SCSI Manager 4.3 is an enhanced version of the SCSI Manager that provides new features as well as compatibility with the original version. SCSI Manager 4.3 is contained in the ROM of high-performance computers such as the Macintosh Quadra 840AV and the Power Macintosh 8100/80. Beginning with system software version 7.5, SCSI Manager 4.3 is also available as a system extension that can be installed in any Macintosh computer that uses the NCR 53C96 SCSI controller chip.

In addition to the capabilities of the original SCSI Manager, SCSI Manager 4.3 provides

- support for asynchronous SCSI I/O
- support for optional SCSI features such as disconnect/reconnect
- a hardware-independent programming interface that minimizes the SCSI-specific tasks a device driver must perform

You should read this chapter if you are writing a SCSI device driver or other software for Macintosh computers that use SCSI Manager 4.3. To make best use of this chapter, you should understand the Device Manager and the implementation of device drivers in Macintosh computers. If you are designing a SCSI peripheral device for the Macintosh, you should read *Designing Cards and Drivers for the Macintosh Family,* third edition, and *Guide to the Macintosh Family Hardware*, second edition.

This chapter assumes you are familiar with the following SCSI specifications established by the American National Standards Institute (ANSI):

- X3.131-1986, *Small Computer System Interface*
- X3.131-1994, *Small Computer System Interface–2*
- X3.232 (draft), *SCSI-2 Common Access Method*

If you are writing a device driver for a block-structured storage device such as hard disk, you should also read the chapter "SCSI Manager" in this book for information about the structure of block devices used by the Macintosh Operating System. Because many Macintosh models continue to use the original SCSI Manager, you may want to design your software to operate with both SCSI Manager 4.3 and the original SCSI Manager.

About SCSI Manager 4.3

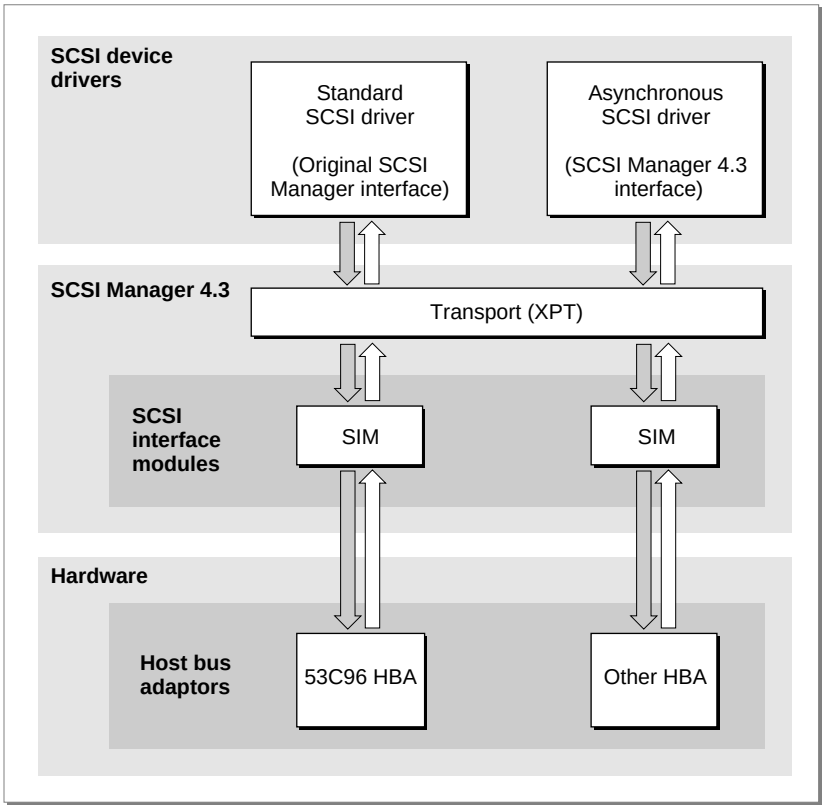
The SCSI Manager 4.3 application program interface (API) is modeled on the Common Access Method (CAM) software interface being developed by ANSI committee X3T9. The SCSI Manager 4.3 interface, however, includes Apple-specific differences required for compatibility with the original SCSI Manager and the Macintosh Operating System.

The CAM specification defines the operation of three functional units—the transport (XPT), the SCSI interface module (SIM), and the host bus adapter (HBA). The XPT is the entry point to SCSI Manager 4.3 and is responsible for passing requests to the appropriate SIM. Each SIM is responsible for managing the HBA for a particular bus.

In addition to the XPT, SCSI Manager 4.3 includes a SIM for managing the NCR 53C96 SCSI controller used in high-performance Macintosh computers. Other SIM

modules and HBA hardware can be added at any time by Apple or third-party developers. For example, a NuBus or PDS expansion card can provide an additional SCSI bus, which device drivers can access through SCSI Manager 4.3 in exactly the same way as the internal bus. Figure 4-1 shows the relationship between device drivers, SCSI Manager 4.3, and the SCSI controller hardware.

Figure 4-1 The SCSI Manager 4.3 architecture



The features and capabilities of SCSI Manager 4.3 include

- ***SCSI-2 compliance.*** All mandatory SCSI-2 messages and protocol actions are supported as defined for an initiator. Optional SCSI-2 hardware features, such as fast and wide transfers, are anticipated by the SCSI Manager 4.3 architecture and supported by the interface.
- ***Concurrent asynchronous I/O.*** SCSI Manager 4.3 handles both synchronous and asynchronous I/O requests. In addition, it allows multiple device drivers to issue multiple requests and attempts to overlap the operations as much as possible.
- ***Hardware-independent programming interface.*** A new hardware-independent interface allows device drivers to work with any SCSI Manager 4.3-compatible host bus adapter (HBA), including those from third-party developers.

- ***Direct memory access (DMA).*** SCSI Manager 4.3 automatically takes advantage of the DMA capabilities available in high-performance Macintosh models. Direct memory access allows the computer to perform other functions while data bytes are transferred to or from the SCSI bus.
- ***Support for multiple buses.*** SCSI Manager 4.3 supports any number of SCSI buses, each with a full complement of devices. For example, on Macintosh computers with dual SCSI buses (such as the Power Macintosh 8100/80), up to 14 SCSI devices can be attached. In addition, developers can design NuBus or PDS expansion cards that offer enhanced SCSI bus capabilities.
- ***Support for multiple logical units on each target.*** SCSI Manager 4.3 allows access to all logical units on a target device. Logical units are treated as separate entities, and I/O requests are queued according to logical unit number (LUN).
- ***Disconnect/reconnect.*** This capability helps maximize SCSI bus utilization by allowing a device to disconnect and release control of the SCSI bus while it processes a command, then reconnect when it is ready to complete the transaction. This allows a device driver to submit requests to multiple targets so that those requests are executed in parallel. For example, the driver for a disk array can issue a request to one disk, which disconnects, then issue another request to a different disk. The two disks can perform their seek operations simultaneously, reducing the effective seek time.
- ***Parity detection.*** SCSI Manager 4.3 detects and handles parity errors in data received from a target. For compatibility reasons, this feature can be disabled on a per-transaction basis. (All Macintosh computers generate parity for write operations, but the original SCSI Manager does not detect parity errors in incoming data.)
- ***Autosense.*** SCSI Manager 4.3 automatically sends a `REQUEST SENSE` command in response to a `CHECK CONDITION` status and retrieves the sense data. This feature can be disabled.
- ***Compatibility.*** SCSI Manager 4.3 supports all original SCSI Manager functions and TIB instructions, except for `scComp` (compare).

Transport

The SCSI Manager 4.3 transport (XPT) provides the software interface to applications and device drivers, and is responsible for

- providing the means to register host bus adapters, their characteristics, and their respective SCSI interface modules
- routing requests to the proper SCSI interface module
- notifying the caller when a request is complete
- providing the high-level facilities for emulating the original SCSI Manager interface. This consists of maintaining a translation table of SCSI ID numbers and their corresponding host bus adapters, and directing original SCSI Manager requests accordingly
- isolating SCSI interface modules from certain operating system requirements, such as those imposed by the Virtual Memory Manager

SCSI Interface Modules

A SCSI interface module (SIM) provides the software interface between the transport (XPT) and a host bus adapter (HBA) in SCSI Manager 4.3. The SIM processes and executes SCSI requests directed to it by the XPT and is responsible for handling all aspects of a SCSI transaction, including

- maintaining the request queue, including freezing and unfreezing for error handling as necessary, and queuing multiple operations for all logical units on all target devices
- managing the selection, disconnection, reconnection, and data pointers of the SCSI protocol
- assigning tags for tag queuing, if supported
- managing the HBA hardware
- identifying abnormal conditions on the SCSI bus and performing error recovery
- providing a time-out mechanism for tracking SCSI command execution
- emulating original SCSI Manager functions, if supported

System Performance

In terms of maximum data transfer (bytes-per-second) over the internal SCSI bus, SCSI Manager 4.3 performs similarly to the original SCSI Manager. This aspect of performance is limited by the capability of the SCSI controller hardware and can be improved by adding a faster HBA.

In terms of overall system performance, the asynchronous capability of SCSI Manager 4.3 can provide significant benefits by allowing application code to regain control of the system while a SCSI transaction is in progress. This concurrency is a key benefit of asynchronous operation. In addition, support for disconnect/reconnect allows applications to initiate multiple I/O requests on multiple targets simultaneously, allowing further increases in throughput.

Multiple bus systems offer the added benefit of concurrency between buses. If DMA is used for both buses, their data transfer periods can be overlapped as well.

Compatibility

All the functions provided by the original SCSI Manager are emulated by the SCSI Manager 4.3 XPT and SIM for the internal SCSI bus. This level of compatibility is optional for third-party SIM/HBA developers. When a SIM registers its HBA with the SCSI Manager 4.3 XPT, the SIM specifies whether or not it is able to emulate the original SCSI Manager functions by setting the `oldCallCapable` field of the SIM initialization record.

When an application or device driver calls the original SCSI Manager function `SCSIGet`, the XPT sets a flag preventing any additional `SCSIGet` function calls but performs no other action. Upon receipt of a `SCSISelect` function call, the XPT issues a `SCSIOldCall` request to the appropriate SIM, which places the request in its queue.

Once the `SCSIOldCall` request begins execution, the SIM emulates subsequent original SCSI Manager function calls passed to it by the XPT. During this emulation, no new requests are processed until the entire transaction is completed and the `SCSIComplete` function returns. Any `SCSIGet` or `SCSISelect` requests received after the start of a `SCSIOldCall` request are rejected and return the `scMgrBusyErr` code.

While the original SCSI Manager emulation is in progress, asynchronous requests made by other applications or device drivers (using SCSI Manager 4.3 functions) are queued but do not execute until the emulation is complete. Requests to other SIMs are not affected and continue to execute normally.

The `SCSIReset` function resets only those buses that are capable of handling original SCSI Manager functions. The `SCSIStat` function returns results as accurate as possible for the SIM/HBA handling the request.

The `scComp` (compare) TIB instruction is not supported by SCSI Manager 4.3 because DMA transfers do not permit this type of compare operation. This should pose few compatibility problems because this instruction is rarely used. You can, of course, write your own code to compare data on a SCSI device with data in memory.

▲ **WARNING**
Applications or device drivers that bypass the SCSI Manager for any part of a transaction are not supported and will interfere with the operation of SCSI Manager 4.3. ▲

Using SCSI Manager 4.3

A fundamental difference between SCSI Manager 4.3 and the original SCSI Manager is that a single function, `SCSIAction`, handles an entire SCSI transaction. You do not need to explicitly arbitrate for the bus, select a device, or send a SCSI command. In most cases, your program does not need to be aware of SCSI bus phases.

The `SCSIAction` function is the entry point for all SCSI Manager 4.3 client functions. These functions provide the services that clients (applications and device drivers) need to communicate with SCSI devices. The only parameter to `SCSIAction` is a pointer to a SCSI Manager parameter block data structure. You use the `scsiFunctionCode` field of the parameter block to specify which function to perform. Most functions use specialized versions of the parameter block to carry the input parameters and return the results. For example, the `SCSIBusInquiry` function requires a SCSI bus inquiry parameter block (`SCSIBusInquiryPB`).

Perhaps the most important `SCSIAction` function is `SCSIExecIO`, which you use to request a SCSI I/O transaction. This function uses the SCSI I/O parameter block (`SCSIExecIOPB`), which specifies the destination of the request (the bus, target, and logical unit), the command descriptor block (CDB), the data buffers that either contain or receive the data, and a variety of other fields and flags required to fulfill the transaction.

You can call the `SCSIExecIO` function either synchronously or asynchronously. If the `scsiCompletion` field of the parameter block contains a pointer to a completion

routine, the SCSI Manager executes the function asynchronously. If you set the `scsiCompletion` field to `nil`, the request is executed synchronously.

Because of interrupt handling considerations, device drivers must issue synchronous `SCSIExecIO` requests as such, rather than issuing them asynchronously and creating a synchronous wait loop inside the device driver. See "Writing a SCSI Device Driver," beginning on page 4-11, for more information about the proper handling of synchronous and asynchronous requests by device drivers. Applications are not subject to the same restrictions as device drivers and may create synchronous wait loops if desired.

Different SIM implementations may require additional fields beyond the standard fields of the SCSI I/O parameter block. Some of these may be input or output fields providing access to special capabilities of a SIM; others may be private fields required during the processing of the request. You can use the `SCSIBusInquiry` function to determine the size of the SCSI I/O parameter block for a particular SIM, as well as the largest parameter block required by any registered SIM.

You can also use the `SCSIBusInquiry` function to get information about various hardware and software characteristics of a SIM and its HBA. You can use this information to form a request that takes advantage of all the capabilities of a SIM.

Parameter blocks are queued separately for each logical unit (LUN) on a target device. When an error occurs during a `SCSIExecIO` request, the SIM freezes the queue for the LUN on which the error occurred, to allow you to perform any necessary error recovery. After correcting the error condition, you must use the `SCSIReleaseQ` function to enable normal handling of I/O requests to that LUN. See "Error Recovery Techniques" on page 4-10 for more information.

Locating SCSI Devices

SCSI Manager 4.3 supports multiple buses, allowing a client to specify a device based on its bus number as well as its target ID and LUN. To emulate original SCSI Manager functions that understand only a target ID, the technique first used in the Macintosh Quadra 900 has been expanded to include not only built-in SCSI buses but any compatible HBA installed in a NuBus or PDS expansion slot.

When multiple buses are registered with the XPT, emulated original SCSI Manager transactions are directed to the first bus that responds to a selection for the requested target ID. The target ID specified in a `SCSISelect` function is called the ***virtual ID*** because it designates a device on the single ***virtual bus*** (which encompasses all original SCSI Manager-compatible buses).

When you make a `SCSISelect` request, the XPT first attempts to select a device on the built-in internal bus. If there is no response on that bus, the XPT tries the built-in external bus (on models that include two SCSI buses), or the first registered add-on bus. Additional buses are searched in the order they were registered.

When the XPT finds a device that responds to the selection, all subsequent `SCSISelect` requests are directed to the bus on which that selection occurred. Until a successful selection occurs on one of the buses, the virtual ID is not assigned to any physical bus.

Once established, the mapping of virtual ID to physical bus is not changed until restart. You can use the `SCSIGetVirtualIDInfo` function to determine which physical bus a device is attached to.

It is possible for devices to be available through the original SCSI Manager interface but not through the SCSI Manager 4.3 interface. For example, a third-party SIM may install its own XPT if SCSI Manager 4.3 is not available. This creates a functional SCSI Manager 4.3 interface that does not include the built-in SCSI bus. Another possibility is the presence of a third-party SCSI adapter that does not comply with SCSI Manager 4.3 but patches the original SCSI Manager interface to create its own virtual bus. To locate all SCSI devices in these environments you must use the SCSI Manager 4.3 functions to scan for devices on all SIMs and then use the original SCSI Manager functions to scan for devices that are not accessible through the SCSI Manager 4.3 interface.

Describing Data Buffers

SCSI Manager 4.3 recognizes three data types for describing the source and destination memory buffers for a SCSI data transfer. The most familiar is a simple buffer, consisting of a single contiguous block of memory. An extension of this is the ***scatter/gather list***, which consists of one or more elements, each of which describes the location and size of one buffer. Scatter/gather lists allow you to group multiple buffers of any size into a single virtual buffer for an I/O transaction.

In addition to these, SCSI Manager 4.3 supports the transfer instruction block (TIB) data type used by the original SCSI Manager interface. This structure is used only for emulating original SCSI Manager functions. During the execution of a `SCSIRead`, `SCSIWrite`, `SCSIRBlind`, or `SCSIWBlind` function, TIB instructions are interpreted by the SCSI Manager to determine the source and destination of the data. See the chapter “SCSI Manager” in this book for more information about TIB instructions.

Handshaking Instructions

In the original SCSI Manager interface, you use TIB instructions to show the SCSI Manager where long delays (greater than 16 microseconds) may occur in a blind transfer. Without these instructions, the SCSI Manager can lose data or crash the system if delays occur at unexpected times in a data transfer.

You use the `scsiHandshake` field of the SCSI I/O parameter block to specify handshaking instructions to SCSI Manager 4.3. This field contains a series of word values, each of which specifies the number of bytes between potential delays in the SCSI data transfer. You terminate the instructions with a value of 0.

For example, a “1, 511” TIB is a common TIB structure used with disk drives that have a 512-byte block size and sometimes experience a delay between the first and second bytes in the block, as well as a delay between the last byte of a block and the first byte of the following block. This TIB structure translates to a `scsiHandshake` field of “1, 511, 0”, which indicates a request to synchronize and transfer 1 byte, synchronize and transfer 511 bytes, synchronize and transfer 1 byte, and so on.

Like the original SCSI Manager, SCSI Manager 4.3 always synchronizes on the first byte of a data phase. In addition, the handshaking cycle is reset whenever a device disconnects. That is, the cycle starts over from the beginning when a device reconnects. The `scsiHandshake` field should also indicate where a device may disconnect.

The handshaking cycle continues across scatter/gather list elements. For example, if the handshake array contains "2048, 0" and the scatter/gather list specifies a transfer of 512 bytes and then 8192 bytes, a handshake synchronization will occur 1536 bytes into the second scatter/gather element.

You should use polled transfers for devices that may experience unpredictable delays during the data phase or can disconnect at unpredictable times.

Error Recovery Techniques

SCSI Manager 4.3 provides a feature called queue freezing that you can use to recover from I/O errors. When a `SCSIExecIO` request returns an error, the SIM freezes the I/O queue for the LUN that caused the error. You can then issue additional requests with the `scsiSIMQHead` flag set so that they will be inserted in front of any requests that were already in the queue. You can use this method to perform retries, block remapping, or other error recovery techniques. After inserting your error handling requests, you call the `SCSIReleaseQ` function to allow the request at the head of the queue to be dispatched. If necessary, multiple requests can be single-stepped by setting the `scsiSIMQFreeze` flag as well as the `scsiSIMQHead` flag on each of the requests and following each with a `SCSIReleaseQ` call.

Note
You can disable queue freezing for a single transaction by setting the `scsiSIMQNoFreeze` flag. ◆

Optional Features

The following optional features may not be supported by all SIMs. You should use the `SCSIBusInquiry` function to determine which features are supported by a particular bus.

- synchronous data transfer
- target command queuing
- HBA engine support
- target mode
- asynchronous event notification

Writing a SCSI Device Driver

This section provides additional information you need to write a device driver that is compatible with both SCSI Manager 4.3 and the original SCSI Manager.

Loading and Initializing a Driver

During system startup of Macintosh models that do not include SCSI Manager 4.3 in ROM, the Start Manager scans the SCSI bus from SCSI ID 6 to SCSI ID 0, looking for devices that have both an `Apple_HFS` and `Apple_Driver` partition. For each device found, the driver is loaded and executed, and installs itself into the unit table. The driver then places an element in the drive queue for any HFS partitions that are on the drive.

When SCSI Manager 4.3 is present in ROM, the Start Manager loads all SCSI Manager 4.3 drivers from all devices on all registered buses. Drivers that support SCSI Manager 4.3 are identified by the string `Apple_Driver43` in the `pmParType` field of the partition map. Traditional (`Apple_Driver`) drivers are then loaded for any devices on the virtual bus that do not contain a SCSI Manager 4.3 driver.

If SCSI Manager 4.3 is not present in ROM, the Start Manager treats SCSI Manager 4.3 drivers exactly like traditional drivers. Because the Start Manager in earlier Macintosh computers checks only the first 12 characters of the `pmParType` field before loading and executing a driver, both SCSI Manager 4.3 drivers and traditional drivers will load on these models. To initialize the driver, the Start Manager jumps to the first byte of the driver's code (using a `JSR` instruction), with register D5 set to the SCSI ID of the device the driver was loaded from.

SCSI Manager 4.3 drivers contain a second entry point at an offset of 8 bytes from the standard entry. Use of this entry point means that SCSI Manager 4.3 is present and that register D5 contains a device identification record. No other registers are used.

There are seven unit table entries (32 through 38) reserved for SCSI drivers controlling devices at SCSI ID 0 through SCSI ID 6 on the virtual SCSI bus. For compatibility with existing SCSI utility software, drivers serving devices on the virtual bus should continue to install themselves in the unit table locations reserved for traditional SCSI drivers. Drivers for devices that are not on the virtual bus should choose a unit number outside the range reserved for traditional SCSI drivers. See the chapter "Device Manager" in this book for information about installing device drivers in the unit table.

To allow clients to determine whether a driver has been loaded for a particular SCSI device, the XPT maintains a driver registration table. This table cross-references device identification records with driver reference numbers. The device identification record is a SCSI Manager 4.3 data structure that specifies a device by its bus, SCSI ID, and logical unit number. The device identification record is defined by the `DeviceIdent` data type, which is described on page 4-19.

