* Status List Pointer:        Word                 General status word
*                    Longword  Total number of blocks
*
* Character devices should indicate $00000000 as the block count.
* Status conditions are bit encoded in the status word.
* Encoding of status for a character device is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |_____ OPEN
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |_________ INTERRUPT
*   |   |   |   |   |   |   |   |   |   |   |   |   |___|_____________ RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |_____________________ BUSY
*   |   |   |___|___|___|___|___|___|___|___|_________________________ 0 RESERVED
*   |   |_____________________________________________________________ LINKED DEV
*   |_________________________________________________________________ 0 RESERVED
*
* Encoding of status for a block device is as follows:
*  ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
* | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
* |___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |_____ DISK SW
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |_________ INTERRUPT
*   |   |   |   |   |   |   |   |   |   |   |   |   |   |_____________ WRITE PROT
*   |   |   |   |   |   |   |   |   |   |   |   |   |_________________ 0 RESERVED
*   |   |   |   |   |   |   |   |   |   |   |   |_____________________ ONLINE
*   |   |   |___|___|___|___|___|___|___|___|_________________________ 0 RESERVED
*   |   |_____________________________________________________________ LINKED DEV
*   |_________________________________________________________________ UNVALIDATED
*
* Valid request counts versus returned status list for this status
* call are as follows:
*
* Request Count: $0002         Status List: General status word
* Request Count: $0006         Status List: General status word and block count
*
```

```
*****************************************************************
dev_stat        entry
                longa   on
                longi   on

                lda     #drvr_bad_parm      ; assume invalid request count
                ldx     <drvr_req_cnt+2     ; and validate request count
                bne     bad_dev_stat
                ldx     <drvr_req_cnt
                cpx     #$0002
                blt     bad_dev_stat
                cpx     #$0007
                blt     ok_dev_stat
bad_dev_stat    anop
                sec
                rts
*
* Request count is valid.  Deterime device status and if appropriate, the
* total number of blocks for the device and return them in the device
* status list.    You insert the code required for this operation.
*
ok_dev_stat     anop
                ldx     #dstat_tbl          ; get pointer to device status list
                ldy     #$0000              ; status list pointer
                sep     #$20                ; 8 bit 'm'
                longa   off
copy_dstat      anop
                lda     |0,x                ; copy device status list to slist_ptr
                sta     [<drvr_slist_ptr],y
                inx
                iny
                cpy     <drvr_req_cnt       ; copy = request count size
                bne     copy_dstat
                rep     #$20                ; 16 bit 'm'
                longa   on
*
```

```
* After returning the device status list check for an OFFLINE
* or DISKSW state.  If either of these conditions exist then
* the driver must call SET_DISKSW via the system service call
* table.
*
                  ldy     #dev_char           ; is this a block device?
                  lda     [<drvr_dib_ptr],y
                  and     #$0080
                  beq     not_blk_stat        ; no
                  lda     [<drvr_slist_ptr]
                  eor     #$0010              ; convert online to offline
                  and     #$0011              ; offline or disk switched?
                  beq     not_blk_stat        ; no
                  jsl     set_disksw          ; else call system service
not_blk_stat      anop
                  brl     set_xfer_cnt        ; update xfer count & exit

                  eject
****************************************************************
*
* This call returns a byte count as the first word in the status
* list followed by the data from the control parameter list.
* The request count specifies how much data is to be returned
* from the list.  If the byte count is smaller than the request
* count then only the number of bytes specified by the byte
* count will be returned and the transfer count will indicate
* this.
*
* Status List:    Word    Number of bytes in control list (including byte count).
*                 Data    Data from the control list (device specific).
*
* This call requires a minimum request count of $00000002 and
* a maximum request count of $0000FFFF.
*
****************************************************************
get_ctrl          entry
                  longa   on
                  longi   on

                  lda     #drvr_bad_parm      ; assume invalid request count
                  ldx     <drvr_req_cnt+2 ; and validate request count
                  bne     bad_get_ctrl
                  ldx     <drvr_req_cnt
                  cpx     #$0002
                  bge     ok_get_ctrl
bad_get_ctrl      anop
                  sec
                  rts
*
```