A device identification record can have only one driver reference number associated with it, but a single driver reference number may be registered to multiple devices. You can use the `SCSICreateRefNumXref`, `SCSILookupRefNumXref`, and `SCSIRemoveRefNumXref`

functions to access the driver registration table. Drivers loaded through the SCSI Manager 4.3 entry point must use the `SCSICreateRefNumXref` function to register with the XPT. This is done automatically by SCSI Manager 4.3 for traditional drivers.

Selecting a Startup Device

After all device drivers are loaded and initialized, the Start Manager searches for the default startup device in the drive queue. If the device is found, it is mounted and the boot process begins. Macintosh models that do not include SCSI Manager 4.3 in ROM identify the boot drive by a driver reference number stored in PRAM. This works well when drivers retain the same reference number between startups, but SCSI Manager 4.3 drivers allocate unit table entries dynamically if the device they are controlling is not on the virtual bus.

Macintosh models that include SCSI Manager 4.3 in ROM designate the startup device using Slot Manager values in PRAM. Slot number 0 is used for devices on the built-in bus or buses. The `dCtlSlot` and `dCtlSlotId` fields of the driver's device control entry must contain the slot number and sResource ID number, respectively. These are available in the bus inquiry data from the SIM. The `dCtlExtDev` field should contain both the SCSI ID and LUN of the device that the driver is controlling. The high-order 5 bits contain the SCSI ID (up to 31 for a 32-bit wide SCSI bus) and the low-order 3 bits contain the LUN.

Transitions Between SCSI Environments

Because SCSI Manager 4.3 can be installed as a system extension in older Macintosh models, your device driver may be loaded before SCSI Manager 4.3 is active. This can also occur if a NuBus or PDS expansion card loads SCSI Manager 4.3 or an equivalent XPT from the card's ROM. In this case, the expansion card will load a subset of the SCSI Manager 4.3 XPT and a SIM responsible for the card's HBA, but it will not load a SIM for the built-in bus. This creates a situation in which SCSI Manager 4.3 is loaded but some buses may be accessible only through the original interface.

To determine whether to use the SCSI Manager 4.3 interface, your driver should first check for the presence of the `_SCSIAtomic` trap (0xA089). If the trap exists, the driver can pass the SCSI ID of its device to the `SCSIGetVirtualIDInfo` function to get the device identification record of its device. If the `scsiExists` field of the parameter block returns `true`, the device is available through the SCSI Manager 4.3 interface. If the `scsiExists` field returns `false`, the device is on a bus that is not available through SCSI Manager 4.3.

The best time for your driver to perform this check is at the first `accRun` tick, which occurs after all system patches are in place. The Event Manager calls your driver at this time if you set the `dNeedTime` flag in the device control entry. If your driver can access its device through SCSI Manager 4.3, it should allocate and initialize a SCSI I/O parameter block at this time.

Even if your driver is loaded and initialized by a ROM-based SCSI Manager 4.3, you can use the first `accRun` tick to check for new features that may have been installed by a system patch.

Handling Asynchronous Requests

When a client makes a read or write request to a device driver, the Device Manager places the request in the driver's I/O queue. When the driver is ready to accept the request, the Device Manager passes it to the driver's prime routine. The prime routine should fill in a SCSI I/O parameter block with the appropriate values and call the `SCSIExecIO` function. The XPT passes the parameter block to the proper SIM, which then adds the request to its queue and possibly starts processing it before returning back to the driver.

If the `SCSIExecIO` function returns `noErr`, the request was accepted and the contents of the parameter block cannot be reliably viewed by the driver. At this point, virtually nothing can be assumed about the request. It may only have been queued, or it may have proceeded all the way to completion.

IMPORTANT

Once a parameter block is accepted by XPT, do not attempt to examine the parameter block until the completion routine is called. ▲

If `SCSIExecIO` returns an error result, the request was rejected and the completion routine will not be called. This is usually due to an input parameter error.

Completion routines can execute before the XPT returns to your driver. Because the completion routine may initiate a new request to the driver, it is possible that by the time control returns to the calling function, the parameter block is being used for a completely different transaction.

Asynchronous I/O requests from a client to a device driver can occur at interrupt time. Because you cannot allocate memory at interrupt time, you must reserve memory for parameter blocks, scatter/gather lists, and any other structures you need when the driver is initialized. You cannot use the stack for this purpose (as you can for synchronous requests) because parameters on the stack are discarded when the device driver returns from its prime routine.

Asynchronous requests may start at any time and may end at any time. There is no implied ordering of requests with respect to when they were issued. An earlier request may start later, or a later request may complete earlier. However, a series of requests to the same device (bus number, target ID, and LUN) is issued to that device in the order received (unless the `scsiSIMQHead` flag is set in the `scsiFlags` field of the SCSI I/O parameter block, in which case the request is inserted at the head of the queue).

Handling Immediate Requests

If your device driver supports immediate requests, it must be reentrant. The Device Manager neither sets nor checks the `drvrActive` flag in the `dCtlFlags` field of the device control entry before making an immediate request. Asynchronous operation makes it even more likely that an immediate request will happen when your driver is busy because the immediate request may have been made from application time while your driver was asynchronous. When this happens you need to be careful not to reuse parameter blocks or other variables that might be busy.

Virtual Memory Compatibility

Because page faults can occur while interrupts are disabled, SCSI device drivers can receive synchronous I/O requests from the Virtual Memory Manager when the processor interrupt level is not 0. The SCSI Manager handles the resulting SCSI transaction without the benefit of interrupts. This requires that all synchronous wait loops be performed either in the SCSI Manager or in the Device Manager, where code is provided to poll the SCSI interrupt sources.

When your driver receives a synchronous I/O request, it can issue the subsequent SCSI I/O request synchronously as well, or it can issue the SCSI request asynchronously and return to the Device Manager. This second option is generally preferred because it simplifies driver design. The Device Manager waits for the synchronous request to complete, allowing your driver to handle it asynchronously. The driver should jump to `IODone` after it receives the SCSI completion callback. If a single driver request translates to multiple SCSI requests, and your driver handles them asynchronously, the driver should not call `IODone` until after the callbacks for all of the SCSI requests have been received.

IMPORTANT

Because SCSI completion routines must not cause a page fault, all code and data used by SCSI completion routines must be held in real memory. This is automatic for device drivers loaded in the system heap. Applications (or drivers within applications) must use the `HoldMemory` function to ensure their completion routine code and data is held. See the chapter "Virtual Memory Manager" in *Inside Macintosh: Memory* for more information. ▲

Writing a SCSI Interface Module

This section provides additional information that HBA developers need to write a SCSI interface module.

SIM Initialization and Operation

When SCSI Manager 4.3 is present in ROM, the Start Manager loads any SIM drivers it finds in the declaration ROM of all installed expansion cards. A SIM driver may contain the actual SIM, or it may contain code to load the SIM from some other location (such as a device attached to the expansion card). The Start Manager searches for SIM drivers using the Slot Manager `SNextTypeSRsrc` function, and loads all drivers matching the following criteria:

sResource type	Constant	Value
`spCatagory`	`CatIntBus`	12
`spCType`	`TypSIM`	12
`spDrvrSW`	`DrvrSwScsi43`	1

After loading a SIM driver, the Start Manager calls the driver's open routine. If the SIM is contained in the driver, it should register itself with the XPT at this time. If the registration is successful, the open routine should return `noErr`. If the open routine returns an error result, the Start Manager removes the driver from the unit table and releases it from memory. A SIM loader can use this technique to remove itself after loading and registering the actual SIM. Because no other driver entry points are used, you do not need to implement the close, prime, status, or control routines, but they should return appropriate errors.

For Macintosh models that do not include SCSI Manager 4.3 in ROM, your SIM can either provide its own temporary XPT or wait until SCSI Manager 4.3 is installed by the system before registering with the XPT. If you wait for SCSI Manager 4.3 to load, devices on your bus cannot be used as the boot device or as the paging device for virtual memory but can be mounted after SCSI Manager 4.3 is running and your bus is registered.

If your SIM supplies its own XPT, your SIM and XPT must be prepared for the possibility that a system patch will install a new XPT later. To provide a consistent environment for driver clients of your SIM when the XPT is replaced, your XPT must maintain information about any virtual ID numbers it assigns (including a driver registration table) and correctly fill in the XPT fields of the bus inquiry record. When the SCSI Manager4.3 XPT loads, it uses the `SCSIGetVirtualIDInfo`, `SCSILookupRefNumXref`, and `SCSIBusInquiry` functions to query your XPT, then calls the `SetTrapAddress` function to install itself. Next, it uses your XPT to send a `SCSIRegisterWithNewXPT` command to each registered SIM. A SIM must respond by using the `SCSIReregisterBus` function to export its assigned bus number, entry points, and static data storage pointer to the new XPT. Finally, the SCSI Manager 4.3 XPT calls your XPT with a `SCSIKillXPT` command. Your XPT should then release any memory it has allocated and remove or disable any patches it may have installed.

Your XPT must reserve bus number 0 for the built-in SCSI bus. For Macintosh computers with dual SCSI buses, you must reserve bus numbers 0 and 1. If the SCSI Manager 4.3 XPT is installed after your XPT, it will assign these bus numbers to the built-in buses.

After determining the presence of the XPT, a SIM should register itself using the `SCSIRegisterBus` function. The SIM initialization record for this request contains the SIM's function entry points, required static data storage size, and the `oldCallCapable` status of the SIM. The SIM initialization record, defined by the `SIMInitInfo` data type, is shown on page 4-36. The XPT allocates the requested number of bytes for the SIM's static storage, fills in the appropriate fields of the SIM initialization record, and then calls the SIM's `SIMInit` function. If the `SIMInit` function returns `noErr`, the XPT completes the registration process, making the SIM available to the system. If `SIMInit` returns an error, the registration request fails.

Once the registration is complete, the XPT makes calls to the `SIMAction` entry point whenever a `SCSIAction` request is received that is destined for this bus. The XPT passes a pointer to the parameter block and a pointer to the SIM's static storage to the `SIMAction` function. The SIM should parse the parameter block for illegal or unsupported parameters and return an error result if necessary. After queuing the request, the `SIMAction` function should return to the XPT. When the request completes, the SIM calls the XPT's `MakeCallback` function with the appropriate parameter block. The XPT then calls the client's completion routine.

Other types of requests should be implemented to conform to the function descriptions provided in this chapter. Functions or features not implemented by the SIM should return appropriate errors (for example, `scsiFunctionNotAvailable` or `scsiProvideFail`).

The `SIMInteruptPoll` function is called during the Device Manager's synchronous wait loop to give time to the SIM when interrupts are masked. The sole parameter is a pointer to the SIM's static data, which is passed on the stack. Because this call does not imply the presence of an interrupt, the SIM should check for interrupts before proceeding.

The `EnteringSIM` and `ExitingSIM` functions provide compatibility with the Virtual Memory Manager and should be called every time the SIM is entered and exited, respectively. In other words, these two function calls should surround all SIM entry and exit points, including interrupt handlers and callbacks to client code made through the `MakeCallback` function.

Parameter blocks must appear to the client to be queued on a per-LUN basis, because queue freezing and unfreezing are performed one LUN at a time. The actual implementation may vary as long as this appearance is maintained.

Supporting the Original SCSI Manager

If your SIM indicates that it is capable of supporting original SCSI Manager functions, the XPT adds it to the list of buses that are searched when a `SCSISelect` request is received.

The XPT is responsible for converting original SCSI Manager functions into the proper format and submitting them to the SIM. It also receives the results for each of the functions from the SIM and returns them to the client.

When it receives a `SCSIGet` request, the XPT simply notes that the call was made by setting an internal flag, then returns to the caller. In response to a `SCSISelect` request, the XPT generates a `SCSIOldCall` request and submits it to the SIM's `SIMAction` entry point. The `scsiDevice` field of the parameter block contains the bus number of the SIM, the target ID specified in the `SCSISelect` request, and a LUN of 0. This parameter block should be queued like any other.

When your SIM receives a `SCSIOldCall` request, it should attempt to select the device and return a result code to the XPT in the `scsiOldCallResult` field of the parameter block (`scsiRequestComplete` if successful and `scsiSelectTimeout` if not). Intermediate function results are not communicated through the `scsiResult` field because this would be interpreted as completion of the entire transaction rather than only the portion of the transaction resulting from a single original function. As subsequent original function calls are made, the XPT fills in the appropriate fields of the parameter block and calls the SIM's `NewOldCall` entry point. Table 4-1 shows the original function parameters and the fields that are filled in by the XPT.

Table 4-1 Original SCSI Manager parameter conversion

Function	Parameter	Direction	Parameter block field	Notes
`SCSIGet`				XPT handles internally.
`SCSISelect`	`targetID`	→	`scsiDevice`	`bus` set by XPT, `LUN` = 0.
`SCSICmd`	`buffer`	→	`scsiCDB`	Field is a pointer.
	`count`	→	`scsiCDBLength`	
`SCSIRead`, `SCSIWrite`, `SCSIRBlind`, `SCSIWBlind`	`tibPtr`	→	`scsiDataPtr`	Field is a pointer.
`SCSIComplete`	`stat`	←	`scsiSCSIstatus`	Field contains status.
	`message`	←	`scsiSCSImessage`	Field contains message.
	`wait`	→	`scsiTimeout`	Time in Time Manager format.
`SCSIMsgIn`	`message`	←	`scsiSCSImessage`	Field contains message.
`SCSIMsgOut`	`message`	→	`scsiSCSImessage`	Field contains message.
`SCSIReset`				Translated to `SCSIResetBus`.
`SCSIStat`				XPT handles internally.

To provide the highest level of compatibility with the original SCSI Manager, a SIM should be able to perform a SCSI arbitration and select process independently of a SCSI message-out or command phase. A SIM that requires the CDB or message-out bytes in order to perform a select operation will be unable to execute the `SCSISelect` function

properly, and must always return `noErr` to a `SCSISelect` request. This can create a false indication of the presence of a device at a SCSI ID, causing all future `SCSISelect` requests to that SCSI ID to be directed only to that bus. Devices installed on buses that registered after that bus would not be accessible through the original interface.

Handshaking of Blind Transfers

Handshaking instructions are used to prevent bus errors when a target fails to deliver the next byte within the processor bus error timeout period. This timeout is 250 milliseconds for the Macintosh SE and 16 microseconds for all Macintosh models since the Macintosh II.

The SCSI Manager 4.3 SIM requires this handshaking information for blind transfers when DMA is not available. Your SIM does not need to pay attention to the `scsiHandshake` field unless your hardware requires it.

Supporting DMA

DMA typically requires that the data buffer affected by the transfer be locked (so that the physical address does not change) and that it be non-cacheable. SCSI Manager 4.3 provides an improved version of the `LockMemory` function, which you can call at interrupt time as long as the affected pages are already held in real memory. You can also call the `GetPhysical` function at interrupt time, but only on pages that are locked.

Loading Drivers

The Start Manager is normally responsible for loading SCSI drivers. However, if the startup device specified in PRAM is on a third-party HBA and the SIM is a Slot Manager device, the Start Manager will call the boot record of the card's declaration ROM. The boot record code should examine the `dCtlExtDev` field to determine which SCSI device is the startup device and then load a driver from that device (and only that device).

All other drivers are loaded by the Start Manager, but SIMs are given the opportunity to override this if necessary. Before the Start Manager attempts to load a driver from a device, it calls the SIM with a `SCSILoadDriver` request. If the function succeeds, the Start Manager does nothing further with that device. If the function fails (the normal case), the Start Manager reads the partition map on the device and loads a driver from it. If this fails, the Start Manager calls the SIM again with a `SCSILoadDriver` request, this time with the `scsiDiskLoadFailed` parameter set to indicate that no driver was available on the media.

This facility allows a SIM to provide a default driver to be used instead of any driver that may be on the device. For example, if a SIM does support the original SCSI Manager, it can use the second `SCSILoadDriver` request to load a SCSI Manager 4.3-compatible driver if none is present on the device.

SCSI Manager 4.3 Reference

This section describes the data structures, functions, and constants that are specific to SCSI Manager 4.3.

The "Data Structures" section shows the C declarations for the data structures defined by SCSI Manager 4.3.

The "SCSI Manager 4.3 Functions" section describes the functions you use to communicate with SCSI devices, the functions that a SIM uses to communicate with the XPT, and the functions a SIM must include in order to be compatible with SCSI Manager 4.3.

Data Structures

This section describes the parameter blocks you use to communicate with the SCSI Manager and the data structures you use to define values within them.

IMPORTANT
Always set unused or reserved fields to 0 before passing a parameter block to any of the SCSI Manager 4.3 functions. ▲

Simple Data Types

SCSI Manager 4.3 uses these simple data types:

```
typedef char   SInt8;
typedef short  SInt16;
typedef long   SInt32;
typedef unsigned char   UInt8;
typedef unsigned short  UInt16;
typedef unsigned long   UInt32;
```

Device Identification Record

You use the device identification record to specify a target device by its bus, SCSI ID, and logical unit number (LUN). The device identification record is defined by the `DeviceIdent` data type.

```
struct DeviceIdent
{
   UInt8    diReserved;
   UInt8    bus;
   UInt8    targetID;
   UInt8    LUN;
};
typedef struct DeviceIdent DeviceIdent;
```

Field descriptions

`bus`	The bus number of the SIM/HBA for the target device.
`targetID`	The SCSI ID number of the target device.
`LUN`	The target LUN, or 0 if the device does not support logical units.

Command Descriptor Block Record

You use the command descriptor block record to pass SCSI commands to the `SCSIAction` function. The SCSI commands can be stored within this structure, or you can provide a pointer to them. You set the `scsiCDBIsPointer` flag in the SCSI parameter block if this record contains a pointer.

The command descriptor block record is defined by the `CDB` data type.

```
union CDB
{
   UInt8    *cdbPtr;
   UInt8    cdbBytes[maxCDBLength];
};
typedef union CDB CDB, *CDBPtr;
```

Field descriptions

`cdbPtr`	A pointer to a buffer containing a CDB.
`cdbBytes`	A buffer in which you can place a CDB.

Scatter/Gather List Element

You use scatter/gather lists to specify the data buffers to be used for a transfer. A scatter/gather list consists of one or more elements, each of which describes the location and size of one buffer.

The scatter/gather list element is defined by the `SGRecord` data type.

```
struct SGRecord
{
   Ptr      SGAddr;
   SInt32   SGCount;
};
typedef struct SGRecord SGRecord;
```

Field descriptions

`SGAddr`	A pointer to a data buffer.
`SGCount`	The size of the data buffer, in bytes.

SCSI Manager Parameter Block Header

You use the SCSI Manager parameter block to pass information to the `SCSIAction` function. Because many of the functions that you access through `SCSIAction` require additional information, the parameter block consists of a common header (`SCSIPBHdr`) followed by function-specific fields, if any. This section describes the parameter block header common to all `SCSIAction` functions. The function-specific extensions are described in the following sections.

The SCSI Manager parameter block header is defined by the `SCSI_PB` data type.

```
#define SCSIPBHdr              \
   struct   SCSIHdr *qLink;    \
   SInt16   scsiReserved1;     \
   UInt16   scsiPBLength;      \
   UInt8    scsiFunctionCode;  \
   UInt8    scsiReserved2;     \
   OSErr    scsiResult;        \
   DeviceIdent scsiDevice;     \
   CallbackProc scsiCompletion;\
   UInt32   scsiFlags;         \
   UInt8    *scsiDriverStorage;\
   Ptr      scsiXPTprivate;    \
   SInt32   scsiReserved3;

struct SCSI_PB
{
   SCSIPBHdr
};
typedef struct SCSI_PB SCSI_PB;
```

Field descriptions

`qLink` A pointer to the next entry in the request queue. This field is used internally by the SCSI Manager and must be set to 0 when the parameter block is initialized. The SCSI Manager functions always set this field to 0 before returning, so you do not need to set it to 0 again before reusing a parameter block.

`scsiPBLength` The size of the parameter block, in bytes, including the parameter block header.

`scsiFunctionCode` A function selector code that specifies the service being requested. Table 4-2 on page 4-39 lists these codes.

`scsiResult` The result code returned by the XPT or SIM when the function completes. The value `scsiRequestInProgress` indicates that the request is still in progress or queued.

`scsiDevice` A 4-byte value that uniquely identifies the target device for a request. The `DeviceIdent` data type designates the bus number, target SCSI ID, and logical unit number (LUN).

`scsiCompletion` A pointer to a completion routine.

`scsiFlags` Flags indicating the transfer direction and any special handling required for this request.

`scsiDirectionMask`
A bit field that specifies transfer direction, using these constants:

`scsiDirectionIn`	Data in
`scsiDirectionOut`	Data out
`scsiDirectionNone`	No data phase expected

`scsiDisableAutosense`
Disable the automatic `REQUEST SENSE` feature.

`scsiCDBLinked`
The parameter block contains a linked CDB. This option may not be supported by all SIMs.

`scsiQEnable` Enable target queue actions. This option may not be supported by all SIMs.

`scsiCDBIsPointer`
Set if the `scsiCDB` field of a SCSI I/O parameter block contains a pointer. If clear, the `scsiCDB` field contains the actual CDB. In either case, the `scsiCDBLength` field contains the number of bytes in the SCSI command descriptor block.

`scsiInitiateSyncData`
Set if the SIM should attempt to initiate a synchronous data transfer by sending the `SDTR` message. If successful, the device normally remains in the synchronous transfer mode until it is reset or until you specify asynchronous mode by setting the `scsiDisableSyncData` flag. Because `SDTR` negotiation occurs every time this flag is set, you should set it only when negotiation is actually needed.

`scsiDisableSyncData`
Disable synchronous data transfer. The SIM sends an `SDTR` message with a REQ/ACK offset of 0 to indicate asynchronous data transfer mode. You should set this flag only when negotiation is actually needed.

`scsiSIMQHead` Place the parameter block at the head of the SIM queue. This can be used to insert error handling at the head of a frozen queue.

`scsiSIMQFreeze`
Freeze the SIM queue after completing this transaction. See “Error Recovery Techniques” on page 4-10 for information about using this flag.

`scsiSIMQNoFreeze`
Disable SIM queue freezing for this transaction.

`scsiDoDisconnect`
Explicitly allow device to disconnect.

`scsiDontDisconnect`
Explicitly prohibit device disconnection. If this flag and the `scsiDoDisconnect` flag are both 0, the SIM determines whether to allow or prohibit disconnection, based on performance criteria.

`scsiDataReadyForDMA`
Data buffer is locked and non-cacheable.

`scsiDataPhysical`
Data buffer address is physical.

`scsiSensePhysical`
Autosense data pointer is physical.

`scsiDriverStorage`
A pointer to the device driver's private storage. This field is not affected or used by the SCSI Manager.

SCSI I/O Parameter Block

You use the SCSI I/O parameter block to pass information to the `SCSIExecIO` function. The SCSI I/O parameter block is defined by the `SCSIExecIOPB` data type.

```
#define SCSI_IO_Macro \
   SCSIPBHdr                                \
   UInt16         scsiResultFlags;          \
   UInt16         scsiReserved12;           \
   UInt8          *scsiDataPtr;             \
   SInt32         scsiDataLength;           \
   UInt8          *scsiSensePtr;            \
   SInt8          scsiSenseLength;          \
   UInt8          scsiCDBLength;            \
   UInt16         scsiSGListCount;          \
   UInt32         scsiReserved4;            \
   UInt8          scsiSCSIstatus;           \
   SInt8          scsiSenseResidual;        \
   UInt16         scsiReserved5;            \
   SInt32         scsiDataResidual;         \
   CDB            scsiCDB;                  \
   SInt32         scsiTimeout;              \
   UInt8          *scsiReserved13;          \
   UInt16         scsiReserved14;           \
   UInt16         scsiIOFlags;              \
   UInt8          scsiTagAction;            \
```

```
   UInt8          scsiReserved6;            \
   UInt16         scsiReserved7;            \
   UInt16         scsiSelectTimeout;        \
   UInt8          scsiDataType;             \
   UInt8          scsiTransferType;         \
   UInt32         scsiReserved8;            \
   UInt32         scsiReserved9;            \
   UInt16         scsiHandshake[8];         \
   UInt32         scsiReserved10;           \
   UInt32         scsiReserved11;           \
   struct SCSI_IO *scsiCommandLink;         \
   UInt8          scsiSIMpublics[8];        \
   UInt8          scsiAppleReserved6[8];    \
   UInt16         scsiCurrentPhase;         \
   SInt16         scsiSelector;             \
   OSErr          scsiOldCallResult;        \
   UInt8          scsiSCSImessage;          \
   UInt8          XPTprivateFlags;          \
   UInt8          XPTextras[12];

struct SCSI_IO
{
   SCSI_IO_Macro
};
typedef struct SCSI_IO SCSI_IO;
typedef SCSI_IO SCSIExecIOPB;
```

Field descriptions

`SCSIPBHdr` A macro that includes the SCSI Manager parameter block header, described on page 4-21.

`scsiResultFlags`
Output flags that modify the `scsiResult` field.

`scsiSIMQFrozen`
The SIM queue for this LUN is frozen because of an error. You must call the `SCSIReleaseQ` function to release the queue and resume processing requests.

`scsiAutosenseValid`
An automatic `REQUEST SENSE` was performed after this I/O because of a `CHECK CONDITION` status message from the device. The data contained in the `scsiSensePtr` buffer is valid.

`scsiBusNotFree`
The SCSI Manager was unable to clear the bus after an error. You may need to call the `SCSIResetBus` function to restore operation.

`scsiDataPtr` A pointer to a data buffer or scatter/gather list. You specify the data type using the `scsiDataType` field.

`scsiDataLength` The amount of data to be transferred, in bytes.

`scsiSensePtr` A pointer to the autosense data buffer. If autosense is enabled (the `scsiDisableAutosense` flag is not set), the SCSI Manager returns `REQUEST SENSE` information in this buffer.

`scsiSenseLength` The size of the autosense data buffer, in bytes.

`scsiCDBLength` The length of the SCSI command descriptor block, in bytes.

`scsiSGListCount` The number of elements in the scatter/gather list.

`scsiSCSIstatus` The status returned by the SCSI device.

`scsiSenseResidual`
The automatic `REQUEST SENSE` residual length (that is, the number of bytes that were expected but not transferred). This number is negative if extra bytes had to be transferred to force the target off of the bus.

`scsiDataResidual`
The data transfer residual length (that is, the number of bytes that were expected but not transferred). This number is negative if extra bytes had to be transferred to force the target off the bus.

`scsiCDB` This field can contain either the actual CDB or a pointer to the CDB. You set the `scsiCDBIsPointer` flag if this field contains a pointer.

`scsiTimeout` The length of time the SIM should allow before reporting a timeout of the SCSI bus. The time value is represented in Time Manager format (positive values for milliseconds, negative values for microseconds). The timer is started when the I/O request is sent to the target. If the request does not complete within the specified time, the SIM attempts to issue an `ABORT` message, either by reselecting the device or by asserting the attention (/ATN) signal. A value of 0 specifies the default timeout for the SIM. The default timeout for the SCSI Manager 4.3 SIM is infinite (that is, no timeout).

`scsiIOFlags` Additional I/O flags describing the data transfer.

`scsiNoParityCheck`
Disable parity error detection for this transaction.

`scsiDisableSelectWAtn`
Do not send the `IDENTIFY` message for LUN selection. The LUN is still required in the `scsiDevice` field so that the request can be placed in the proper queue. The LUN field in the CDB is untouched. The purpose is to provide compatibility with older devices that do not support this aspect of the SCSI-2 specification.

`scsiSavePtrOnDisconnect`
Perform a `SAVE DATA POINTER` operation automatically in response to a `DISCONNECT` message from the target. The purpose of this flag is to provide compatibility with devices that do not properly implement this aspect of the SCSI-2 specification.

`scsiNoBucketIn`
: Prohibit bit-bucketing during the data-in phase of the transaction. ***Bit-bucketing*** is the practice of throwing away excess data bytes when a target tries to supply more data than the initiator expects. For example, if the CDB requests more data than you specified in the `scsiDataLength` field, the SCSI Manager normally throws away the excess and returns the `scsiDataRunError` result code. If this flag is set, the SCSI Manager refuses any extra data, terminates the I/O request, and leaves the bus in the data-in phase. You must reset the bus to restore operation. This flag is intended only for debugging purposes.

`scsiNoBucketOut`
: Prohibit bit-bucketing during the data-out phase of the transaction. If a target requests more data than you specified in the `scsiDataLength` field, the SCSI Manager normally sends an arbitrary number of meaningless bytes (0xEE) until the target releases the bus. If this flag is set, the SCSI Manager terminates the I/O request when the last byte is sent and leaves the bus in the data-out phase. You must reset the bus to restore operation. This flag is intended only for debugging purposes.

`scsiDisableWide`
: Disable wide data transfer negotiation for this transaction if it had been previously enabled. This option may not be supported by all SIMs.

`scsiInitiateWide`
: Attempt wide data transfer negotiation for this transaction if it is not already enabled. This option may not be supported by all SIMs.

`scsiRenegotiateSense`
: Attempt to renegotiate synchronous or wide transfers before issuing a `REQUEST SENSE`. This is necessary when the error was caused by problems operating in synchronous or wide transfer mode. It is optional because some devices flush sense data after performing negotiation.

`scsiTagAction` Reserved.

`scsiSelectTimeout`
: An optional `SELECT` timeout value, in milliseconds. The default is 250 ms, as specified by SCSI-2. The accuracy of this period is dependent on the HBA. A value of 0 specifies the default timeout. Some SIMs ignore this parameter and always use a value of 250 ms.

`scsiDataType` The data type pointed to by the `scsiDataPtr` field. You specify the type using one of the following constants:

`scsiDataBuffer`
The `scsiDataPtr` field contains a pointer to a contiguous data buffer, and the `scsiDataLength` field contains the length of the buffer, in bytes.

`scsiDataSG` The `scsiDataPtr` field contains a pointer to a scatter/gather list. The `scsiDataLength` field contains the total number of bytes to be transferred, and the `scsiSGListCount` field contains the number of elements in the scatter/gather list.

`scsiDataTIB` The `scsiDataPtr` field contains a pointer to a transfer instruction block. This is used by the XPT during original SCSI Manager emulation, when communicating with a SIM that supports this.

`scsiTransferType`
The type of transfer mode to use during the data phase. You specify the type using one of the following constants:

`scsiTransferBlind`
Use DMA, if available; otherwise, perform a blind transfer using the handshaking information contained in the `scsiHandshake` field.

`scsiTransferPolled`
Use polled transfer mode. The `scsiHandshake` field is not required for this mode.

`scsiHandshake[8]`
Handshaking instructions for blind transfers, consisting of an array of word values, terminated by 0. The SIM polls for data ready after transferring the amount of data specified in each successive `scsiHandshake` entry. When it encounters a 0 value, the SIM starts over at the beginning of the list. Handshaking always starts from the beginning of the list every time a device transitions to data phase. See "Handshaking Instructions," beginning on page 4-9, for more information.

`scsiCommandLink`
A pointer to a linked parameter block. This field provides support for SCSI linked commands. This optional feature ensures that a set of commands sent to a device are executed in sequential order without interference from other applications. You create a list of commands using this pointer to link additional parameter blocks. Each parameter block except the last should have the `scsiCDBLinked` flag set in the `scsiFlags` field. A `CHECK CONDITION` status from the device will abort linked command execution. Linked commands may not be supported by all SIMs.

`scsiSIMpublics[8]`
An additional input field available for use by SIM developers.

`scsiCurrentPhase`
The current SCSI bus phase reported by the SIM after handling an original SCSI Manager function. This field is used only by the XPT and SIM during original SCSI Manager emulation. The phases are defined by the following constant values:

```
enum {
   kDataOutPhase,
   kDataInPhase,
   kCommandPhase,
   kStatusPhase,
   kPhaseIllegal0,
   kPhaseIllegal1,
   kMessageOutPhase,
   kMessageInPhase,
   kBusFreePhase,
   kArbitratePhase,
   kSelectPhase
};
```

`scsiSelector` The function selector code that was passed to the `_SCSIDispatch` trap during original SCSI Manager emulation. The SIM uses this field to determine which original SCSI Manager function to perform.

`scsiOldCallResult`
The result code from an emulated original SCSI Manager function. The SIM returns results to all original SCSI Manager functions in this field, except for the `SCSIComplete` result, which it returns in `scsiResult`.

`scsiSCSIMessage` The message byte returned by an emulated `SCSIComplete` function. This field is only used by the XPT and SIM during original SCSI Manager emulation.

`XPTprivateFlags` Reserved.

`XPTextras[12]` Reserved.

SCSI Bus Inquiry Parameter Block

You use the SCSI bus inquiry parameter block with the `SCSIBusInquiry` function to get information about a bus. The SCSI bus inquiry parameter block is defined by the `SCSIBusInquiryPB` data type.

```
struct SCSIBusInquiryPB
{
   SCSIPBHdr
   UInt16   scsiEngineCount;
   UInt16   scsiMaxTransferType;
   UInt32   scsiDataTypes;
```

```
    UInt16   scsiIOpbSize;
    UInt16   scsiMaxIOpbSize;
    UInt32   scsiFeatureFlags;
    UInt8    scsiVersionNumber;
    UInt8    scsiHBAInquiry;
    UInt8    scsiTargetModeFlags;
    UInt8    scsiScanFlags;
    UInt32   scsiSIMPrivatesPtr;
    UInt32   scsiSIMPrivatesSize;
    UInt32   scsiAsyncFlags;
    UInt8    scsiHiBusID;
    UInt8    scsiInitiatorID;
    UInt16   scsiBIReserved0;
    UInt32   scsiBIReserved1;
    UInt32   scsiFlagsSupported;
    UInt16   scsiIOFlagsSupported;
    UInt16   scsiWeirdStuff;
    UInt16   scsiMaxTarget;
    UInt16   scsiMaxLUN;
    SInt8    scsiSIMVendor[16];
    SInt8    scsiHBAVendor[16];
    SInt8    scsiControllerFamily[16];
    SInt8    scsiControllerType[16];
    SInt8    scsiXPTversion[4];
    SInt8    scsiSIMversion[4];
    SInt8    scsiHBAversion[4];
    UInt8    scsiHBAslotType;
    UInt8    scsiHBAslotNumber;
    UInt16   scsiSIMsRsrcID;
    UInt16   scsiBIReserved3;
    UInt16   scsiAdditionalLength;
};
typedef struct SCSIBusInquiryPB SCSIBusInquiryPB;
```

Field descriptions

`SCSIPBHdr` A macro that includes the SCSI Manager parameter block header, described on page 4-21.

`scsiEngineCount`
The number of engines on the HBA. This value is 0 for a built-in SCSI bus. See the CAM specification for information about HBA engines.

`scsiMaxTransferType`
The number of data transfer types available on the HBA.

`scsiDataTypes` A bit mask describing the data types supported by the SIM/HBA. Bits 3 through 15 and bit 31 are reserved by Apple Computer, Inc. Bits 16 through 30 are available for use by SIM developers. The following bits are currently defined. These types correspond to the `scsiDataType` field of the SCSI I/O parameter block.

```
enum {
   scsiBusDataBuffer    = 0x00000001,
   scsiBusDataTIB       = 0x00000002,
   scsiBusDataSG        = 0x00000004,
   /* bits 3 to 15 are reserved by Apple */
   /* bits 16 to 30 are available for 3rd parties */
   scsiBusDataReserved  = 0x80000000
};
```

`scsiIOpbSize` The minimum size of a SCSI I/O parameter block for this SIM.

`scsiMaxIOpbSize` The minimum size of a SCSI I/O parameter block for all currently registered SIMs. That is, the largest registered `scsiIOpbSize.`

`scsiFeatureFlags`

These flags describe various physical characteristics of the SCSI bus.

`scsiBusInternal`
The bus is at least partly internal to the computer.

`scsiBusExternal`
The bus extends outside of the computer.

`scsiBusInternalExternal`
The bus is both internal and external.

`scsiBusInternalExternalUnknown`
The internal/external state of the bus is unknown.

`scsiBusCacheCoherentDMA`
DMA is cache coherent.

`scsiBusOldCallCapable`
The SIM supports the original SCSI Manager interface.

`scsiBusDifferential`
The bus uses a differential SCSI interface.

`scsiBusFastSCSI`
The bus supports SCSI-2 fast data transfers.

`scsiBusDMAavailable`
DMA is available.

`scsiVersionNumber`

The version number of the SIM/HBA.

`scsiHBAInquiry` Flags describing the capabilities of the bus.

- `scsiBusMDP` Supports the `MODIFY DATA POINTER` message.
- `scsiBusWide32` Supports 32-bit wide transfers.
- `scsiBusWide16` Supports 16-bit wide transfers.
- `scsiBusSDTR` Supports synchronous transfers.
- `scsiBusLinkedCDB` Supports linked commands.
- `scsiBusTagQ` Supports tagged queuing.
- `scsiBusSoftReset` Supports soft reset.

`scsiTargetModeFlags` Reserved.

`scsiScanFlags` Reserved.

`scsiSIMPrivatesPtr` A pointer to the SIM's private storage.

`scsiSIMPrivatesSize` The size of the SIM's private storage, in bytes.

`scsiAsyncFlags` Reserved.

`scsiHiBusID` The highest bus number currently registered with the XPT. If no buses are registered, this field contains 0xFF (the ID of the XPT).

`scsiInitiatorID` The SCSI ID of the HBA. This value is 7 for a built-in SCSI bus.

`scsiFlagsSupported` A bit mask that defines which `scsiFlags` bits are supported.

`scsiIOFlagsSupported` A bit mask that defines which `scsiIOFlags` bits are supported.

`scsiWeirdStuff` Flags that identify unusual aspects of a SIM's operation.

- `scsiOddDisconnectUnsafeRead1` Indicates that a disconnect or other phase change on a odd byte boundary during a read operation will result in inaccurate residual counts or data loss. If your device can disconnect on odd bytes, use polled transfers instead of blind.
- `scsiOddDisconnectUnsafeWrite1` Indicates that a disconnect or other phase change on a odd byte boundary during a write operation will result in inaccurate residual counts or data loss. If your device can disconnect on odd bytes, use polled transfers instead of blind.
- `scsiBusErrorsUnsafe` Indicates that a delay of more than 16 microseconds or a phase change during a blind transfer on a non-handshaked boundary may cause a system crash. If you cannot predict where delays or disconnects will occur, use polled transfers.

scsiRequiresHandshake
: Indicates that a delay of more than 16 microseconds or a phase change during a blind transfer on a non-handshaked boundary may result in inaccurate residual counts or data loss. If you cannot predict where delays or disconnects will occur, use polled transfers.

scsiTargetDrivenSDTRSafe
: Indicates that the SIM supports target-initiated synchronous data transfer negotiation. If your device supports this feature and this bit is not set, you must set the `scsiDisableSelectWAtn` flag in the `scsiIOFlags` field.

`scsiMaxTarget` The highest SCSI ID value supported by the HBA.

`scsiMaxLUN` The highest logical unit number supported by the HBA.

`scsiSIMVendor[16]`
: An ASCII text string that identifies the SIM vendor. This field returns `'Apple Computer'` for a built-in SCSI bus.

`scsiHBAVendor[16]`
: An ASCII text string that identifies the HBA vendor. This field returns `'Apple Computer'` for a built-in SCSI bus.

`scsiControllerFamily[16]`
: An optional ASCII text string that identifies the family of parts to which the SCSI controller chip belongs. This information is provided at the discretion of the HBA vendor.

`scsiControllerType[16]`
: An optional ASCII text string that identifies the specific type of SCSI controller chip. This information is provided at the discretion of the HBA vendor.

`scsiXPTversion[4]`
: An ASCII text string that identifies the version number of the XPT. You should use the other fields of this parameter block to check for specific features, rather than relying on this value.

`scsiSIMversion[4]`
: An ASCII text string that identifies the version number of the SIM. You should use the other fields of this parameter block to check for specific features, rather than relying on this value.

`scsiHBAversion[4]`
: An ASCII text string that identifies the version number of the HBA. You should use the other fields of this parameter block to check for specific features, rather than relying on this value.

`scsiHBAslotType` The slot type, if any, used by this HBA. You specify the type using one of the following constants:

`scsiMotherboardBus`
: A built-in SCSI bus.

`scsiNuBus` A NuBus slot.

`scsiPDSBus` A processor-direct slot.

`scsiHBAslotNumber`
The slot number for the SIM. Device drivers should copy this value into the `dCtlSlot` field of the device control entry. This value is 0 for a built-in SCSI bus.

`scsiSIMsRsrcID` The sResource ID for the SIM. Device drivers should copy this value into the `dCtlSlotID` field of the device control entry. This value is 0 for a built-in SCSI bus.

`scsiAdditionalLength`
The additional size of this parameter block, in bytes. If this structure includes extra fields to return additional information, this field contains the number of additional bytes.

SCSI Abort Command Parameter Block

You use the SCSI abort command parameter block to identify the SCSI I/O parameter block to be canceled by the `SCSIAbortCommand` function. The SCSI abort command parameter block is defined by the `SCSIAbortCommandPB` data type.

```
struct SCSIAbortCommandPB
{
   SCSIPBHdr
   SCSI_IO * scsiIOptr;
};
typedef struct SCSIAbortCommandPB SCSIAbortCommandPB;
```

Field descriptions

`SCSIPBHdr` A macro that includes the SCSI Manager parameter block header, described on page 4-21.

`scsiIOptr` A pointer to the parameter block to be canceled.

SCSI Terminate I/O Parameter Block

You use the SCSI terminate I/O parameter block to identify the SCSI I/O parameter block to be canceled by the `SCSITerminateIO` function. The SCSI terminate I/O parameter block is defined by the `SCSITerminateIOPB` data type.

```
struct SCSITerminateIOPB
{
   SCSIPBHdr
   SCSI_IO * scsiIOptr;
};
typedef struct SCSITerminateIOPB SCSITerminateIOPB;
```

Field descriptions

`SCSIPBHdr` A macro that includes the SCSI Manager parameter block header, described on page 4-21.

`scsiIOptr` A pointer to the parameter block to be canceled.

SCSI Virtual ID Information Parameter Block

You use the SCSI virtual ID information parameter block with the `SCSIGetVirtualIDInfo` function to get the device identification record for a device on the virtual bus. The SCSI virtual ID information parameter block is defined by the `SCSIGetVirtualIDInfoPB` data type.

```
struct SCSIGetVirtualIDInfoPB
{
   SCSIPBHdr
   UInt16   scsiOldCallID;
   Boolean  scsiExists;
};
typedef struct SCSIGetVirtualIDInfoPB SCSIGetVirtualIDInfoPB;
```

Field descriptions

`SCSIPBHdr` A macro that includes the SCSI Manager parameter block header, described on page 4-21. The device information record is returned in the `scsiDevice` field of the parameter block header.

`scsiOldCallID` The virtual SCSI ID of the device you are searching for.

`scsiExists` The XPT returns `true` in this field if the `scsiDevice` field contains a valid device identification record.