```
* Request count is valid.  Return control list.
*
ok_get_ctrl         anop
                    ldy         #driver_unit          ; internal device #
                    lda         [<drvr_dib_ptr],y
                    asl         a
                    tax
                    lda         |clist_tbl,x          ; get pointer to control list
                    tax
                    lda         |0,x                  ; get length of control list
                    beq         no_clist              ; if list has no content
                    cmp         <drvr_req_cnt         ; is list shorter than request?
                    bge         req_cnt_ok            ; no
                    sta         <drvr_req_cnt         ; else modify request count
req_cnt_ok          anop
                    ldy         #$0000                ; status list index
                    sep         #$20                  ; 8 bit 'm'
                    longa       off
copy_clist          anop
                    lda         |0,x                  ; copy control list to slist_ptr
                    sta         [<drvr_slist_ptr],y
                    inx
                    iny
                    cpy         <drvr_req_cnt         ; copy = request count size
                    bne         copy_clist
                    rep         #$20                  ; 16 bit 'm'
                    longa       on
                    brl         set_xfer_cnt

no_clist            anop
                    sta         [<drvr_slist_ptr]
                    lda         #$0002
                    brl         set_xfer_cnt

                    eject
```

```
*************************************************************
*
* This routine returns the Wait/No Wait mode that the driver
* is currently operating in.  This is returned in a word
* parameter which indicates a request count of $0002.
*
*************************************************************
get_wait            entry
                    longa     on
                    longi     on

                    lda       #drvr_bad_parm        ; assume invalid request count
                    ldx       <drvr_req_cnt+2       ; and validate request count
                    bne       bad_get_wait
                    ldx       <drvr_req_cnt
                    cpx       #$0002
                    beq       ok_get_wait
bad_get_wait        anop
                    sec
                    rts
*
* Request count is valid. Return the wait mode for this device.
*
ok_get_wait         anop
                    ldy       #unit_num
                    lda       [<drvr_dib_ptr],y
                    tax
                    dex
                    lda       |wait_mode_tbl,x
                    sta       [<drvr_slist_ptr]

                    brl       set_xfer_cnt

                    eject
```

```
****************************************************************
*
* This routine returns the format options for the device.
* Consult the driver specification for the format option list.
* This call requires a minimum request count of $00000002.     The
* maximum request count may exceed the size of the format list
* in which case the request count returned will indicate the
* size of the format list.
*
****************************************************************
get_format          entry
                    longa     on
                    longi     on

                    lda       #drvr_bad_parm        ; assume invalid request count
                    ldx       <drvr_req_cnt+2       ; and validate request count
                    bne       bad_get_format
                    ldx       <drvr_req_cnt
                    cpx       #$0002
                    bge       ok_get_format
bad_get_format      anop
                    sec
                    rts
*
* Request count is valid. Return the format options for this device.
*
ok_get_format       anop
                    ldx       #format_tbl           ; get pointer to format option list
                    lda       |0,x                  ; get # entries in option list
                    asl       a                     ; list length = (n * 16) + 8
                    asl       a
                    asl       a
                    asl       a
                    clc
                    adc       #$0008                ; now have option list length
                    cmp       <drvr_req_cnt         ; is request longer then list length?
                    bge       req_count_ok
                    sta       <drvr_req_cnt
req_count_ok        anop
                    ldy       #$0000                ; status list index
                    sep       #$20                  ; 8 bit 'm'
                    longa     off
copy_format         anop
                    lda       |0,x                  ; copy format option list to slist_ptr
                    sta       [<drvr_slist_ptr],y
                    inx
                    iny
                    cpy       <drvr_req_cnt         ; copy = request count size
                    bne       copy_format
                    rep       #$20                  ; 16 bit 'm'
                    longa     on
                    brl       set_xfer_cnt
```

```
***************************************************************
*
* GET_PARTN_MAP:
*
* This routine normally would return the partition map for the
* device.  Since our sample driver does not support partitions,
* the call returns with no error and a transfer count of NIL.
*
***************************************************************
get_partn_map       entry
                    longa     on
                    longi     on

                    lda       #no_error
                    clc
                    rts

                    eject
```

```
****************************************************************
*
* This is a common exit routine for successful status calls.
* The transfer count is set to the same value as the request
* count prior to returning with no error.
*
****************************************************************
set_xfer_cnt        entry
                    lda       <drvr_req_cnt          ; set transfer count
                    sta       <drvr_tran_cnt
                    lda       <drvr_req_cnt+2
                    sta       <drvr_tran_cnt+2
                    lda       #no_error
                    clc
                    rts