SCSI Load Driver Parameter Block

The Start Manager uses this parameter block with the `SCSILoadDriver` function to load a driver for a SCSI device. The SCSI load driver parameter block is defined by the `SCSILoadDriverPB` data type.

```
struct SCSILoadDriverPB
{
   SCSIPBHdr
   SInt16   scsiLoadedRefNum;
   Boolean  scsiDiskLoadFailed;
};
typedef struct SCSILoadDriverPB SCSILoadDriverPB;
```

Field descriptions

`SCSIPBHdr` A macro that includes the SCSI Manager parameter block header, described on page 4-21.

`scsiLoadedRefNum`
If the driver is successfully loaded, this field contains the driver reference number returned by the SIM.

`scsiDiskLoadFailed`
If this field is set to `true`, the SIM should attempt to load its own driver regardless of whether there is one on the device. If this field is set to `false`, the SIM has the option of loading a driver from the device or using one of its own.

SCSI Driver Identification Parameter Block

You use the SCSI driver identification parameter block with the `SCSICreateRefNumXref`, `SCSILookupRefNumXref`, and `SCSIRemoveRefNumXref` functions to exchange device driver registration information. The SCSI driver identification parameter block is defined by the `SCSIDriverPB` data type.

```
struct SCSIDriverPB
{
   SCSIPBHdr
   SInt16      scsiDriver;
   UInt16      scsiDriverFlags;
   DeviceIdent scsiNextDevice;
};
typedef struct SCSIDriverPB SCSIDriverPB;
```

Field descriptions

`SCSIPBHdr` A macro that includes the SCSI Manager parameter block header, described on page 4-21.

`scsiDriver` The driver reference number of the device driver associated with this device identification record.

`scsiDriverFlags`
Driver information flags. These flags are not interpreted by the XPT but can be used to provide information about the driver to other clients. The following flags are defined:

`scsiDeviceSensitive`
Only the device driver should access this device. SCSI utilities and other applications that bypass drivers should check this flag before accessing a device.

`scsiDeviceNoOldCallAccess`
This driver or device does not accept original SCSI Manager requests.

`scsiNextDevice` The device identification record of the next device in the driver registration list.

SIM Initialization Record

You use the SIM initialization record to provide information about your SIM when you register it with the XPT using the `SCSIRegisterBus` function. The SIM initialization record is defined by the `SIMInitInfo` data type.

```
struct SIMInitInfo {
   UInt8             *SIMstaticPtr;
   SInt32            staticSize;
   SIMInitProc       SIMInit;
   SIMActionProc     SIMAction;
   SCSIProc          SIM_ISR;
   InterruptPollProc SIMInterruptPoll;
   SIMActionProc     NewOldCall;
   UInt16            ioPBSize;
   Boolean           oldCallCapable;
   UInt8             simInfoUnused1;
   SInt32            simInternalUse;
   SCSIProc          XPT_ISR;
   SCSIProc          EnteringSIM;
   SCSIProc          ExitingSIM;
   MakeCallbackProc  MakeCallback;
   UInt16            busID;
   UInt16            simInfoUnused3;
   SInt32            simInfoUnused4;
};
typedef struct SIMInitInfo SIMInitInfo;
```

Field descriptions

`SIMstaticPtr` A pointer to the storage allocated by the XPT for the SIM's static variables.

`staticSize` The amount of memory requested by the SIM for storing its static variables.

`SIMInit` A pointer to the SIM's initialization function. See the description of the `SIMInit` function on page 4-60 for more information.

`SIMAction` A pointer to the SIM function that handles `SCSIAction` requests. See the description of the `SIMAction` function on page 4-61 for more information.

`SIM_ISR` Reserved.

`SIMInterruptPoll` A pointer to the SIM's interrupt polling function. The Device Manager periodically calls this routine while waiting for a synchronous request to complete if the processor's interrupt priority level is not 0. This allows the Virtual Memory Manager to initiate SCSI transactions when interrupts are disabled. See the description of the `SIMInterruptPoll` function on page 4-61 for more information.

`NewOldCall`	If the `oldCallCapable` field is set to `true`, this field contains a pointer to the SIM function that handles original SCSI Manager requests. See the description of the `NewOldCall` function beginning on page 4-63 for more information.
`ioPBSize`	The minimum size that a SCSI I/O parameter block must be for use with this SIM.
`oldCallCapable`	A Boolean value that indicates whether the SIM emulates original SCSI Manager functions.
`simInfoUnused1`	Reserved.
`simInternalUse`	A long word available for use by the SIM. This field is not affected or used by the SCSI Manager.
`XPT_ISR`	Reserved.
`EnteringSIM`	A pointer to the XPT `EnteringSIM` function. This function provides support for virtual memory. Your SIM must call this function prior to executing any other SIM code. See the description of the `EnteringSIM` function on page 4-58 for more information.
`ExitingSIM`	A pointer to the XPT `ExitingSIM` function. Your SIM must call this function before passing control to any code that could cause a page fault, including completion routines. See the description of the `ExitingSIM` function on page 4-59 for more information.
`MakeCallback`	A pointer to the XPT `MakeCallback` function. Your SIM must call this function after completing a transaction. The XPT then calls the completion routine specified in the `scsiCompletion` field of the parameter block header. See the description of the `MakeCallback` function on page 4-59 for more information.
`busID`	The bus number assigned by the XPT to this SIM/HBA.

SCSI Manager 4.3 Functions

This section describes the functions you use to communicate with SCSI devices and with the XPT and SIM components of SCSI Manager 4.3.

- "Client Functions" describes the functions that applications and device drivers use to communicate with SCSI devices and the XPT.
- "SIM Support Functions" describes the functions a SIM uses to register its bus and communicate with the XPT.
- "SIM Internal Functions" describes the functions that a SIM must provide in order to be compatible with SCSI Manager 4.3 and the functions that a SIM must include if it supports original SCSI Manager emulation.

Client Functions

This section describes the functions that clients (applications and device drivers) use to communicate with SCSI devices and the XPT.

SCSIAction

You use the `SCSIAction` function to initiate a SCSI transaction or request a service from the XPT or SIM.

```
OSErr SCSIAction(SCSI_PB *scsiPB);
```

`scsiPB` A pointer to a SCSI Manager parameter block.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The function selector code.
←	`scsiResult`	`OSErr`	The returned result code.
→	`scsiDevice`	`DeviceIdent`	A 4-byte value that uniquely identifies the target device.
→	`scsiCompletion`	`CallbackProc`	A pointer to a completion routine. If this field is set to `nil`, the function is executed synchronously.
→	`scsiFlags`	`UInt32`	Flags indicating the transfer direction and any special handling required for the request. See page 4-22 for descriptions of these flags.
→	`scsiDriverStorage`	`UInt8 *`	Optional pointer to the device driver's private storage.

DESCRIPTION

The `SCSIAction` function initiates the request specified by the `scsiFunctionCode` field of the parameter block. Certain types of requests are handled by the XPT, but most are handled by the SIM. Table 4-2 lists the function selector codes. See the following sections for descriptions of the functions you access through `SCSIAction`.

When called asynchronously, `SCSIAction` normally returns the `NoErr` result code, indicating that the request was queued successfully. The result of the SCSI transaction is returned in the `scsiResult` field upon completion. If the `SCSIAction` function returns an error code, the request was not queued and the completion routine will not be called.

When the completion routine is called, it receives the A5 world that existed when the `SCSIAction` request was received. If A5 was invalid when the request was made, it is also invalid in the completion routine.

Your completion routine should use the following function prototype:

```
pascal void (*CallbackProc) (void * scsiPB);
```

There is no implied ordering of asynchronous requests made to different devices. An earlier request may be started later, and a later request may complete earlier. However, a series of requests to the same device is issued to that device in the order received, except when the `scsiSIMQHead` flag is set in the `scsiFlags` field of the parameter block.

When called synchronously, the `SCSIAction` function returns the actual result of the operation. It also places this result in the `scsiResult` field.

Table 4-2 `SCSIAction` function selector codes

Code	Function	Operation
00	`SCSINop`	No operation.
01	`SCSIExecIO`	Execute a SCSI I/O transaction.
02	Reserved	
03	`SCSIBusInquiry`	Bus inquiry.
04	`SCSIReleaseQ`	Release a frozen SIM queue.
05–0F	Reserved	
10	`SCSIAbortCommand`	Abort a SCSI command.
11	`SCSIResetBus`	Reset the SCSI bus.
12	`SCSIResetDevice`	Reset a SCSI device.
13	`SCSITerminateIO`	Terminate I/O transaction.
14–7F	Reserved	
80	`SCSIGetVirtualIDInfo`	Return `DeviceIdent` of a virtual SCSI ID.
81	Reserved	
82	`SCSILoadDriver`	Load a driver from a SCSI device.
83	Reserved	
84	`SCSIOldCall`	SIM support function for original SCSI Manager emulation.
85	`SCSICreateRefNumXref`	Register a device driver.
86	`SCSILookupRefNumXref`	Find a driver reference number.
87	`SCSIRemoveRefNumXref`	Deregister a device driver.
88	`SCSIRegisterWthNewXPT`	XPT was replaced; SIM needs to reregister.
89–BF	Reserved	
C0–FF	Vendor unique	Requests in this range are passed directly to the SIM without evaluation by the XPT.

RESULT CODES

`noErr`	0	Asynchronous request successfully queued, or synchronous request successfully completed

Note
Result codes for specific `SCSIAction` function requests are listed in the following sections. See page 4-90 for a list of all result codes. ◆

SCSINop

The `SCSINop` function does nothing.

```
OSErr SCSIAction(SCSI_PB *scsiPB);
```

`scsiPB` A pointer to a SCSI Manager parameter block.

Parameter block

→	`scsiFunctionCode`	`UInt8`	The `SCSINop` function selector code (0x00).

DESCRIPTION

The `SCSINop` function performs no action, returns no values in the parameter block, and does not call a completion routine. It is provided for compatibility with the CAM specification, and may be useful for debugging.

RESULT CODES

`noErr`	0	No error

SCSIExecIO

You use the `SCSIExecIO` function to perform SCSI I/O operations.

```
OSErr SCSIAction(SCSIExecIOPB *scsiPB);
```

`scsiPB` A pointer to a SCSI I/O parameter block, which is described on page 4-23.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block. This value must be equal to or greater than the `scsiIOpbSize` for the SIM.
→	`scsiFunctionCode`	`UInt8`	The `SCSIExecIO` function selector code (0x01).
←	`scsiResult`	`OSErr`	The returned result code.
→	`scsiDevice`	`DeviceIdent`	The device identification record.
→	`scsiCompletion`	`CallbackProc`	A pointer to a completion routine. If this field is set to `nil`, the function is executed synchronously.
→	`scsiFlags`	`UInt32`	Flags indicating the transfer direction and any special handling required for the request. See page 4-22 for descriptions of these flags.
→	`scsiDriverStorage`	`UInt8 *`	Optional pointer to the device driver's private storage.

←	scsiResultFlags	UInt16	Output flags that modify the scsiResult field. See page 4-24.
→	scsiDataPtr	UInt8 *	A pointer to a data buffer or scatter/gather list.
→	scsiDataLength	UInt32	The amount of data to be transferred.
→	scsiSensePtr	UInt8 *	A pointer to the autosense buffer.
→	scsiSenseLength	UInt8	The size of the autosense buffer.
→	scsiCDBLength	UInt8	The size of the CDB.
→	scsiSGListCount	UInt16	The number of elements in the scatter/gather list.
←	scsiSCSIstatus	UInt8	Status returned by the SCSI device.
←	scsiSenseResidual	SInt8	The autosense residual length.
←	scsiDataResidual	SInt32	The data transfer residual length.
→	scsiCDB	CDB	The CDB, or a pointer to the CDB, depending on the setting of the scsiCDBIsPointer flag.
→	scsiTimeout	SInt32	The SCSI bus timeout period.
→	scsiIOFlags	UInt16	Additional I/O flags. See page 4-25.
→	scsiSelectTimeout	UInt16	Optional SELECT timeout value.
→	scsiDataType	UInt8	The data type pointed to by the scsiDataPtr field. See page 4-27.
→	scsiTransferType	UInt8	The transfer mode (polled or blind). See page 4-27.
→	scsiHandshake[8]	UInt16	Handshaking instructions.
→	scsiCommandLink	SCSI_IO *	Optional pointer to a linked CDB.

DESCRIPTION

The SCSIExecIO function sends a request to a SIM to carry out a SCSI transaction. The SIM performs all the actions necessary to fulfill the request, including arbitrating for the bus, selecting the device, sending the CDB, receiving or sending data, performing disconnect operations, and so on. The parameter block contains all the information required for the SIM to complete the SCSI request, including issuing a REQUEST SENSE command if necessary.

RESULT CODES

noErr	0	No error
scsiRequestInProgress	1	Parameter block request is in progress
scsiCDBLengthInvalid	-7863	The CDB length supplied is not supported by this SIM; typically this means it was too big
scsiTransferTypeInvalid	-7864	The scsiTransferType requested is not supported by this SIM
scsiDataTypeInvalid	-7865	SIM does not support the requested scsiDataType
scsiIDInvalid	-7866	The initiator ID is invalid
scsiLUNInvalid	-7867	The logical unit number is invalid
scsiTIDInvalid	-7868	The target ID is invalid
scsiBusInvalid	-7869	The bus ID is invalid

`scsiRequestInvalid`	-7870	The parameter block request is invalid
`scsiFunctionNotAvailable`	-7871	The requested function is not supported by this SIM
`scsiPBLengthError`	-7872	The parameter block length supplied was too small for this SIM
`scsiQLinkInvalid`	-7881	The `qLink` field was not 0
`scsiNoSuchXref`	-7882	No driver has been cross-referenced with this device
`scsiDeviceConflict`	-7883	Attempt to register more than one driver to a device
`scsiNoHBA`	-7884	No HBA detected
`scsiDeviceNotThere`	-7885	SCSI device not installed or available
`scsiProvideFail`	-7886	Unable to provide the requested service
`scsiBusy`	-7887	SCSI subsystem is busy
`scsiTooManyBuses`	-7888	SIM registration failed because the XPT registry is full
`scsiCDBReceived`	-7910	The SCSI CDB was received
`scsiNoNexus`	-7911	Nexus is not established
`scsiTerminated`	-7912	Parameter block request terminated by the host
`scsiBDRsent`	-7913	A SCSI bus device reset (BDR) message was sent to the target
`scsiWrongDirection`	-7915	Data phase was in an unexpected direction
`scsiSequenceFail`	-7916	Target bus phase sequence failure
`scsiUnexpectedBusFree`	-7917	Unexpected bus free phase
`scsiDataRunError`	-7918	Data overrun/underrun error
`scsiAutosenseFailed`	-7920	Automatic `REQUEST SENSE` command failed
`scsiParityError`	-7921	An uncorrectable parity error occurred
`scsiSCSIBusReset`	-7922	Execution of this parameter block was halted because of a SCSI bus reset
`scsiMessageRejectReceived`	-7923	`REJECT` message received
`scsiIdentifyMessageRejected`	-7924	The target issued a `REJECT` message in response to the `IDENTIFY` message; the LUN probably does not exist
`scsiCommandTimeout`	-7925	The timeout value for this parameter block was exceeded and the parameter block was aborted
`scsiSelectTimeout`	-7926	Target selection timeout
`scsiUnableToTerminate`	-7927	Unable to terminate I/O parameter block request
`scsiNonZeroStatus`	-7932	The target returned non-zero status upon completion of the request
`scsiUnableToAbort`	-7933	Unable to abort parameter block request
`scsiRequestAborted`	-7934	Parameter block request aborted by the host

SCSIBusInquiry

You use the `SCSIBusInquiry` function to get information about a SCSI bus.

```
OSErr SCSIAction(SCSIBusInquiryPB *scsiPB);
```

`scsiPB` A pointer to a SCSI bus inquiry parameter block, which is described on page 4-28.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The `SCSIBusInquiry` function selector code (0x03).
←	`scsiResult`	`OSErr`	The returned result code.
→	`scsiDevice`	`DeviceIdent`	The device identification record. Only the bus number is required.
→	`scsiCompletion`	`CallbackProc`	Unused. Must be `nil`.
←	`scsiEngineCount`	`UInt16`	The number of HBA engines.
←	`scsiMaxTransferType`	`UInt16`	The number of data transfer types available on the HBA.
←	`scsiDataTypes`	`UInt32`	The data types supported.
←	`scsiIOpbSize`	`UInt16`	The minimum parameter block size for this SIM.
←	`scsiMaxIOpbSize`	`UInt16`	The largest parameter block size currently registered.
←	`scsiFeatureFlags`	`UInt32`	Features of the SIM/HBA.
←	`scsiVersionNumber`	`UInt8`	The version of the SIM/HBA
←	`scsiHBAInquiry`	`UInt8`	Features of the SIM/HBA.
←	`scsiSIMPrivatesPtr`	`UInt32`	A pointer to the SIM's storage.
←	`scsiSIMPrivatesSize`	`UInt32`	The size of the SIM's storage.
←	`scsiHiBusID`	`UInt8`	The highest registered bus number.
←	`scsiInitiatorID`	`UInt8`	SCSI ID of the HBA.
←	`scsiFlagsSupported`	`UInt32`	Bit mask of supported `scsiFlags`.
←	`scsiIOFlagsSupported`	`UInt16`	Bit mask of supported `scsiIOFlags`.
←	`scsiWeirdStuff`	`UInt16`	Additional flags.
←	`scsiMaxTarget`	`UInt16`	The highest SCSI ID value supported by the HBA.
←	`scsiMaxLUN`	`UInt16`	The highest logical unit number supported by the HBA.
←	`scsiSIMVendor`	`SInt8[16]`	SIM vendor string.
←	`scsiHBAVendor`	`SInt8[16]`	HBA vendor string.
←	`scsiControllerFamily`	`SInt8[16]`	Controller family string.
←	`scsiControllerType`	`SInt8[16]`	Controller type string.
←	`scsiXPTversion`	`SInt8[4]`	XPT version string.
←	`scsiSIMversion`	`SInt8[4]`	SIM version string.
←	`scsiHBAversion`	`SInt8[4]`	HBA version string.
←	`scsiHBAslotType`	`UInt8`	The slot type of the HBA.

←	scsiHBAslotNumber	UInt8	The slot number of the HBA.
←	scsiSIMsRsrcID	UInt16	The sResource ID of the SIM.
←	scsiAdditionalLength	UInt16	The additional size of this parameter block, if any.

DESCRIPTION

The SCSIBusInquiry function returns information about the SIM and HBA for a bus. This function is typically used to find the minimum size of the SCSI I/O parameter block for a particular SIM. You can also use this function to determine whether a bus supports various optional features such as synchronous or wide transfer modes. Because this function is always executed synchronously, the scsiCompletion field must be set to nil.

To find all buses, first request information about the XPT by setting the bus number in the scsiDevice field to 0xFF, then use the value returned in the scsiHiBusID field to set the limits of the search.

RESULT CODES

noErr	0	No error
scsiBusInvalid	-7869	The bus ID is invalid
scsiRequestInvalid	-7870	The parameter block request is invalid
scsiPBLengthError	-7872	The parameter block is too small for this SIM
scsiQLinkInvalid	-7881	The qLink field was not 0
scsiNoHBA	-7884	No HBA detected
scsiBusy	-7887	SCSI subsystem is busy

SCSIReleaseQ

You use the SCSIReleaseQ function to release a frozen queue for a LUN.

```
OSErr SCSIAction(SCSI_PB *scsiPB);
```

scsiPB A pointer to a SCSI Manager parameter block.

Parameter block

→	scsiPBLength	UInt16	The size of the parameter block.
→	scsiFunctionCode	UInt8	The SCSIReleaseQ function selector code (0x04).
←	scsiResult	OSErr	The returned result code.
→	scsiDevice	DeviceIdent	The device identification record.
→	scsiCompletion	CallbackProc	Unused. Must be set to nil.

DESCRIPTION

The SCSIReleaseQ function releases a frozen I/O queue for the logical unit number specified in the scsiDevice field. If an I/O request returns with the scsiSIMQFrozen flag set in the scsiResultFlags field, you must call this function to restore normal operation.

Queue freezing provides the opportunity to insert error-handling requests at the beginning of the queue using the `scsiSIMQHead` flag. You then release the queue using this function. Subsequent errors will continue to freeze the queue, allowing you to step through the queue one request at a time without aborting any other pending requests.

Because this function is always executed synchronously, the `scsiCompletion` field must be set to `nil`. Unlike other synchronous functions, however, you can call `SCSIReleaseQ` from a completion routine.

RESULT CODES

`noErr`	0	No error
`scsiIDInvalid`	-7866	The initiator ID is invalid
`scsiLUNInvalid`	-7867	The logical unit number is invalid
`scsiTIDInvalid`	-7868	The target ID is invalid
`scsiBusInvalid`	-7869	The bus ID is invalid
`scsiRequestInvalid`	-7870	The parameter block request is invalid
`scsiPBLengthError`	-7872	The parameter block is too small for this SIM
`scsiQLinkInvalid`	-7881	The `qLink` field was not 0

SEE ALSO

See "Error Recovery Techniques" on page 4-10 for more information about queue freezing.

SCSIAbortCommand

You can use the `SCSIAbortCommand` function to cancel an I/O request.

```
OSErr SCSIAction(SCSIAbortCommandPB *scsiPB);
```

`scsiPB` A pointer to a SCSI abort command parameter block, which is described on page 4-33.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The `SCSIAbortCommand` function selector code (0x10).
←	`scsiResult`	`OSErr`	The returned result code.
→	`scsiDevice`	`DeviceIdent`	The device identification record.
→	`scsiCompletion`	`CallbackProc`	A pointer to a completion routine. If this field is set to `nil`, the function is executed synchronously.
→	`scsiDriverStorage`	`UInt8 *`	Optional pointer to the device driver's private storage.
→	`scsiIOptr`	`SCSI_IO *`	A pointer to the SCSI I/O parameter block to be canceled.

DESCRIPTION

The `SCSIAbortCommand` function cancels the `SCSIExecIO` request identified by the `scsiIOptr` field. If the request has not yet been delivered to the device, it is removed from the queue and its completion routine is called with a result code of `scsiRequestAborted`. If the request has already been started, the SIM attempts to send an `ABORT` message to the device, either by asserting the /ATN signal or by reselecting the device. The function returns the `scsiUnableToAbort` result code if the specified request has already been completed.

SPECIAL CONSIDERATIONS

Because the interrupt that calls the completion routine can pre-empt the `SCSIAbortCommand` request, this function can produce unexpected results if the completion routine for the canceled request reuses the parameter block.

RESULT CODES

`noErr`	0	No error
`scsiBusInvalid`	-7869	The bus ID is invalid
`scsiRequestInvalid`	-7870	The parameter block request is invalid
`scsiPBLengthError`	-7872	The parameter block is too small for this SIM
`scsiQLinkInvalid`	-7881	The `qLink` field was not 0
`scsiUnableToAbort`	-7933	Unable to abort parameter block request

SEE ALSO

See the description of the `SCSITerminateIO` function on page 4-48 for information about another method of canceling a request.

SCSIResetBus

You use the `SCSIResetBus` function to reset a SCSI bus.

```
OSErr SCSIAction(SCSI_PB *scsiPB);
```

`scsiPB` A pointer to a SCSI Manager parameter block.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The `SCSIResetBus` function selector code (0x11).
←	`scsiResult`	`OSErr`	The returned result code.
→	`scsiDevice`	`DeviceIdent`	The device identification record. Only the bus number is required.
→	`scsiCompletion`	`CallbackProc`	A pointer to a completion routine. If set to `nil`, the function is executed synchronously.

→	scsiDriverStorage	UInt8 *	Optional pointer to the device driver's private storage.

DESCRIPTION

The SCSIResetBus function directs the HBA to assert the SCSI bus reset signal, causing all devices on the bus to clear pending I/O and forcing the bus into the bus free phase. In addition, the SIM calls the completion routines for all requests that were already delivered to devices. The appropriate LUN queue is frozen for each of the requests that were reset, unless the scsiSIMQNoFreeze flag is set.

SPECIAL CONSIDERATIONS

The SCSIResetBus function interrupts SCSI communications and can cause data loss. You should use this function only to restore operation in the event that a device refuses to release the bus. You can use the SCSIResetDevice function to reset a single device when the SCSI bus is operational and the device is still responding to selection.

RESULT CODES

noErr	0	No error
scsiBusInvalid	-7869	The bus ID is invalid
scsiRequestInvalid	-7870	The parameter block request is invalid
scsiPBLengthError	-7872	The parameter block is too small for this SIM
scsiQLinkInvalid	-7881	The qLink field was not 0

SCSIResetDevice

You use the SCSIResetDevice function to reset a SCSI device.

```
OSErr SCSIAction(SCSI_PB *scsiPB);
```

scsiPB A pointer to a SCSI Manager parameter block.

Parameter block

→	scsiPBLength	UInt16	The size of the parameter block.
→	scsiFunctionCode	UInt8	The SCSIResetDevice function selector code (0x12).
←	scsiResult	OSErr	The returned result code.
→	scsiDevice	DeviceIdent	The device identification record.
→	scsiCompletion	CallbackProc	A pointer to a completion routine. If set to nil, the function is executed synchronously.
→	scsiDriverStorage	UInt8 *	Optional pointer to the device driver's private storage.

DESCRIPTION

The `SCSIResetDevice` function attempts to send a `BUS DEVICE RESET` message to the target. If the device is currently on the bus, the SIM asserts the /ATN signal and sends the message at the next message-out phase. If the target is not on the bus, the SIM selects it and sends an `IDENTIFY` message followed by a `BUS DEVICE RESET` message.

SPECIAL CONSIDERATIONS

The `BUS DEVICE RESET` message clears all I/O transactions for all logical units of the target device. This function may result in data loss and should be used only to restore operation in the event that a device fails to respond to other messages.

RESULT CODES

`noErr`	0	No error
`scsiBusInvalid`	-7869	The bus ID is invalid
`scsiRequestInvalid`	-7870	The parameter block request is invalid
`scsiPBLengthError`	-7872	The parameter block is too small for this SIM
`scsiQLinkInvalid`	-7881	The `qLink` field was not 0
`scsiMessageRejectReceived`	-7923	`REJECT` message received

SCSITerminateIO

You can use the `SCSITerminateIO` function to cancel an I/O request.

```
OSErr SCSIAction(SCSITerminateIOPB *scsiPB);
```

`scsiPB` A pointer to a SCSI terminate I/O parameter block, which is described on page 4-33.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The `SCSITerminateIO` function selector code (0x13).
←	`scsiResult`	`OSErr`	The returned result code.
→	`scsiDevice`	`DeviceIdent`	The device identification record.
→	`scsiCompletion`	`CallbackProc`	A pointer to a completion routine. If this field is set to `nil`, the function is executed synchronously.
→	`scsiDriverStorage`	`UInt8 *`	Optional pointer to the device driver's private storage.
→	`scsiIOptr`	`SCSI_IO *`	A pointer to the SCSI I/O parameter block to be canceled.

DESCRIPTION

The `SCSITerminateIO` function cancels the `SCSIExecIO` request identified by the `scsiIOptr` field. If the request has not yet been delivered to the device, it is removed from the queue and its completion routine is called with a result code of `scsiTerminated`. If the request has already been started, the SIM attempts to send a `TERMINATE IO PROCESS` message to the device, either by asserting the /ATN signal or by reselecting the device. The function returns the `scsiUnableToTerminate` result code if the specified request has already been completed.

The `SCSITerminateIO` function differs from the `SCSIAbortCommand` function (described on page 4-45) only in the message it sends over the SCSI bus. `TERMINATE IO PROCESS` is an optional SCSI-2 message that instructs the device to complete a request normally although prematurely, while attempting to maintain media integrity.

SPECIAL CONSIDERATIONS

Because the interrupt that calls the completion routine can pre-empt the `SCSITerminateIO` request, this function can produce unexpected results if the completion routine for the canceled request reuses the parameter block.

RESULT CODES

`noErr`	0	No error
`scsiBusInvalid`	-7869	The bus ID is invalid
`scsiRequestInvalid`	-7870	The parameter block request is invalid
`scsiPBLengthError`	-7872	The parameter block is too small for this SIM
`scsiQLinkInvalid`	-7881	The `qLink` field was not 0
`scsiUnableToTerminate`	-7927	Unable to terminate I/O parameter block request

SCSIGetVirtualIDInfo

You can use the `SCSIGetVirtualIDInfo` funtion to get the device identification record for a virtual SCSI ID.

```
OSErr SCSIAction(SCSIGetVirtualInfoPB *scsiPB);
```

`scsiPB` A pointer to a SCSI virtual ID information parameter block, which is described on page 4-34.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The `SCSIGetVirtualIDInfo` function selector code (0x80).
←	`scsiResult`	`OSErr`	The returned result code.
←	`scsiDevice`	`DeviceIdent`	The device identification record.
→	`scsiCompletion`	`CallbackProc`	Unused. Must be set to `nil`.
→	`scsiOldCallID`	`UInt16`	The virtual SCSI ID to search for.

←	scsiExists	Boolean	Returns true if the scsiDevice field contains a valid device identification record.

DESCRIPTION

The SCSIGetVirtualIDInfo function returns the device identification record of a device on the virtual bus. This function is typically used by a device driver during the transition from a ROM-based original SCSI Manager to SCSI Manager 4.3. If a device with the specified SCSI ID is not found on the virtual bus, or the device exists but is not accessible through the SCSI Manager 4.3 interface, the scsiExists field returns false and the scsiDevice field should be ignored.

Because this function is always executed synchronously, the scsiCompletion field must be nil.

RESULT CODES

noErr	0	No error
scsiTIDInvalid	-7868	The target ID is invalid
scsiPBLengthError	-7872	The parameter block is too small for this SIM
scsiQLinkInvalid	-7881	The qLink field was not 0

SCSILoadDriver

The Start Manager uses the SCSILoadDriver function to provide an opportunity for a SIM to load a driver other than one found on the media.

```
OSErr SCSIAction(SCSILoadDriverPB *scsiPB);
```

scsiPB A pointer to a SCSI load driver parameter block, which is described on page 4-34.

Parameter block

→	scsiPBLength	UInt16	The size of the parameter block.
→	scsiFunctionCode	UInt8	The SCSILoadDriver function selector code (0x82).
←	scsiResult	OSErr	The returned result code.
→	scsiDevice	DeviceIdent	The device identification record.
→	scsiCompletion	CallbackProc	A pointer to a completion routine. If this field is set to nil, the function is executed synchronously.
→	scsiDriverStorage	UInt8 *	Optional pointer to the device driver's private storage.
←	scsiLoadedRefNum	UInt16	The driver reference number returned by the SIM.
→	scsiDiskLoadFailed	Boolean	Set to true if a driver could not be loaded from the media.

DESCRIPTION

The `SCSILoadDriver` function is called by the Start Manager to load device drivers for SCSI devices. You can use this function to load a driver for a device that was not available at system startup.

The Start Manager can call this function both before and after attempting to load a driver from the media. On the first attempt, the `scsiDiskLoadFailed` field is set to `false`, indicating to the SIM that it can choose to load a driver from the media or install another (typically newer) driver of its own choosing.

If the first attempt to load a driver fails, the Start Manager calls the `SCSILoadDriver` function a second time, with the `scsiDiskLoadFailed` field set to `true` to indicate that a driver could not be loaded from the media. The SIM then loads its own driver, if possible, or returns an error result.

SPECIAL CONSIDERATIONS

The `SCSILoadDriver` function may move memory; you should not call it at interrupt time.

RESULT CODES

`noErr`	0	No error
`scsiFunctionNotAvailable`	-7871	The requested function is not supported by this SIM

SCSICreateRefNumXref

You use the `SCSICreateRefNumXref` function to register a device driver with the XPT.

```
OSErr SCSIAction(SCSIDriverPB *scsiPB);
```

`scsiPB` A pointer to a SCSI driver identification parameter block, which is described on page 4-35.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The `SCSICreateRefNumXref` function selector code (0x85).
←	`scsiResult`	`OSErr`	The returned result code.
→	`scsiDevice`	`DeviceIdent`	The device identification record.
→	`scsiCompletion`	`CallbackProc`	Unused. Must be set to `nil`.
→	`scsiDriver`	`SInt16`	The driver reference number.
→	`scsiDriverFlags`	`UInt16`	Optional driver flags.

DESCRIPTION

The `SCSICreateRefNumXref` function adds an element to the XPT's driver registration table. You specify a device identification record in the `scsiDevice` field and a driver reference number in the `scsiDriver` field. The `scsiDriverFlags` field provides information about the driver that other clients can access using the `SCSILookupRefNumXref` function. The XPT does not interpret these flags.

A device identification record can have only one driver reference number associated with it, but a driver reference number may be registered to multiple devices. This function returns the `scsiDeviceConflict` result code if a driver is already registered to the specified device identification record.

Because this function is always executed synchronously, the `scsiCompletion` field must be set to `nil`.

SPECIAL CONSIDERATIONS

The `SCSICreateRefNumXref` function is executed synchronously and may move memory; you should not call it at interrupt time.

RESULT CODES

`noErr`	0	No error
`scsiQLinkInvalid`	-7881	The `qLink` field was not 0
`scsiDeviceConflict`	-7883	Attempt to register more than one driver to a device

SEE ALSO

See "Loading and Initializing a Driver," beginning on page 4-11, for more information about how device drivers are registered with the XPT.

SCSILookupRefNumXref

You can use the `SCSILookupRefNumXref` function to determine if a driver is installed for a SCSI device.

```
OSErr SCSIAction(SCSIDriverPB *scsiPB);
```

`scsiPB` A pointer to a SCSI driver identification parameter block, which is described on page 4-35.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The `SCSILookupRefNumXref` function selector code (0x86).
←	`scsiResult`	`OSErr`	The returned result code.
→	`scsiDevice`	`DeviceIdent`	The device identification record.
→	`scsiCompletion`	`CallbackProc`	Unused. Must be set to `nil`.

←	scsiDriver	SInt16	The driver reference number.
←	scsiDriverFlags	UInt16	Optional driver flags.
←	scsiNextDevice	DeviceIdent	The device identification record of the next device in the driver registration table.

DESCRIPTION

The SCSILookupRefNumXref function returns the driver reference number for a device. You specify a device identification record in the scsiDevice field, and the function returns the driver reference number in the scsiDriver field. If no driver is registered for the device, the function returns nil in the scsiDriver field.

The scsiDriverFlags field returns the flags that were set when the driver was registered. The scsiNextDevice field returns the device identification record of the next device in the driver registration table. If this is the last device in the table, the function returns 0xFF in the scsiNextDevice.bus field.

To find all registered drivers you should first call this function with a value of 0xFF in the scsiDevice.bus field. The function returns the device identification record of the first device in the list in the scsiNextDevice field. You can then find other drivers by moving the scsiNextDevice value into the scsiDevice field and repeating the operation until the function returns 0xFF in the scsiNextDevice.bus field.

Because this function is always executed synchronously, the scsiCompletion field must be set to nil.

RESULT CODES

noErr	0	No error
scsiQLinkInvalid	-7881	The qLink field was not 0

SCSIRemoveRefNumXref

You use the SCSIRemoveRefNumXref function to deregister a device driver with the XPT.

```
OSErr SCSIAction(SCSIDriverPB *scsiPB);
```

scsiPB — A pointer to a SCSI driver identification parameter block, which is described on page 4-35.

Parameter block

→	scsiPBLength	UInt16	The size of the parameter block.
→	scsiFunctionCode	UInt8	The SCSIRemoveRefNumXref function selector code (0x87).
←	scsiResult	OSErr	The returned result code.
→	scsiDevice	DeviceIdent	The device identification record.
→	scsiCompletion	CallbackProc	Unused. Must be set to nil.

DESCRIPTION

The `SCSIRemoveRefNumXref` function removes a driver entry from the XPT's driver registration table. You specify the device identification record in the `scsiDevice` field.

Because this function is always executed synchronously, the `scsiCompletion` field must be set to `nil`.

SPECIAL CONSIDERATIONS

The `SCSIRemoveRefNumXref` function is executed synchronously, and may move memory; you should not call it at interrupt time.

RESULT CODES

`noErr`	0	No error
`scsiQLinkInvalid`	-7881	The `qLink` field was not 0
`scsiNoSuchXref`	-7882	No driver has been cross-referenced with this device

SEE ALSO

See "Loading and Initializing a Driver," beginning on page 4-11, for more information about how device drivers are registered with the XPT.

SIM Support Functions

This section describes the functions a SIM uses to register its bus and communicate with the XPT. If you are writing a SIM, you use these functions to

- register, deregister, or reregister your SIM with the XPT
- remove the existing XPT if you replace it
- inform the XPT when your code is running
- call a completion routine

SCSIRegisterBus

You use the `SCSIRegisterBus` function to register a SIM and HBA for use with the XPT.

```
OSErr SCSIRegisterBus(SIMInitInfo *SIMinfoPtr);
```

`SIMinfoPtr` A pointer to a SIM initialization record, which is described on page 4-36.

Parameter block

←	`SIMstaticPtr`	`UInt8 *`	A pointer to the allocated static storage.
→	`staticSize`	`SInt32`	The amount of memory requested for static storage.

→	SIMInit	SIMInitProc	A pointer to the SIMInit function.
→	SIMAction	SIMActionProc	A pointer to the SIMAction function.
→	SIMInterruptPoll	InterruptPollProc	A pointer to the SIMInterruptPoll function.
→	NewOldCall	SIMActionProc	A pointer to the NewOldCall function.
→	ioPBSize	UInt16	The SCSI I/O parameter block size for this SIM.
→	oldCallCapable	Boolean	Set to true if the SIM emulates original SCSI Manager functions.
←	EnteringSIM	SCSIProc	A pointer to the EnteringSIM function.
←	ExitingSIM	SCSIProc	A pointer to the ExitingSIM function.
←	MakeCallback	MakeCallbackProc	A pointer to the MakeCallback function.
←	busID	UInt16	The bus number assigned to this SIM/HBA.

DESCRIPTION

You use the SIM initialization record to specify the characteristics of the HBA, the SIM's function entry points, and the number of bytes required for static data storage (global variables). The XPT returns a pointer to the allocated storage and a bus number that identifies the bus in all future transactions. In addition, the XPT returns pointers to the EnteringSIM, ExitingSIM, and MakeCallback functions.

Before assigning a bus number, the XPT calls the SIM's SIMInit function, which instructs the SIM to initialize itself. If the SIMInit function returns noErr, the XPT assigns a bus number and returns from the SCSIRegisterBus function. At this point the SIM is installed and should be ready to accept requests.

SPECIAL CONSIDERATIONS

The SCSIRegisterBus function may move memory; you should not call it at interrupt time.

RESULT CODES

noErr	0	No error
scsiTooManyBuses	-7888	SIM registration failed because the XPT registry is full

SEE ALSO

See "Writing a SCSI Interface Module," beginning on page 4-15, for more information about using this function.

SCSIDeregisterBus

You can use the `SCSIDeregisterBus` function to deregister a bus that is no longer available.

```
OSErr SCSIDeregisterBus(SCSI_PB *scsiPB);
```

`scsiPB` A pointer to a SCSI Manager parameter block.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
←	`scsiResult`	`OSErr`	The returned result code.
→	`scsiDevice`	`DeviceIdent`	The device identification record. Only the bus number is required.
→	`scsiCompletion`	`CallbackProc`	Unused. Must be set to `nil`.

DESCRIPTION

The `SCSIDeregisterBus` function attempts to remove the SIM specified by the `scsiDevice.bus` field of the parameter block. The XPT marks the bus number as invalid and any subsequent requests to it are rejected. This function is not normally used by the Macintosh Operating System and may not be supported in all implementations.

Because this function is always executed synchronously, the `scsiCompletion` field must be set to `nil`.

SPECIAL CONSIDERATIONS

The `SCSIDeregisterBus` function may move memory; you should not call it at interrupt time.

RESULT CODES

`noErr`	0	No error
`scsiBusInvalid`	-7869	The bus ID is invalid
`scsiFunctionNotAvailable`	-7871	The function is not supported by this SIM

SCSIReregisterBus

You can use the `SCSIReregisterBus` function to reregister a bus if its entry points change or if the XPT is replaced.

```
OSErr SCSIReregisterBus(SIMInitInfo *SIMinfoPtr);
```

`SIMinfoPtr` A pointer to a SIM initialization record, which is described on page 4-36.

Parameter block

→	SIMstaticPtr	UInt8 *	A pointer to the SIM's existing static storage.
→	staticSize	SInt32	The size of the SIM's static storage.
→	SIMInit	SIMInitProc	A pointer to the SIMInit function.
→	SIMAction	SIMActionProc	A pointer to the SIMAction function.
→	SIMInterruptPoll	InterruptPollProc	A pointer to the SIMInterruptPoll function.
→	NewOldCall	SIMActionProc	A pointer to the NewOldCall function.
→	ioPBSize	UInt16	The SCSI I/O parameter block size for this SIM.
→	oldCallCapable	Boolean	Set to true if the SIM emulates original SCSI Manager functions.
←	EnteringSIM	SCSIProc	A pointer to the EnteringSIM function.
←	ExitingSIM	SCSIProc	A pointer to the ExitingSIM function.
←	MakeCallback	MakeCallbackProc	A pointer to the MakeCallback function.
→	busID	UInt16	The bus number requested.

DESCRIPTION

You normally call the SCSIReregisterBus function in response to a SCSIRegisterWithNewXPT request. This function is identical to SCSIRegisterBus except that the bus number and static storage pointer are passed *to* the XPT, rather than being returned by it. In addition, the XPT does not call the SIMInit function.

This function allows a SIM to retain its bus number and static storage if the XPT changes. It is also useful if you need to change the SIM's function entry points or other information.

SPECIAL CONSIDERATIONS

The SCSIReregisterBus function may move memory; you should not call it at interrupt time.

RESULT CODES

noErr	0	No error
scsiBusInvalid	-7869	The bus ID is invalid
scsiTooManyBuses	-7888	SIM registration failed because the XPT registry is full

SCSIKillXPT

You use the `SCSIKillXPT` function to remove an XPT that has been replaced.

```
OSErr SCSIKillXPT(void *);
```

DESCRIPTION

The `SCSIKillXPT` function forces the XPT to release any memory it allocated and remove any patches it may have installed. This function is typically called by a new XPT after it has installed itself and reregistered all existing SIMs.

SPECIAL CONSIDERATIONS

The `SCSIKillXPT` function may move memory; you should not call it at interrupt time.

RESULT CODES

`noErr`	0	No error

EnteringSIM

You use the `EnteringSIM` function to inform the XPT that your SIM code is running.

```
void EnteringSIM();
```

DESCRIPTION

The `EnteringSIM` function informs the XPT that subsequent code is not reentrant and instructs the Virtual Memory Manager to defer execution of VBL tasks, Time Manager tasks, completion routines, and any other code that could cause a page fault. A SIM must call this function whenever its code begins executing and call the corresponding `ExitingSIM` function on exit.

SPECIAL CONSIDERATIONS

You get the address of this function from the `EnteringSIM` field of the SIM initialization record.

SEE ALSO

See "Writing a SCSI Interface Module," beginning on page 4-15, for more information about using this function.