                    end

                    eject
****************************************************************
*
* DRIVER CALL:      CONTROL
*
* This routine supports all the standard device control calls.
*
* Control Code:     $0000     Reset Device
*                   $0001     Format Device
*                   $0002     Eject Media
*                   $0003     Set Control Parameters
*                   $0004     Set Wait/No Wait Mode
*                   $0005     Set Format Options
*                   $0006     Assign Partition Owner
*                   $0007     Arm Signal
*                   $0008     Disarm Signal
*
* ENTRY: via a 'JSR'
*                   <drvr_dev_num = Device Number of current device being accessed
*                   <drvr_clist_ptr = Pointer to control list
*                   <drvr_ctrl_code = Control code
*                   <drvr_req_cnt = Number of bytes to be transferred
*                   <drvr_tran_cnt = $00000000
*                   A Reg = Call Number
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Undefined
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
* EXIT: via an 'RTS'
*                   <drvr_tran_cnt = Number of bytes transferred
*                   A Reg = Error code
*                   X Reg = Undefined
```

```
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Same as entry
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x 0  0       No error occurred
*                            x x 0 0 0 0 x 1  0       Error occurred
*
****************************************************************
control              start
                     using     driver_data
                     longa     on
                     longi     on

*
* Need to verify that the control code specifies a
* legal control request.
*
                     lda       <drvr_ctrl_code      ; is this a legal control request?
                     cmp       #$0009
                     blt       legal_control        ; yes
                     lda       #drvr_bad_code       ; else return 'BAD CODE' error
                     rts
*
* It's a legal control.  Dispatch to the appropriate control routine.
*
legal_control        anop
                     asl       a
                     tax
                     lda       |control_table,x
                     pha
                     rts       ; dispatch is via an 'RTS'

                     eject
****************************************************************
*
* This routine will reset the device to its default conditions
* as specified by the default control parameter list.  The
* control list contents will be updated to reflect the parameter
* changes that have taken effect.
*
* CONTROL LIST: None
*
****************************************************************
dev_reset        entry
                 tdc
                 sta       <sup_parm_ptr        ; set pointer to supervisor parameters
                 stz       <sup_parm_ptr+2
                 lda       |sup_num             ; get supervisor driver number
                 ldx       #$0002               ; supervisor specific call
                 jsl       sup_drvr_disp        ; call supervisory driver
                 rts

                 eject
```

```
****************************************************************
*
* This routine will physically format the media.  No additional
* information associated with any particular file system will
* be written to the media. Check task count for disk switch
* prior to execution of read.
*
* CONTROL LIST:   None
*
****************************************************************
format          entry
                tdc
                sta        <sup_parm_ptr          ; set pointer to supervisor parameters
                stz        <sup_parm_ptr+2
                lda        |sup_num               ; get supervisor driver number
                ldx        #$0002                 ; supervisor specific call
                jsl        sup_drvr_disp          ; call supervisory driver
                rts

                eject
****************************************************************
*
* This routine will physically eject media from the device.
* Character devices will not perform any action as a result of
* this call.
*
* CONTROL LIST: None
*
****************************************************************
media_eject     entry
                tdc
                sta        <sup_parm_ptr          ; set pointer to supervisor parameters
                stz        <sup_parm_ptr+2
                lda        |sup_num               ; get supervisor driver number
                ldx        #$0002                 ; supervisor specific call
                jsl        sup_drvr_disp          ; call supervisory driver
                rts

                eject
****************************************************************
*
* This routine will set the configuration parameter list as specified
* by the contents of the configuration list.  Note that the first
* word of the configuration list must have the same value as the
* current configuration parameter list.
*
* CONTROL LIST:     Word      Size of configuration parameter list
*                   Data      Configuration parameter list
*
****************************************************************
set_ctrl        entry
                longa      on
                longi      on
```