ExitingSIM

The `ExitingSIM` function is the counterpart to the `EnteringSIM` function.

```
void ExitingSIM();
```

DESCRIPTION

The `ExitingSIM` function informs the XPT that the SIM is about to pass control to an external routine that might cause a page fault. A SIM must call this function before returning to the XPT or calling a completion routine.

SPECIAL CONSIDERATIONS

You get the address of this function from the `ExitingSIM` field of the SIM initialization record.

SEE ALSO

See "Writing a SCSI Interface Module," beginning on page 4-15, for more information about using this function.

MakeCallback

You use the `MakeCallback` function to signal the XPT to call a completion routine.

```
void MakeCallback(SCSI_IO *scsiPB);
```

`scsiPB` A pointer to a SCSI I/O parameter block, which is described on page 4-23.

Parameter block

→ `scsiCompletion CallbackProc` A pointer to a completion routine.

DESCRIPTION

The `MakeCallback` function instructs the XPT to execute the completion routine in the SCSI I/O parameter block. The XPT restores the client's A5 world and then calls the completion routine. A SIM should always use this function rather than calling completion routines directly because the XPT may chose to defer the actual execution of the routine until page faults are safe.

You should surround a call to `MakeCallback` with calls to `ExitingSIM` and `EnteringSIM` so that the Virtual Memory Manager can properly handle any page faults caused by the completion routine.

SPECIAL CONSIDERATIONS

You get the address of this function from the `MakeCallback` field of the SIM initialization record.

SEE ALSO

See "Writing a SCSI Interface Module," beginning on page 4-15, for more information about using this function.

SIM Internal Functions

This section describes the functions that a SIM must provide to be compatible with SCSI Manager 4.3 and the functions that a SIM must include if it supports original SCSI Manager emulation. These functions are called by the XPT to control or provide information to the SIM.

See "Writing a SCSI Interface Module," beginning on page 4-15, for more information about using these functions.

SIMInit

The XPT calls this function to initialize a SIM. The `SIMInit` function must conform to the following type definition:

```
typedef OSErr (*SIMInitProc) (Ptr SIMinfoPtr);
```

`SIMinfoPtr` A pointer to a SIM initialization record, which is described on page 4-36.

DESCRIPTION

The XPT calls this function after a SIM has called `SCSIRegisterBus`. Before returning from the `SCSIRegisterBus` function, the XPT calls this function to initialize the SIM. The SIM is responsible for initializing the HBA.

The XPT passes a pointer to the SIM initialization record, which contains pointers to the SIM's static data storage and the required XPT entry points (`EnteringSIM`, `ExitingSIM`, and `MakeCallback`).

RESULT CODES

`noErr`	0	No error
`scsiNoHBA`	-7884	No HBA detected

SIMAction

The XPT calls this function when a `SCSIAction` request is received that needs to be serviced by the SIM. The `SIMAction` function must conform to the following type definition:

```
typedef void (*SIMActionProc) (void * scsiPB, Ptr SIMGlobals);
```

`scsiPB` A pointer to a SCSI Manager parameter block.

`SIMGlobals` A pointer to the SIM's static data storage.

DESCRIPTION

The `SIMAction` function is responsible for handling `SCSIAction` requests directed to the SIM's bus. The XPT passes the client's parameter block to the SIM, which should then queue the request, execute it, and call the completion routine. The SIM must conform to the behavior defined for the `SCSIAction` function.

In addition to supporting all client functions, the `SIMAction` function may optionally support two requests made by the XPT, `SCSIOldCall` and `SCSIRegisterWithNewXPT`.

RESULT CODES

The `SIMAction` function returns a result code in the `scsiResult` field of the parameter block. The code should be appropriate for the `SCSIAction` request being processed.

SIMInterruptPoll

The XPT calls this function when interrupts are disabled during a synchronous wait loop, to give the SIM an opportunity to handle interrupts from the HBA. The `SIMAction` function must conform to the following type definition:

```
typedef void (*InterruptPollProc) (Ptr SIMGlobals);
```

`SIMGlobals` A pointer to the SIM's static data storage.

DESCRIPTION

If the Device Manager is waiting for a synchronous request to complete, and processor interrupts are masked at level 2 (the level of NuBus interrupts) or higher, the XPT periodically calls the `SIMInterruptPoll` function for each SIM. The SIM can then check whether an interrupt is pending from the HBA, and execute its interrupt service routine if necessary.

SCSIOldCall

The XPT calls this function when a client calls the original SCSI Manager function `SCSISelect`.

```
typedef void (*SIMActionProc) (void * scsiPB, Ptr SIMGlobals);
```

`scsiPB` A pointer to a SCSI I/O parameter block, which is described on page 4-23.

`SIMGlobals` A pointer to the SIM's static data storage.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The `SCSIOldCall` function selector code (0x84).
→	`scsiDevice`	`DeviceIdent`	The device identification record.
→	`scsiCompletion`	`CallbackProc`	A pointer to a completion routine. If this field is set to `nil`, the function is executed synchronously.
→	`scsiDriverStorage`	`UInt8 *`	Optional pointer to the device driver's private storage.
←	`scsiCurrentPhase`	`UInt16`	The current SCSI bus phase.
→	`scsiSelector`	`SInt16`	The `SCSISelect` trap selector (0x02).
←	`scsiOldCallResult`	`OSErr`	The result code from `SCSISelect`.

DESCRIPTION

This function indicates the beginning of an original SCSI Manager transaction. A SIM that supports original SCSI Manager emulation should attempt to select the device described in the `scsiDevice` field. Because the entire SCSI transaction is not completed by a call to `SCSIOldCall`, the result code for this function is returned in the `scsiOldCallResult` field rather than the `scsiResult` field, as with other functions. Subsequent original SCSI Manager function calls for this transaction are made through the `NewOldCall` function.

If the SIM successfully selects the device, it should queue the parameter block like any other SCSI I/O parameter block. The parameter block should not be removed until the `NewOldCall` function completes a `SCSIComplete` command.

To provide full compatibility with the original SCSI Manager, a SIM must be able to perform a SCSI arbitration and select process independent of a SCSI message-out or command phase. If the SIM requires the CDB or message-out bytes it will not be able to perform the select operation at the time of the `SCSIOldCall` request. The SIM should return `noErr` in the `scsiOldCallResult` field and wait for a subsequent I/O request before actually selecting the device.

RESULT CODES

The `SCSIOldCall` function returns an appropriate `SCSISelect` result code in the `scsiOldCallResult` field of the parameter block.

NewOldCall

The XPT calls this function when a client calls any of the original SCSI Manager functions other than `SCSISelect` (which is handled by `SCSIOldCall`). The `NewOldCall` function must conform to the following type definition:

```
typedef void (*SIMActionProc) (void * scsiPB, Ptr SIMGlobals);
```

`scsiPB` A pointer to a SCSI I/O parameter block, which is described on page 4-23.

`SIMGlobals` A pointer to the SIM's static data storage.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
←	`scsiResult`	`OSErr`	The `SCSIComplete` result code.
→	`scsiDevice`	`DeviceIdent`	The device identification record.
→	`scsiCompletion`	`CallbackProc`	A pointer to a completion routine. If this field is set to `nil`, the function is executed synchronously.
→	`scsiDriverStorage`	`UInt8 *`	Optional pointer to the device driver's private storage.
←	`scsiCurrentPhase`	`UInt16`	The current SCSI bus phase.
→	`scsiSelector`	`SInt16`	The `_SCSIDispatch` trap selector.
←	`scsiOldCallResult`	`OSErr`	The function result code.
←	`scsiSCSImessage`	`UInt8`	The `SCSIComplete` message byte.

DESCRIPTION

After an original SCSI Manager transaction begins, the `NewOldCall` function receives all subsequent original SCSI Manager function requests until the transaction is completed. The XPT converts all original SCSI Manager function requests (except `SCSIGet` and `SCSIStat`) into SCSI Manager 4.3 parameter block requests and sends them to the appropriate SIM.

A SIM uses the `scsiSelector` field of the parameter block to determine which function to perform and should return the current bus phase and message byte in the appropriate fields after each request.

The XPT converts a `SCSIReset` request into a `SCSIResetBus` request and sends it to all SIMs that support original SCSI Manager emulation. The XPT handles `SCSIStat` requests itself, using the information returned in the `scsiCurrentPhase` field.

RESULT CODES

Result codes from all emulated functions except `SCSIComplete` are returned in the `scsiOldCallResult` field. The `SCSIComplete` result is returned in `scsiResult`. This indicates to the XPT that the transaction is complete and that the SIM is ready to start a new original SCSI Manager transaction. See the chapter "SCSI Manager" in this book for a list of original SCSI Manager result codes.

SCSIRegisterWithNewXPT

This function informs a SIM that a new XPT layer has been installed. The SIM should call the `SCSIReregisterBus` function to register itself with the new XPT.

```
typedef void (*SIMActionProc) (void * scsiPB, Ptr SIMGlobals);
```

`scsiPB` A pointer to a SCSI Manager parameter block.

`SIMGlobals` A pointer to the SIM's static data storage.

Parameter block

→	`scsiPBLength`	`UInt16`	The size of the parameter block.
→	`scsiFunctionCode`	`UInt8`	The `SCSIRegisterWithNewXPT` function selector code (0x88).

DESCRIPTION

After a new XPT installs itself, and before it removes the old XPT, it sends the `SCSIRegisterWithNewXPT` request to all SIMs registered with the old XPT. Each SIM should then call the `SCSIReregisterBus` function to register with the new XPT. This allows SIMs to keep their existing bus number and static data storage when installing themselves in a new XPT.

RESULT CODES

`noErr`	0	No error
`scsiQLinkInvalid`	-7881	The `qLink` field was not 0

Summary of SCSI Manager 4.3

C Summary

Constants

```
enum {
   scsiVERSION = 43
};

/* SCSI Manager function codes */
enum {
   SCSINop                 = 0x00,  /* no operation */
   SCSIExecIO              = 0x01,  /* execute a SCSI IO transaction */
   SCSIBusInquiry          = 0x03,  /* bus inquiry */
   SCSIReleaseQ            = 0x04,  /* release a frozen SIM queue */
   SCSIAbortCommand        = 0x10,  /* abort a SCSI command */
   SCSIResetBus            = 0x11,  /* reset the SCSI bus */
   SCSIResetDevice         = 0x12,  /* reset a SCSI device */
   SCSITerminateIO         = 0x13,  /* terminate I/O transaction */
   SCSIGetVirtualIDInfo    = 0x80,  /* return DeviceIdent of virtual ID */
   SCSILoadDriver          = 0x82,  /* load a driver from a SCSI device */
   SCSIOldCall             = 0x84,  /* begin old-API emulation */
   SCSICreateRefNumXref    = 0x85,  /* register a device driver */
   SCSILookupRefNumXref    = 0x86,  /* find a driver reference number */
   SCSIRemoveRefNumXref    = 0x87,  /* deregister a device driver */
   SCSIRegisterWithNewXPT  = 0x88,  /* XPT replaced; SIM must reregister */
   vendorUnique            = 0xC0   /* 0xC0 through 0xFF */
};

/* allocation lengths for parameter block fields */
enum {
   handshakeDataLength     = 8,     /* handshake data length */
   maxCDBLength            = 16,    /* space for the CDB bytes/pointer */
   vendorIDLength          = 16     /* ASCII string length for vendor ID  */
};
```

```
/* types for the scsiTransferType field */
enum {
   scsiTransferBlind    = 0,        /* DMA if available, otherwise blind */
   scsiTransferPolled               /* polled */
};

/* types for the scsiDataType field */
enum {
   scsiDataBuffer       = 0,        /* single contiguous buffer supplied */
   scsiDataTIB          = 1,        /* TIB supplied (ptr in scsiDataPtr) */
   scsiDataSG           = 2         /* scatter/gather list supplied */
};

/* flags for the scsiResultFlags field */
enum {
   scsiSIMQFrozen       = 0x0001,   /* the SIM queue is frozen */
   scsiAutosenseValid   = 0x0002,   /* autosense data valid for target */
   scsiBusNotFree       = 0x0004    /* SCSI bus is not free */
};

/* bit numbers of the scsiFlags field */
enum {
   kbSCSIDisableAutosense  = 29,    /* disable autosense feature */
   kbSCSIFlagReservedA     = 28,
   kbSCSIFlagReserved0     = 27,
   kbSCSICDBLinked         = 26,    /* the PB contains a linked CDB */
   kbSCSIQEnable           = 25,    /* target queue actions are enabled */
   kbSCSICDBIsPointer      = 24,    /* the CDB field contains a pointer */
   kbSCSIFlagReserved1     = 23,
   kbSCSIInitiateSyncData  = 22,    /* attempt sync data transfer and SDTR */
   kbSCSIDisableSyncData   = 21,    /* disable sync, go to async */
   kbSCSISIMQHead          = 20,    /* place PB at the head of SIM queue */
   kbSCSISIMQFreeze        = 19,    /* freeze the SIM queue */
   kbSCSISIMQNoFreeze      = 18,    /* disable SIM queue freezing */
   kbSCSIDoDisconnect      = 17,    /* definitely do disconnect */
   kbSCSIDontDisconnect    = 16,    /* definitely don't disconnect */
   kbSCSIDataReadyForDMA   = 15,    /* data buffer(s) are ready for DMA */
   kbSCSIFlagReserved3     = 14,
   kbSCSIDataPhysical      = 13,    /* S/G buffer data ptrs are physical */
   kbSCSISensePhysical     = 12,    /* autosense buffer ptr is physical */
   kbSCSIFlagReserved5     = 11,
   kbSCSIFlagReserved6     = 10,
   kbSCSIFlagReserved7     = 9,
   kbSCSIFlagReserved8     = 8,
```

```
   kbSCSIDataBufferValid   = 7,     /* data buffer valid */
   kbSCSIStatusBufferValid = 6,     /* status buffer valid */
   kbSCSIMessageBufferValid= 5,     /* message buffer valid */
   kbSCSIFlagReserved9     = 4
};

/* bit masks for the scsiFlags field */
enum {
   scsiDirectionMask       = 0xC0000000, /* data direction mask */
   scsiDirectionNone       = 0xC0000000, /* data direction (11: no data) */
   scsiDirectionReserved   = 0x00000000, /* data direction (00: reserved) */
   scsiDirectionOut        = 0x80000000, /* data direction (10: DATA OUT) */
   scsiDirectionIn         = 0x40000000, /* data direction (01: DATA IN) */
   scsiDisableAutosense    = 0x20000000, /* disable auto sense feature */
   scsiFlagReservedA       = 0x10000000,
   scsiFlagReserved0       = 0x08000000,
   scsiCDBLinked           = 0x04000000, /* the PB contains a linked CDB */
   scsiQEnable             = 0x02000000, /* target queue actions enabled */
   scsiCDBIsPointer        = 0x01000000, /* the CDB field is a pointer */
   scsiFlagReserved1       = 0x00800000,
   scsiInitiateSyncData    = 0x00400000, /* attempt sync data xfer & SDTR */
   scsiDisableSyncData     = 0x00200000, /* disable sync, go to async */
   scsiSIMQHead            = 0x00100000, /* place PB at the head of queue */
   scsiSIMQFreeze          = 0x00080000, /* freeze the SIM queue */
   scsiSIMQNoFreeze        = 0x00040000, /* disallow SIM Q freezing */
   scsiDoDisconnect        = 0x00020000, /* definitely do disconnect */
   scsiDontDisconnect      = 0x00010000, /* definitely don't disconnect */
   scsiDataReadyForDMA     = 0x00008000, /* buffer(s) are ready for DMA */
   scsiFlagReserved3       = 0x00004000,
   scsiDataPhysical        = 0x00002000, /* S/G buffer ptrs are physical */
   scsiSensePhysical       = 0x00001000, /* autosense ptr is physical */
   scsiFlagReserved5       = 0x00000800,
   scsiFlagReserved6       = 0x00000400,
   scsiFlagReserved7       = 0x00000200,
   scsiFlagReserved8       = 0x00000100
};

/* bit masks for the scsiIOFlags field */
enum {
   scsiNoParityCheck       = 0x0002,   /* disable parity checking */
   scsiDisableSelectWAtn   = 0x0004,   /* disable select w/Atn */
   scsiSavePtrOnDisconnect = 0x0008,   /* SaveDataPointer on disconnect */
   scsiNoBucketIn          = 0x0010,   /* don’t bit-bucket on input */
   scsiNoBucketOut         = 0x0020,   /* don’t bit-bucket on output */
```

```
   scsiDisableWide         = 0x0040,   /* disable wide negotiation */
   scsiInitiateWide        = 0x0080,   /* initiate wide negotiation */
   scsiRenegotiateSense    = 0x0100,   /* renegotiate sync/wide */
   scsiIOFlagReserved0080  = 0x0080,
   scsiIOFlagReserved8000  = 0x8000
};

/* SIM queue actions. */
enum {
   scsiSimpleQTag          = 0x20,     /* tag for a simple queue */
   scsiHeadQTag            = 0x21,     /* tag for head of queue  */
   scsiOrderedQTag         = 0x22      /* tag for ordered queue */
};

/* scsiHBAInquiry field bits */
enum {
   scsiBusMDP        = 0x80,     /* supports Modify Data Pointer message */
   scsiBusWide32     = 0x40,     /* supports 32-bit wide SCSI */
   scsiBusWide16     = 0x20,     /* supports 16-bit wide SCSI */
   scsiBusSDTR       = 0x10,     /* supports SDTR message */
   scsiBusLinkedCDB  = 0x08,     /* supports linked CDBs */
   scsiBusTagQ       = 0x02,     /* supports tag queue message */
   scsiBusSoftReset  = 0x01      /* supports soft reset */
};

/* scsiDataTypes field bits  */
/* bits 0-15 Apple-defined, 16-30 vendor unique, 31 = reserved */
enum {
   scsiBusDataBuffer       = (1<<scsiDataBuffer),  /* single buffer */
   scsiBusDataTIB          = (1<<scsiDataTIB), /* TIB (ptr in scsiDataPtr) */
   scsiBusDataSG           = (1<<scsiDataSG),      /* scatter/gather list */
   scsiBusDataReserved     = 0x80000000
};

/* scsiScanFlags field bits */
enum {
   scsiBusScansDevices     = 0x80, /* bus scans and maintains device list */
   scsiBusScansOnInit      = 0x40, /* bus scans at startup */
   scsiBusLoadsROMDrivers  = 0x20  /* may load ROM drivers for targets */
};
```

```
/* scsiFeatureFlags field bits */
enum {
   scsiBusInternalExternalMask   = 0x000000C0, /* internal/external mask*/
   scsiBusInternalExternalUnknown = 0x00000000, /* unknown if in or out */
   scsiBusInternalExternal       = 0x000000C0, /* both inside and outside */
   scsiBusInternal               = 0x00000080, /* bus goes inside the box */
   scsiBusExternal               = 0x00000040, /* bus goes outside the box */
   scsiBusCacheCoherentDMA       = 0x00000020, /* DMA is cache coherent */
   scsiBusOldCallCapable         = 0x00000010, /* SIM supports old API */
   scsiBusDifferential           = 0x00000004, /* uses differential bus */
   scsiBusFastSCSI               = 0x00000002, /* HBA supports fast SCSI */
   scsiBusDMAavailable           = 0x00000001  /* DMA is available */
};

/* scsiWeirdStuff field bits */
enum {
   /* disconnects on odd byte boundries are unsafe with DMA or blind reads */
   scsiOddDisconnectUnsafeRead1  = 0x0001,
   /* disconnects on odd byte boundries unsafe with DMA or blind writes */
   scsiOddDisconnectUnsafeWrite1 = 0x0002,
   /* non-handshaked delays or disconnects on blind transfer may hang */
   scsiBusErrorsUnsafe           = 0x0004,
   /* non-handshaked delays or disconnects on blind transfer may corrupt */
   scsiRequiresHandshake         = 0x0008,
   /* targets that initiate synchronous negotiations are supported */
   scsiTargetDrivenSDTRSafe      = 0x0010
};

/* scsiHBAslotType values */
enum {
   scsiMotherboardBus            = 0x01,  /* a built-in Apple bus */
   scsiNuBus                     = 0x02,  /* a SIM on a NuBus card */
   scsiPDSBus                    = 0x03   /* a SIM on a PDS card  */
};

/* flags for the scsiDriverFlags field */
enum {
   scsiDeviceSensitive = 0x0001, /* only driver should access this device */
   scsiDeviceNoOldCallAccess = 0x0002 /* device does not support old API */
};
```

```
/* SCSI Phases (used by SIMs that support the original SCSI Manager) */
enum {
   kDataOutPhase,       /* encoded MSG, C/D, I/O bits */
   kDataInPhase,
   kCommandPhase,
   kStatusPhase,
   kPhaseIllegal0,
   kPhaseIllegal1,
   kMessageOutPhase,
   kMessageInPhase,
   kBusFreePhase,       /* additional phases */
   kArbitratePhase,
   kSelectPhase
};
```

Data Types

```
/* SCSI callback function prototypes */
typedef pascal void (*CallbackProc) (void * scsiPB);
typedef void   (*AENCallbackProc) (void);
typedef OSErr  (*SIMInitProc) (Ptr SIMinfoPtr);
typedef void   (*SIMActionProc) (void * scsiPB, Ptr SIMGlobals);
typedef void   (*SCSIProc) (void );
typedef void   (*MakeCallbackProc) (void * scsiPB);
typedef SInt32 (*InterruptPollProc) (Ptr SIMGlobals);

struct DeviceIdent
{
   UInt8 diReserved;               /* reserved */
   UInt8 bus;                      /* SCSI - bus number */
   UInt8 targetID;                 /* SCSI - target SCSI ID */
   UInt8 LUN;                      /* SCSI - logical unit number */
};
typedef struct DeviceIdent DeviceIdent;

union CDB
{
   UInt8 *cdbPtr;                  /* pointer to the CDB, or */
   UInt8 cdbBytes[maxCDBLength];   /* the actual CDB to send */
};
typedef union CDB CDB, *CDBPtr;
```

```
struct SGRecord
{
   Ptr      SGAddr;             /* scatter/gather buffer address */
   UInt32   SGCount;            /* buffer size */
};
typedef struct SGRecord SGRecord;

#define SCSIPBHdr \
   struct SCSIHdr *qLink;        /*    internal use, must be nil */      \
   SInt16   scsiReserved1;       /* -> reserved for input */             \
   UInt16   scsiPBLength;        /* -> length of the entire PB  */       \
   UInt8    scsiFunctionCode;    /* -> function selector */              \
   UInt8    scsiReserved2;       /* <- reserved for output*/             \
   OSErr    scsiResult;          /* <- returned result */                \
   DeviceIdent   scsiDevice;     /* -> device ID (bus+target+LUN) */     \
   CallbackProc  scsiCompletion; /* -> completion routine pointer */     \
   UInt32   scsiFlags;           /* -> assorted flags */                 \
   UInt8    *scsiDriverStorage;  /* <> pointer for driver private use */ \
   Ptr      scsiXPTprivate;      /*    private field for XPT */          \
   SInt32   scsiReserved3;       /*    reserved */

struct SCSI_PB
{
   SCSIPBHdr
};
typedef struct SCSI_PB SCSI_PB;

#define SCSI_IO_Macro \
   SCSIPBHdr                     /*    header information fields */     \
   UInt16   scsiResultFlags;     /* <- flags that modify scsiResult */  \
   UInt16   scsiReserved12;      /* -> reserved */                       \
   UInt8    *scsiDataPtr;        /* -> data pointer */                   \
   UInt32   scsiDataLength;      /* -> data transfer length */           \
   UInt8    *scsiSensePtr;       /* -> autosense data buffer pointer */ \
   UInt8    scsiSenseLength;     /* -> size of the autosense buffer */  \
   UInt8    scsiCDBLength;       /* -> number of bytes for the CDB */   \
   UInt16   scsiSGListCount;     /* -> number of S/G list entries */    \
   UInt32   scsiReserved4;       /* <- reserved for output */            \
   UInt8    scsiSCSIstatus;      /* <- returned SCSI device status */   \
   SInt8    scsiSenseResidual;   /* <- autosense residual length */     \
   UInt16   scsiReserved5;       /* <- reserved for output */            \
   SInt32   scsiDataResidual;    /* <- data residual length */          \
   CDB      scsiCDB;             /* -> actual CDB or pointer to CDB */  \
   SInt32   scsiTimeout;         /* -> timeout value */                  \
```

```
   UInt8    *scsiReserved13;     /* -> reserved */                      \
   UInt16   scsiReserved14;      /* -> reserved */                      \
   UInt16   scsiIOFlags;         /* -> additional I/O flags */          \
   UInt8    scsiTagAction;       /* -> what to do for tag queuing */    \
   UInt8    scsiReserved6;       /* -> reserved for input */            \
   UInt16   scsiReserved7;       /* -> reserved for input */            \
   UInt16   scsiSelectTimeout;   /* -> select timeout value */          \
   UInt8    scsiDataType;        /* -> data description type */         \
   UInt8    scsiTransferType;    /* -> transfer type (blind/polled) */  \
   UInt32   scsiReserved8;       /* -> reserved for input */            \
   UInt32   scsiReserved9;       /* -> reserved for input */            \
   UInt16   scsiHandshake[handshakeDataLength]; /* -> handshake info */ \
   UInt32   scsiReserved10;      /* -> reserved for input */            \
   UInt32   scsiReserved11;      /* -> reserved for input */            \
   struct   SCSI_IO *scsiCommandLink; /* -> linked command pointer */   \
   UInt8    scsiSIMpublics[8];   /* -> reserved for SIM input */        \
   UInt8    scsiAppleReserved6[8];    /* -> reserved for input */       \
   /* XPT private fields for original SCSI Manager emulation */         \
   UInt16   scsiCurrentPhase;    /* <- bus phase after old call */      \
   SInt16   scsiSelector;        /* -> selector for old call */         \
   OSErr    scsiOldCallResult;   /* <- result of old call */            \
   UInt8    scsiSCSImessage;     /* <- SCSIComplete message byte */     \
   UInt8    XPTprivateFlags;     /* <> XPT private flags */             \
   UInt8    XPTextras[12];       /*    reserved */

struct SCSI_IO
{
   SCSI_IO_Macro
};
typedef struct SCSI_IO SCSI_IO;
typedef SCSI_IO SCSIExecIOPB;

struct SCSIBusInquiryPB
{
   SCSIPBHdr                     /*    header information fields */
   UInt16   scsiEngineCount;     /* <- number of engines on HBA */
   UInt16   scsiMaxTransferType; /* <- number of xfer types supported */
   UInt32   scsiDataTypes;       /* <- data types supported by this SIM */
   UInt16   scsiIOpbSize;        /* <- size of SCSI_IO PB for this SIM */
   UInt16   scsiMaxIOpbSize;     /* <- largest SCSI_IO PB for all SIMs */
   UInt32   scsiFeatureFlags;    /* <- supported features flags field */
   UInt8    scsiVersionNumber;   /* <- version number for the SIM/HBA */
   UInt8    scsiHBAInquiry;      /* <- mimic of INQ byte 7 for the HBA */
   UInt8    scsiTargetModeFlags; /* <- flags for target mode support */
```

```
   UInt8    scsiScanFlags;          /* <- scan related feature flags */
   UInt32   scsiSIMPrivatesPtr;     /* <- pointer to SIM private data */
   UInt32   scsiSIMPrivatesSize;    /* <- size of SIM private data */
   UInt32   scsiAsyncFlags;         /* <- reserved for input */
   UInt8    scsiHiBusID;            /* <- highest path ID in the subsystem */
   UInt8    scsiInitiatorID;        /* <- ID of the HBA on the SCSI bus */
   UInt16   scsiBIReserved0;        /*    reserved */
   UInt32   scsiBIReserved1;        /*    reserved */
   UInt32   scsiFlagsSupported;     /* <- which scsiFlags are supported */
   UInt16   scsiIOFlagsSupported;   /* <- which scsiIOFlags are supported */
   UInt16   scsiWeirdStuff;         /* <- flags for strange behavior */
   UInt16   scsiMaxTarget;          /* <- maximum target ID supported */
   UInt16   scsiMaxLUN;             /* <- maximum LUN supported */
   SInt8 scsiSIMVendor[vendorIDLength]; /* <- vendor ID of the SIM */
   SInt8 scsiHBAVendor[vendorIDLength]; /* <- vendor ID of the HBA */
   SInt8 scsiControllerFamily[vendorIDLength]; /* <- controller family */
   SInt8 scsiControllerType[vendorIDLength];   /* <- controller model */
   SInt8 scsiXPTversion[4];         /* <- version number of XPT */
   SInt8 scsiSIMversion[4];         /* <- version number of SIM */
   SInt8 scsiHBAversion[4];         /* <- version number of HBA */
   UInt8    scsiHBAslotType;        /* <- type of slot this HBA is in */
   UInt8    scsiHBAslotNumber;      /* <- slot number of this HBA */
   UInt16   scsiSIMsRsrcID;         /* <- sResource ID of this SIM */
   UInt16   scsiBIReserved3;        /* <- reserved for input */
   UInt16   scsiAdditionalLength;   /* <- additional length of PB */
};
typedef struct SCSIBusInquiryPB SCSIBusInquiryPB;

struct SCSIAbortCommandPB
{
   SCSIPBHdr               /* header information fields */
   SCSI_IO  *scsiIOptr;    /* -> pointer to the PB to abort */
};
typedef struct SCSIAbortCommandPB SCSIAbortCommandPB;

struct SCSITerminateIOPB
{
   SCSIPBHdr               /* header information fields */
   SCSI_IO  *scsiIOptr;    /* -> pointer to the PB to terminate */
};
typedef struct SCSITerminateIOPB SCSITerminateIOPB;
```

```
struct SCSIGetVirtualIDInfoPB
{
   SCSIPBHdr               /* header information fields */
   UInt16   scsiOldCallID; /* -> SCSI ID of device in question */
   Boolean  scsiExists;    /* <- true if device exists */
};
typedef struct SCSIGetVirtualIDInfoPB SCSIGetVirtualIDInfoPB;

struct SCSIDriverPB
{
   SCSIPBHdr                     /* header information fields */
   SInt16      scsiDriver;       /* -> driver refNum, for CreateRefNumXref */
                                 /* <- for LookupRefNumXref */
   UInt16      scsiDriverFlags;  /* <> details of driver/device */
   DeviceIdent scsiNextDevice;   /* <- DeviceIdent of the next driver */
};
typedef struct SCSIDriverPB SCSIDriverPB;

struct SCSILoadDriverPB
{
   SCSIPBHdr                     /* header information fields */
   SInt16   scsiLoadedRefNum;    /* <- SIM returns driver reference number */
   Boolean  scsiDiskLoadFailed;  /* -> if true, previous call failed */
};
typedef struct SCSILoadDriverPB SCSILoadDriverPB;

struct SIMInitInfo
{
   UInt8          *SIMstaticPtr;  /* <- pointer to the SIM's static data */
   SInt32         staticSize;     /* -> size requested for SIM static data */
   SIMInitProc    SIMInit;        /* -> pointer to the SIMInit function */
   SIMActionProc  SIMAction;      /* -> pointer to the SIMAction function */
   SCSIProc       SIM_ISR;        /*    reserved */
   InterruptPollProc SIMInterruptPoll; /* -> pointer to SIMInterruptPoll */
   SIMActionProc  NewOldCall;     /* -> pointer to NewOldCall function */
   UInt16         ioPBSize;       /* -> size of SCSI_IO PB for this SIM */
   Boolean        oldCallCapable; /* -> true if SIM handles old-API calls */
   UInt8          simInfoUnused1; /*    reserved */
   SInt32         simInternalUse; /*    not affected or viewed by XPT */
   SCSIProc       XPT_ISR;        /*    reserved */
   SCSIProc       EnteringSIM;    /* <- pointer to EnteringSIM function */
   SCSIProc       ExitingSIM;     /* <- pointer to ExitingSIM function */
   MakeCallbackProc MakeCallback; /* <- pointer to MakeCallback function */
   UInt16         busID;          /* <- bus number for the registered bus */
```

```
   UInt16         simInfoUnused3; /* <- reserved */
   SInt32         simInfoUnused4; /* <- reserved */
};
typedef struct SIMInitInfo SIMInitInfo;
```

Functions

```
OSErr SCSIAction        (SCSI_PB *scsiPB);
OSErr SCSIRegisterBus   (SIMInitInfo *SIMinfoPtr);
OSErr SCSIDeregisterBus (SCSI_PB *scsiPB);
OSErr SCSIReregisterBus (SIMInitInfo *SIMinfoPtr);
OSErr SCSIKillXPT       (void *);
```

Pascal Summary

Constants

```
CONST
   scsiVERSION    = 43;

   {SCSI Manager function codes}
   SCSINop                    = $00;   {no operation}
   SCSIExecIO                 = $01;   {execute a SCSI IO transaction}
   SCSIBusInquiry             = $03;   {bus inquiry}
   SCSIReleaseQ               = $04;   {release a frozen SIM queue}
   SCSIAbortCommand           = $10;   {abort a SCSI command}
   SCSIResetBus               = $11;   {reset the SCSI bus}
   SCSIResetDevice            = $12;   {reset a SCSI device}
   SCSITerminateIO            = $13;   {terminate I/O transaction}
   SCSIGetVirtualIDInfo       = $80;   {return DeviceIdent of virtual ID}
   SCSILoadDriver             = $82;   {load a driver from a SCSI device}
   SCSIOldCall                = $84;   {begin old-API emulation}
   SCSICreateRefNumXref       = $85;   {register a device driver}
   SCSILookupRefNumXref       = $86;   {find a driver reference number}
   SCSIRemoveRefNumXref       = $87;   {deregister a device driver}
   SCSIRegisterWithNewXPT     = $88;   {XPT replaced; SIM must reregister}
   vendorUnique               = $C0;   {$C0 through $FF}
```

```
{allocation lengths for parameter block fields}
handshakeDataLength        = 8;     {handshake data length}
maxCDBLength               = 16;    {space for the CDB bytes/pointer}
vendorIDLength             = 16;    {ASCII string length for Vendor ID}

{types for the scsiTransferType field}
scsiTransferBlind          = 0;     {DMA if available, otherwise blind}
scsiTransferPolled         = 1;     {polled}

{types for the scsiDataType field}
scsiDataBuffer             = 0;     {single contiguous buffer supplied}
scsiDataTIB                = 1;     {TIB supplied (ptr in scsiDataPtr)}
scsiDataSG                 = 2;     {scatter/gather list supplied}

{flags for the scsiResultFlags field}
scsiSIMQFrozen             = $0001; {the SIM queue is frozen}
scsiAutosenseValid         = $0002; {autosense data valid for target}
scsiBusNotFree             = $0004; {SCSI bus is not free}

{bit numbers in the scsiFlags field}
kbSCSIDisableAutosense     = 29;    {disable auto sense feature}
kbSCSIFlagReservedA        = 28;
kbSCSIFlagReserved0        = 27;
kbSCSICDBLinked            = 26;    {the PB contains a linked CDB}
kbSCSIQEnable              = 25;    {target queue actions are enabled}
kbSCSICDBIsPointer         = 24;    {the CDB field contains a pointer}
kbSCSIFlagReserved1        = 23;
kbSCSIInitiateSyncData     = 22;    {attempt sync data transfer and SDTR}
kbSCSIDisableSyncData      = 21;    {disable sync, go to async}
kbSCSISIMQHead             = 20;    {place PB at the head of SIM queue}
kbSCSISIMQFreeze           = 19;    {freeze the SIM queue}
kbSCSISIMQNoFreeze         = 18;    {disable SIM queue freezing}
kbSCSIDoDisconnect         = 17;    {definitely do disconnect}
kbSCSIDontDisconnect       = 16;    {definitely don't disconnect}
kbSCSIDataReadyForDMA      = 15;    {data buffer(s) are ready for DMA}
kbSCSIFlagReserved3        = 14;
kbSCSIDataPhysical         = 13;    {S/G buffer data ptrs are physical}
kbSCSISensePhysical        = 12;    {autosense buffer ptr is physical}
kbSCSIFlagReserved5        = 11;
kbSCSIFlagReserved6        = 10;
kbSCSIFlagReserved7        = 9;
kbSCSIFlagReserved8        = 8;
kbSCSIDataBufferValid      = 7;     {data buffer valid}
```

```
kbSCSIStatusBufferValid    = 6;     {status buffer valid}
kbSCSIMessageBufferValid   = 5;     {message buffer valid}
kbSCSIFlagReserved9        = 4;

{bit masks for the scsiFlags field}
scsiDirectionMask          = $C0000000;   {data direction mask}
scsiDirectionNone          = $C0000000;   {data direction (11: no data)}
scsiDirectionReserved      = $00000000;   {data direction (00: reserved)}
scsiDirectionOut           = $80000000;   {data direction (10: DATA OUT)}
scsiDirectionIn            = $40000000;   {data direction (01: DATA IN)}
scsiDisableAutosense       = $20000000;   {disable auto sense feature}
scsiFlagReservedA          = $10000000;
scsiFlagReserved0          = $08000000;
scsiCDBLinked              = $04000000;   {the PB contains a linked CDB}
scsiQEnable                = $02000000;   {target queue actions enabled}
scsiCDBIsPointer           = $01000000;   {the CDB field is a pointer}
scsiFlagReserved1          = $00800000;
scsiInitiateSyncData       = $00400000;   {attempt sync data xfer & SDTR}
scsiDisableSyncData        = $00200000;   {disable sync; go to async}
scsiSIMQHead               = $00100000;   {place PB at the head of queue}
scsiSIMQFreeze             = $00080000;   {freeze the SIM queue}
scsiSIMQNoFreeze           = $00040000;   {disallow SIM Q freezing}
scsiDoDisconnect           = $00020000;   {definitely do disconnect}
scsiDontDisconnect         = $00010000;   {definitely don't disconnect}
scsiDataReadyForDMA        = $00008000;   {buffer(s) are ready for DMA}
scsiFlagReserved3          = $00004000;
scsiDataPhysical           = $00002000;   {S/G buffer ptrs are physical}
scsiSensePhysical          = $00001000;   {autosense ptr is physical}
scsiFlagReserved5          = $00000800;
scsiFlagReserved6          = $00000400;
scsiFlagReserved7          = $00000200;
scsiFlagReserved8          = $00000100;

{bit masks for the scsiIOFlags field}
scsiNoParityCheck          = $0002;       {disable parity checking}
scsiDisableSelectWAtn      = $0004;       {disable select w/Atn}
scsiSavePtrOnDisconnect    = $0008;       {SaveDataPointer on disconnect}
scsiNoBucketIn             = $0010;       {don’t bit-bucket on input}
scsiNoBucketOut            = $0020;       {don’t bit-bucket on output}
scsiDisableWide            = $0040;       {disable wide negotiation}
scsiInitiateWide           = $0080;       {initiate wide negotiation}
scsiRenegotiateSense       = $0100;       {renegotiate sync/wide}
scsiIOFlagReserved0080     = $0080;
scsiIOFlagReserved8000     = $8000;
```

```
{SIM queue actions}
scsiSimpleQTag              = $20;   {tag for a simple queue}
scsiHeadQTag                = $21;   {tag for head of queue}
scsiOrderedQTag             = $22;   {tag for ordered queue}

{scsiHBAInquiry field bits}
scsiBusMDP                  = $80;   {supports Modify Data Pointer message}
scsiBusWide32               = $40;   {supports 32-bit wide SCSI}
scsiBusWide16               = $20;   {supports 16-bit wide SCSI}
scsiBusSDTR                 = $10;   {supports SDTR message}
scsiBusLinkedCDB            = $08;   {supports linked CDBs}
scsiBusTagQ                 = $02;   {supports tag queue message}
scsiBusSoftReset            = $01;   {supports soft reset}

{scsiDataTypes field bits}
{bits 0-15 Apple-defined, 16-30 vendor unique, 31 = reserved}
scsiBusDataBuffer           = $00000001; {single buffer}
scsiBusDataTIB              = $00000002; {TIB (pointer in scsiDataPtr)}
scsiBusDataSG               = $00000004; {scatter/gather list}
scsiBusDataReserved         = $80000000;

{scsiScanFlags field bits}
scsiBusScansDevices         = $80;   {bus scans and maintains device list}
scsiBusScansOnInit          = $40;   {bus scans at startup}
scsiBusLoadsROMDrivers      = $20;   {may load ROM drivers for targets}

{scsiFeatureFlags field bits}
scsiBusInternalExternalMask      = $000000C0; {internal/external mask}
scsiBusInternalExternalUnknown   = $00000000; {unknown if in or out}
scsiBusInternalExternal          = $000000C0; {both inside and outside}
scsiBusInternal                  = $00000080; {bus goes inside the box}
scsiBusExternal                  = $00000040; {bus goes outside the box}
scsiBusCacheCoherentDMA          = $00000020; {DMA is cache coherent}
scsiBusOldCallCapable            = $00000010; {SIM supports old-API}
scsiBusDifferential              = $00000004; {uses differential bus}
scsiBusFastSCSI                  = $00000002; {HBA supports fast SCSI}
scsiBusDMAavailable              = $00000001; {DMA is available}

{scsiWeirdStuff field bits}
scsiOddDisconnectUnsafeRead1     = $0001; {odd byte disconnects unsafe}
scsiOddDisconnectUnsafeWrite1    = $0002; {odd byte disconnects unsafe}
scsiBusErrorsUnsafe              = $0004; {delays or disconnects may hang}
scsiRequiresHandshake            = $0008; {delays/disconnects may corrupt}
scsiTargetDrivenSDTRSafe         = $0010; {target-driven STDR supported}
```

```
   {scsiHBAslotType values}
   scsiMotherboardBus         = $01;   {a built-in Apple bus}
   scsiNuBus                  = $02;   {a SIM on a NuBus card}
   scsiPDSBus                 = $03;   {a SIM on a PDS card}

   {flags for the scsiDriverFlags field}
   scsiDeviceSensitive        = $0001; {only driver should access the device}
   scsiDeviceNoOldCallAccess  = $0002; {device does not support old API}

   {SCSI Phases (used by SIMs that support the original SCSI Manager)}
   kDataOutPhase              = $00;   {encoded MSG, C/D, I/O bits}
   kDataInPhase               = $01;
   kCommandPhase              = $02;
   kStatusPhase               = $03;
   kPhaseIllegal0             = $04;
   kPhaseIllegal1             = $05;
   kMessageOutPhase           = $06;
   kMessageInPhase            = $07;
   kBusFreePhase              = $08;   {additional phases}
   kArbitratePhase            = $09;
   kSelectPhase               = $0A;
```