```
                lda         #drvr_bad_parm        ; assume invalid request count
                ldx         <drvr_req_cnt+2       ; and validate request count
                bne         bad_set_ctrl
                ldx         <drvr_req_cnt
                cpx         #$0002
                bge         ok_set_ctrl
bad_set_ctrl    anop
                sec
                rts
*
* Request count is valid.  Set configuration list.
*
ok_set_ctrl     anop
                ldy         #driver_unit          ; internal device #
                lda         [<drvr_dib_ptr],y
                asl         a
                tax
                lda         |clist_tbl,x          ; get pointer to configuration list
                tax
                lda         |0,x                  ; are lengths the same?
                cmp         [<drvr_clist_ptr]
                beq         req_cnt_ok            ; yes
                lda         #drvr_bad_parm        ; else return an error
                sec
                rts
req_cnt_ok      anop
                ldy         #$0000                ; status list index
                sep         #$20                  ; 8 bit 'm'
                longa       off
copy_clist      anop                              ; set new configuration list
                lda         [<drvr_clist_ptr],y
                sta         |0,x
                inx
                iny
                tya
                cmp         [<drvr_clist_ptr]
                bne         copy_clist
                rep         #$20                  ; 16 bit 'm'
                longa       on
                tdc
                sta         <sup_parm_ptr         ; set pointer to supervisor parameters
                stz         <sup_parm_ptr+2
                lda         |sup_num              ; get supervisor driver number
                ldx         #$0002                ; supervisor specific call
                jsl         sup_drvr_disp         ; call supervisory driver
                rts
                brl         set_xfer_cnt

                eject
*****************************************************************
*
* This routine will set the WAIT/NO WAIT mode as specified
```

```
* by the contents of the control list.
*
* CONTROL LIST:     Word      Wait / No Wait Mode
*
***************************************************************
set_wait            entry
                    longa     on
                    longi     on

                    lda       #drvr_bad_parm         ; assume invalid request count
                    ldx       <drvr_req_cnt+2        ; and validate request count
                    bne       bad_set_wait
                    ldx       <drvr_req_cnt
                    cpx       #$0002
                    beq       ok_set_wait
bad_set_wait        anop
                    sec
                    rts
*
* Request count is valid. Set the wait mode for this device.
*
ok_set_wait         anop
                    ldy       #driver_unit
                    lda       [<drvr_dib_ptr],y
                    tax
                    lda       [<drvr_slist_ptr]
                    sta       |wait_mode_tbl,x
                    brl       set_xfer_cnt

                    eject
***************************************************************
*
* This routine will set the format option as specified
* by the contents of the control list.
*
* CONTROL LIST:     Word      Format Option Referenc Number
*
***************************************************************
set_format          entry
                    longa     on
                    longi     on

                    lda       #drvr_bad_parm         ; assume invalid request count
                    ldx       <drvr_req_cnt+2        ; and validate request count
                    bne       bad_fmt_opt
                    ldx       <drvr_req_cnt
                    cpx       #$0002
                    beq       ok_set_format
bad_fmt_opt         anop
                    sec
                    rts
*
```

```
* Request count is valid. Set the format option for this device.
*
ok_set_format       anop
                    ldy      #driver_unit
                    lda      [<drvr_dib_ptr],y
                    tax
                    lda      [<drvr_slist_ptr]
                    sta      |format_mode,x
                    brl      set_xfer_cnt

                    eject
*****************************************************************
*
* This routine will set the partition owner as specified
* by the contents of the control list.          Note that this call
* is only supported by partitioned devices such as CD ROM.
* Non partitioned devices should perform no action and return
* with no error.
*
* CONTROL LIST:  Word        String length
*                Name        Name of partition owner
*
*****************************************************************
set_partn        entry
                 tdc
                 sta         <sup_parm_ptr          ; set pointer to supervisor parameters
                 stz         <sup_parm_ptr+2
                 lda         |sup_num               ; get supervisor driver number
                 ldx         #$0002                 ; supervisor specific call
                 jsl         sup_drvr_disp          ; call supervisory driver
                 rts

                 eject
*****************************************************************
*
* This routine is envoked by an application to install a signal
* into the event mechanism.
*
* CONTROL LIST:     Word      Signal Code
*                   Word      Signal Priority
*                   Long      Signal Handler Address
*
*****************************************************************
arm_signal          entry
                    lda       #no_error
                    clc
                    rts
```

```
***************************************************************
*
* This routine is remove a signal from the event mechanism that
* was previously installed with the arm_signal call.
*
* CONTROL LIST: Word  Signal Code
*
***************************************************************
disarm_signal       entry
                    lda       #no_error
                    clc
                    rts

***************************************************************
*
* SET_PARTN_MAP:
*
* This routine normally would set the partition map for the
* device.  Since our sample driver does not support partitions,
* the call returns with no error and a transfer count of NIL.
*
***************************************************************
set_partn_map       entry
                    longa     on
                    longi     on

                    lda       #no_error
                    clc
                    rts

                    end

                    eject
***************************************************************
*
* DRIVER CALL:      FLUSH
*
* This call writes any data in the devices internal buffer to
* the device.  It should be noted that this is a WAIT MODE call
* which is only supported by devices which maintain their own
* internal I/O buffer.  Devices that cannot write in NO WAIT mode
* do not support this call and will return with no error.
*
* ENTRY: via a 'JSR'
*                   <drvr_dev_num = Device Number of current device being accessed
*                   <drvr_tran_cnt = $00000000
*                   A Reg = Call Number
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Undefined
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
```

```
*
* EXIT: via an 'RTS'
*                   <drvr_tran_cnt = Number of bytes transferred
*                   A Reg = Error code
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Same as entry
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x 0  0       No error occurred
*                           x x 0 0 0 0 x 1  0       Error occurred
*
****************************************************************
flush               start
                    using     driver_data
                    longa     on
                    longi     on
*
* Pass on the standard GS/OS call parameters to the supervisory driver.
*
                 tdc
                 sta         <sup_parm_ptr          ; set pointer to supervisor parameters
                 stz         <sup_parm_ptr+2
                 lda         |sup_num               ; get supervisor driver number
                 ldx         #$0002                 ; supervisor specific call
                 jsl         sup_drvr_disp          ; call supervisory driver
                 rts

                 end

                 eject
****************************************************************
*
* DRIVER CALL:      SHUTDOWN
*
* This call prepares the driver for shutdown.  This may include
* closing a character device as well as releasing any and all
* system resources that may have been aquired by either a
* STARTUP or OPEN call.  The driver must return an error if the
* code segment is still in use.  When no error is returned, the
* driver dispatcher will purge the driver's memory segment.
*
* ENTRY: via a 'JSR'
*                   <drvr_dev_num = Device Number of current device being accessed
*                   <drvr_tran_cnt = $00000000
*                   A Reg = Call Number
*                   X Reg = Undefined
*                   Y Reg = Undefined
*                   Dir Reg = GS/OS Direct Page
*                   B Reg = Undefined
*                   P Reg = N V M X D I Z C  E
*                           x x 0 0 0 0 x x  0
*
```

```
* EXIT: via an 'RTS'
*                    <drvr_tran_cnt = Number of bytes transferred
*                    A Reg = Error code
*                    X Reg = Undefined
*                    Y Reg = Undefined
*                    Dir Reg = GS/OS Direct Page
*                    B Reg = Same as entry
*                    P Reg = N V M X D I Z C  E
*                            x x 0 0 0 0 x 0  0      No error occurred
*                            x x 0 0 0 0 x 1  0      Error occurred
*
******************************************************************
shutdn               start
                     using     driver_data
                     longa     on
                     longi     on

                     dec       |startup_count
                     bne       not_last

                     lda       #no_error
                     clc
                     rts

not_last             anop
                     lda       #drvr_busy
                     sec
                     rts

                     end
```

Appendix E GS/OS Error Codes and Constants

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

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

■ **Table E-1** GS/OS errors

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

■ **Table E-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

■ **Table E-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 segments 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.

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.

AppleDisk 3.5 driver: A GS/OS loaded driver that controls Apple 3.5 drives.

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*.

AppleDisk 5.25 driver: A GS/OS loaded driver that controls 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 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*.

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.

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 FST: The part of the GS/OS file system level that makes character devices appear to application programs as if they were sequential files.

character driver: A driver that controls a character device. Also called *character device driver*.

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.

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 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.**

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.

control list: A buffer used in some control subcalls to pass data to devices.

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.

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. Disk drives, printers, mice, 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.**

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.

directory entry: See **file entry.**

directory file: A file that describes and points to other files on disk. Compare **standard file, extended file.**

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.

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.

filename: The string of characters that identifies a particular file within its directory. Compare **pathname.**

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 initalization 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 subcall ArmSignal (or the Arm_Signal). 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.

load file: The output of the linker. Load files contain memory images that the **System Loader** can load into memory, together with relocation dictionaries that the loader uses to relocate references.

loaded drivers: Drivers that are written to work directly with GS/OS, and that are usually loaded in from the system disk at boot time.

long prefix: A GS/OS prefix whose maximum total length is approximately 8,000 characters. Prefix designators 8/ through 31/ refer to long prefixes. Compare **short prefix.**

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. 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 re-execute a program whose user ID has been pulled off the **quit return stack** but which 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 re-execute 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, 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 subcall GetDeviceStatus (or Get_Device_Status) 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 subcall GetFormatOptions (or Get_Format_Options) 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 individal 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. 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 zero, only a single byte is needed to specify a zero-page address.Compare **direct page.**