Data Types

```
TYPE
   {SCSI callback function prototypes}
   CallbackProc               = ProcPtr;
   AENCallbackProc            = ProcPtr;
   SIMInitProc                = ProcPtr;
   SIMActionProc              = ProcPtr;
   SCSIProc                   = ProcPtr;
   MakeCallbackProc           = ProcPtr;
   InterruptPollProc          = ProcPtr;

TYPE
   DI =
   PACKED RECORD
      diReserved:             Byte;    {reserved}
      bus:                    Byte;    {SCSI - bus number}
      targetID:               Byte;    {SCSI - target SCSI ID}
      LUN:                    Byte;    {SCSI - logical unit number}
   END;
   DeviceIdent = DI;
```

```
CDBRec =
PACKED RECORD
CASE Integer OF
   0: cdbPtr:           ^Byte;                 {pointer to the CDB, or}
   1: cdbBytes:         ARRAY [0..15] OF Byte; {the actual CDB to send}
END;
CDB = CDBRec;
CDBPtr = ^CDBRec;

SGR =
PACKED RECORD
   SGAddr:              Ptr;           {scatter/gather buffer address}
   SGCount:             LongInt;       {buffer size}
END;
SGRecord = SGR;

SCSIHdr =
PACKED RECORD
   qLink:               ^SCSIHdr;      {   internal use, must be NIL}
   scsiReserved1:       Integer;       {-> reserved for input}
   scsiPBLength:        Integer;       {-> length of the entire PB}
   scsiFunctionCode:    Byte;          {-> function selector}
   scsiReserved2:       Byte;          {<- reserved for output}
   scsiResult:          OSErr;         {<- returned result}
   scsiDevice:          DeviceIdent;   {-> device ID (bus+target+LUN)}
   scsiCompletion:      CallbackProc;  {-> completion routine pointer}
   scsiFlags:           LongInt;       {-> assorted flags}
   scsiDriverStorage:   ^Byte;         {<> pointer for driver private use}
   scsiXPTprivate:      Ptr;           {   private field for XPT}
   scsiReserved3:       LongInt;       {   reserved}
END;
SCSI_PB = SCSIHdr;

SCSI_IO =
PACKED RECORD
   qLink:               ^SCSIHdr;      {   internal use, must be NIL}
   scsiReserved1:       Integer;       {-> reserved for input}
   scsiPBLength:        Integer;       {-> length of the entire PB}
   scsiFunctionCode:    Byte;          {-> function selector}
   scsiReserved2:       Byte;          {<- reserved for output}
   scsiResult:          OSErr;         {<- returned result}
   scsiDevice:          DeviceIdent;   {-> device ID (bus+target+LUN)}
   scsiCompletion:      CallbackProc;  {-> completion routine pointer}
   scsiFlags:           LongInt;       {-> assorted flags}
```

```
   scsiDriverStorage:   ^Byte;         {<> pointer for driver private use}
   scsiXPTprivate:      Ptr;           {   private field for XPT}
   scsiReserved3:       LongInt;       {   reserved}
   scsiResultFlags:     Integer;       {<- flags that modify scsiResult}
   scsiReserved12:      Integer;       {-> reserved}
   scsiDataPtr:         ^Byte;         {-> data pointer}
   scsiDataLength:      LongInt;       {-> data transfer length}
   scsiSensePtr:        ^Byte;         {-> autosense data buffer pointer}
   scsiSenseLength:     Byte;          {-> size of the autosense buffer}
   scsiCDBLength:       Byte;          {-> number of bytes for the CDB}
   scsiSGListCount:     Integer;       {-> number of S/G list entries}
   scsiReserved4:       LongInt;       {<- reserved for output}
   scsiSCSIstatus:      Byte;          {<- returned SCSI device status}
   scsiSenseResidual:   Char;          {<- autosense residual length}
   scsiReserved5:       Integer;       {<- reserved for output}
   scsiDataResidual:    LongInt;       {<- data residual length}
   scsiCDB:             CDB;           {-> actual CDB or pointer to CDB}
   scsiTimeout:         LongInt;       {-> timeout value}
   scsiReserved13:      ^Byte;         {-> reserved}
   scsiReserved14:      Integer;       {-> reserved}
   scsiIOFlags:         Integer;       {-> additional I/O flags}
   scsiTagAction:       Byte;          {-> what to do for tag queuing}
   scsiReserved6:       Byte;          {-> reserved for input}
   scsiReserved7:       Integer;       {-> reserved for input}
   scsiSelectTimeout:   Integer;       {-> select timeout value}
   scsiDataType:        Byte;          {-> data description type}
   scsiTransferType:    Byte;          {-> transfer type (blind/polled)}
   scsiReserved8:       LongInt;       {-> reserved for input}
   scsiReserved9:       LongInt;       {-> reserved for input}
   scsiHandshake:       ARRAY [0..7] OF Integer; {-> handshake info}
   scsiReserved10:      LongInt;       {-> reserved for input}
   scsiReserved11:      LongInt;       {-> reserved for input}
   scsiCommandLink:     ^SCSI_IO;      {-> linked command pointer}
   scsiSIMpublics:      ARRAY [0..7] OF Byte; {-> reserved for SIM input}
   scsiAppleReserved6:  ARRAY [0..7] OF Byte; {-> reserved for input}
   scsiCurrentPhase:    Integer;       {<- bus phase after old call}
   scsiSelector:        Integer;       {-> selector for old call}
   scsiOldCallResult:   OSErr;         {<- result of old call}
   scsiSCSImessage:     Byte;          {<- SCSIComplete message byte}
   XPTprivateFlags:     Byte;          {<> XPT private flags}
   XPTextras:           ARRAY [0..11] OF Byte; {reserved}
END;
SCSIExecIOPB = SCSI_IO;
```

```
SCSIBusInquiryPB =
PACKED RECORD
   qLink:                 ^SCSIHdr;     {   internal use, must be NIL}
   scsiReserved1:         Integer;      {-> reserved for input}
   scsiPBLength:          Integer;      {-> length of the entire PB}
   scsiFunctionCode:      Byte;         {-> function selector}
   scsiReserved2:         Byte;         {<- reserved for output}
   scsiResult:            OSErr;        {<- returned result}
   scsiDevice:            DeviceIdent;  {-> device ID (bus+target+LUN)}
   scsiCompletion:        CallbackProc; {-> completion routine pointer}
   scsiFlags:             LongInt;      {-> assorted flags}
   scsiDriverStorage:     ^Byte;        {<> pointer for driver private use}
   scsiXPTprivate:        Ptr;          {   private field for XPT}
   scsiReserved3:         LongInt;      {   reserved}
   scsiEngineCount:       Integer;      {<- number of engines on HBA}
   scsiMaxTransferType:   Integer;      {<- number of xfer types supported}
   scsiDataTypes:         LongInt;      {<- data types supported by SIM}
   scsiIOpbSize:          Integer;      {<- size of SCSI_IO PB for SIM}
   scsiMaxIOpbSize:       Integer;      {<- largest SCSI_IO PB registered}
   scsiFeatureFlags:      LongInt;      {<- supported features flags field}
   scsiVersionNumber:     Byte;         {<- version number for the SIM/HBA}
   scsiHBAInquiry:        Byte;         {<- mimic of INQ byte 7 for HBA}
   scsiTargetModeFlags:   Byte;         {<- flags for target mode support}
   scsiScanFlags:         Byte;         {<- scan related feature flags}
   scsiSIMPrivatesPtr:    LongInt;      {<- pointer to SIM private data}
   scsiSIMPrivatesSize:   LongInt;      {<- size of SIM private data}
   scsiAsyncFlags:        LongInt;      {<- reserved for input}
   scsiHiBusID:           Byte;         {<- highest bus ID registered}
   scsiInitiatorID:       Byte;         {<- ID of the HBA on the SCSI bus}
   scsiBIReserved0:       Integer;      {   reserved}
   scsiBIReserved1:       LongInt;      {   reserved}
   scsiFlagsSupported:    LongInt;      {<- which scsiFlags are supported}
   scsiIOFlagsSupported:  Integer;      {<- which scsiIOFlags supported}
   scsiWeirdStuff:        Integer;      {<- flags for strange behavior}
   scsiMaxTarget:         Integer;      {<- maximum target ID supported}
   scsiMaxLUN:            Integer;      {<- maximum LUN supported}
   scsiSIMVendor:         ARRAY [0..15] OF Char; {<- vendor ID of the SIM}
   scsiHBAVendor:         ARRAY [0..15] OF Char; {<- vendor ID of the HBA}
   scsiControllerFamily:  ARRAY [0..15] OF Char; {<- controller family}
   scsiControllerType:    ARRAY [0..15] OF Char; {<- controller model}
   scsiXPTversion:        ARRAY [0..3] OF Char;  {<- version number of XPT}
   scsiSIMversion:        ARRAY [0..3] OF Char;  {<- version number of SIM}
   scsiHBAversion:        ARRAY [0..3] OF Char;  {<- version number of HBA}
```

```
   scsiHBAslotType:      Byte;         {<- type of slot this HBA is in}
   scsiHBAslotNumber:    Byte;         {<- slot number of this HBA}
   scsiSIMsRsrcID:       Integer;      {<- sResource ID of this SIM}
   scsiBIReserved3:      Integer;      {<- reserved for input}
   scsiAdditionalLength: Integer;      {<- additional length of PB}
END;

SCSIAbortCommandPB =
PACKED RECORD
   qLink:                ^SCSIHdr;     {   internal use, must be NIL}
   scsiReserved1:        Integer;      {-> reserved for input}
   scsiPBLength:         Integer;      {-> length of the entire PB}
   scsiFunctionCode:     Byte;         {-> function selector}
   scsiReserved2:        Byte;         {<- reserved for output}
   scsiResult:           OSErr;        {<- returned result}
   scsiDevice:           DeviceIdent;  {-> device ID (bus+target+LUN)}
   scsiCompletion:       CallbackProc; {-> completion routine pointer}
   scsiFlags:            LongInt;      {-> assorted flags}
   scsiDriverStorage:    ^Byte;        {<> pointer for driver private use}
   scsiXPTprivate:       Ptr;          {   private field for XPT}
   scsiReserved3:        LongInt;      {   reserved}
   scsiIOptr:            ^SCSI_IO;     {-> pointer to the PB to abort}
END;

SCSITerminateIOPB =
PACKED RECORD
   qLink:                ^SCSIHdr;     {   internal use, must be NIL}
   scsiReserved1:        Integer;      {-> reserved for input}
   scsiPBLength:         Integer;      {-> length of the entire PB}
   scsiFunctionCode:     Byte;         {-> function selector}
   scsiReserved2:        Byte;         {<- reserved for output}
   scsiResult:           OSErr;        {<- returned result}
   scsiDevice:           DeviceIdent;  {-> device ID (bus+target+LUN)}
   scsiCompletion:       CallbackProc; {-> completion routine pointer}
   scsiFlags:            LongInt;      {-> assorted flags}
   scsiDriverStorage:    ^Byte;        {<> pointer for driver private use}
   scsiXPTprivate:       Ptr;          {   private field for XPT}
   scsiReserved3:        LongInt;      {   reserved}
   scsiIOptr:            ^SCSI_IO;     {-> pointer to the PB to terminate}
END;

SCSIGetVirtualIDInfoPB =
PACKED RECORD
   qLink:                ^SCSIHdr;     {   internal use, must be NIL}
```

```
   scsiReserved1:       Integer;       {-> reserved for input}
   scsiPBLength:        Integer;       {-> length of the entire PB}
   scsiFunctionCode:    Byte;          {-> function selector}
   scsiReserved2:       Byte;          {<- reserved for output}
   scsiResult:          OSErr;         {<- returned result}
   scsiDevice:          DeviceIdent;   {-> device ID (bus+target+LUN)}
   scsiCompletion:      CallbackProc;  {-> completion routine pointer}
   scsiFlags:           LongInt;       {-> assorted flags}
   scsiDriverStorage:   ^Byte;         {<> pointer for driver private use}
   scsiXPTprivate:      Ptr;           {   private field for XPT}
   scsiReserved3:       LongInt;       {   reserved}
   scsiOldCallID:       Integer;       {-> SCSI ID of device in question}
   scsiExists:          Boolean;       {<- true if device exists}
END;

SCSIDriverPB =
PACKED RECORD
   qLink:               ^SCSIHdr;      {   internal use, must be NIL}
   scsiReserved1:       Integer;       {-> reserved for input}
   scsiPBLength:        Integer;       {-> length of the entire PB}
   scsiFunctionCode:    Byte;          {-> function selector}
   scsiReserved2:       Byte;          {<- reserved for output}
   scsiResult:          OSErr;         {<- returned result}
   scsiDevice:          DeviceIdent;   {-> device ID (bus+target+LUN)}
   scsiCompletion:      CallbackProc;  {-> completion routine pointer}
   scsiFlags:           LongInt;       {-> assorted flags}
   scsiDriverStorage:   ^Byte;         {<> pointer for driver private use}
   scsiXPTprivate:      Ptr;           {   private field for XPT}
   scsiReserved3:       LongInt;       {   reserved}
   scsiDriver:          Integer;       {<> driver reference number}
   scsiDriverFlags:     Integer;       {<> details of driver/device}
   scsiNextDevice:      DeviceIdent;   {<- DeviceIdent of the next driver}
END;

SCSILoadDriverPB =
PACKED RECORD
   qLink:               ^SCSIHdr;      {   internal use, must be NIL}
   scsiReserved1:       Integer;       {-> reserved for input}
   scsiPBLength:        Integer;       {-> length of the entire PB}
   scsiFunctionCode:    Byte;          {-> function selector}
   scsiReserved2:       Byte;          {<- reserved for output}
   scsiResult:          OSErr;         {<- returned result}
   scsiDevice:          DeviceIdent;   {-> device ID (bus+target+LUN)}
   scsiCompletion:      CallbackProc;  {-> completion routine pointer}
```

```
   scsiFlags:           LongInt;       {-> assorted flags}
   scsiDriverStorage:   ^Byte;         {<> pointer for driver private use}
   scsiXPTprivate:      Ptr;           {   private field for XPT}
   scsiReserved3:       LongInt;       {   reserved}
   scsiLoadedRefNum:    Integer;       {<- SIM returns driver refNum}
   scsiDiskLoadFailed:  Boolean;       {-> if true, previous call failed}
END;

SIMInitInfo =
PACKED RECORD
   SIMstaticPtr:        ^Byte;         {<- pointer to SIM's static data}
   staticSize:          LongInt;       {-> requested SIM static data size}
   SIMInit:             SIMInitProc;   {-> SIMInit function pointer}
   SIMAction:           SIMActionProc; {-> SIMAction function pointer}
   SIM_ISR:             SCSIProc;      {   reserved}
   SIMInterruptPoll:    InterruptPollProc; {-> SIMInterruptPoll function}
   NewOldCall:          SIMActionProc; {-> NewOldCall function pointer}
   ioPBSize:            Integer;       {-> size of SCSI_IO PB for SIM}
   oldCallCapable:      Boolean;       {-> true if SIM supports old-API}
   simInfoUnused1:      Byte;          {   reserved}
   simInternalUse:      LongInt;       {   not affected or viewed by XPT}
   XPT_ISR:             SCSIProc;      {   reserved}
   EnteringSIM:         SCSIProc;      {<- EnteringSIM function pointer}
   ExitingSIM:          SCSIProc;      {<- ExitingSIM function pointer}
   MakeCallback:        MakeCallbackProc;  {<- MakeCallback function ptr}
   busID:               Integer;       {<- bus number assigned by XPT}
   simInfoUnused3:      Integer;       {<- reserved}
   simInfoUnused4:      LongInt;       {<- reserved}
END;
```

Routines

```
FUNCTION SCSIAction        (VAR ioPtr: SCSI_PB): OSErr;
FUNCTION SCSIRegisterBus   (VAR ioPtr: SIMInitInfo): OSErr;
FUNCTION SCSIDeregisterBus (VAR ioPtr: SIMInitInfo): OSErr;
FUNCTION SCSIReregisterBus (VAR ioPtr: SIMInitInfo): OSErr;
FUNCTION SCSIKillXPT       (VAR ioPtr: SIMInitInfo): OSErr;
```

Assembly-Language Summary

Data Structures

The Device Identification Record

0	`diReserved`	byte	reserved
1	`bus`	byte	bus number
2	`targetID`	byte	target SCSI ID
3	`LUN`	byte	logical unit number

The Command Descriptor Block Record

0	`cdbPtr`	long	CDB buffer pointer
4	`cdbBytes`	16 bytes	CDB buffer

The Scatter/Gather List Element

0	`SGAddr`	long	buffer pointer
4	`SGCount`	long	buffer size

The SCSI Manager Parameter Block Header

0	`qLink`	long	used internally by the SCSI Manager
4	`scsiReserved`	word	reserved
6	`scsiPBLength`	word	parameter block size
8	`scsiFunctionCode`	byte	function selector code
9	`scsiReserved2`	byte	reserved
10	`scsiResult`	word	result code
12	`scsiDevice`	4 bytes	device ID (bus number, target ID, LUN)
16	`scsiCompletion`	long	completion routine
20	`scsiFlags`	long	flags
24	`scsiDriverStorage`	long	driver private data
28	`scsiXPTprivate`	long	reserved
32	`scsiReserved3`	long	reserved

The SCSI I/O Parameter Block

0	`SCSIPBHdr`	36 bytes	parameter block header
36	`scsiResultFlags`	word	I/O result flags
38	`scsiReserved12`	word	reserved
40	`scsiDataPtr`	long	data buffer pointer
44	`scsiDataLength`	long	data buffer size
48	`scsiSensePtr`	long	autosense buffer pointer
52	`scsiSenseLength`	byte	autosense buffer size
53	`scsiCDBLength`	byte	CDB size
54	`scsiSGListCount`	word	number of scatter/gather list entries
56	`scsiReserved4`	long	reserved
60	`scsiSCSIstatus`	byte	SCSI device status

61	scsiSenseResidual	byte	autosense residual length
62	scsiReserved5	word	reserved
64	scsiDataResidual	long	data transfer residual length
68	scsiCDB	16 bytes	command descriptor block record
84	scsiTimeout	long	timeout value, in Time Manager format
88	scsiReserved13	long	reserved
92	scsiReserved14	long	reserved
94	scsiIOFlags	word	I/O flags
96	scsiTagAction	byte	reserved
97	scsiReserved6	byte	reserved
98	scsiReserved7	word	reserved
100	scsiSelectTimeout	word	selection timeout value, in milliseconds
102	scsiDataType	byte	data type of scsiDataPtr
103	scsiTransferType	byte	transfer mode (polled or blind)
104	scsiReserved8	long	reserved
108	scsiReserved9	long	reserved
112	scsiHandshake	16 bytes	handshaking instructions
128	scsiReserved10	long	reserved
132	scsiReserved1	long	reserved
136	scsiCommandLink	long	linked parameter block pointer
140	scsiSIMpublics	8 bytes	additional input to SIM
148	scsiAppleReserved6	8 bytes	reserved
156	scsiCurrentPhase	word	bus phase after original SCSI Manager function
158	scsiSelector	word	_SCSIDispatch selector for original function
160	scsiOldCallResult	word	result code of original function
162	scsiSCSImessage	byte	SCSIComplete message byte
163	XPTprivateFlags	byte	reserved
164	XPTextras	12 bytes	reserved

The SCSI Bus Inquiry Parameter Block

0	SCSIPBHdr	36 bytes	parameter block header
36	scsiEngineCount	word	number of engines on the HBA
38	scsiMaxTransferType	word	number of data transfer types supported
40	scsiDataTypes	long	bit map of supported data types
44	scsiIOpbSize	word	SCSI I/O parameter block size for this SIM
46	scsiMaxIOpbSize	word	largest parameter block for any registered SIM
48	scsiFeatureFlags	long	bus feature flags
52	scsiVersionNumber	byte	SIM/HBA version number
53	scsiHBAInquiry	byte	bus capability flags
54	scsiTargetModeFlags	byte	reserved
55	scsiScanFlags	byte	scan feature flags
56	scsiSIMPrivatesPtr	long	SIM private data pointer
60	scsiSIMPrivatesSize	long	SIM private data size
64	scsiAsyncFlags	long	reserved
68	scsiHiBusID	byte	highest registered bus number
69	scsiInitiatorID	byte	SCSI ID of the HBA
70	scsiBIReserved0	word	reserved
72	scsiBIReserved1	long	reserved
76	scsiFlagsSupported	long	bit map of supported scsiFlags

80	`scsiIOFlagsSupported`	word	bit map of supported `scsiIOFlags`
82	`scsiWeirdStuff`	word	miscellaneous flags
84	`scsiMaxTarget`	word	highest SCSI ID supported by the HBA
86	`scsiMaxLUN`	word	highest LUN supported by the HBA
88	`scsiSIMVendor`	16 bytes	SIM vendor string
104	`scsiHBAVendor`	16 bytes	HBA vendor string
120	`scsiControllerFamily`	16 bytes	SCSI controller family string
136	`scsiControllerType`	16 bytes	SCSI controller type string
152	`scsiXPTversion`	4 bytes	XPT version string
156	`scsiSIMversion`	4 bytes	SIM version string
160	`scsiHBAversion`	4 bytes	HBA version string
164	`scsiHBAslotType`	byte	HBA slot type
165	`scsiHBAslotNumber`	byte	HBA slot number
166	`scsiSIMsRsrcID`	word	SIM sResource ID
168	`scsiBIReserved3`	word	reserved
170	`scsiAdditionalLength`	word	additional size of the parameter block

The SCSI Abort Command Parameter Block

0	`SCSIPBHdr`	36 bytes	parameter block header
36	`scsiIOptr`	long	SCSI I/O parameter block pointer

The SCSI Terminate I/O Parameter Block

0	`SCSIPBHdr`	36 bytes	parameter block header
36	`scsiIOptr`	long	SCSI I/O parameter block pointer

The SCSI Virtual ID Information Parameter Block

0	`SCSIPBHdr`	36 bytes	parameter block header
36	`scsiOldCallID`	word	virtual SCSI ID of the device to search for
38	`scsiExists`	byte	Boolean (`true` if the device was found)

The SCSI Load Driver Parameter Block

0	`SCSIPBHdr`	36 bytes	parameter block header
36	`scsiLoadedRefNum`	word	driver reference number
38	`scsiDiskLoadFailed`	byte	Boolean (`true` if a driver could not be loaded)

The SCSI Driver Identification Parameter Block

0	`SCSIPBHdr`	36 bytes	parameter block header
36	`scsiDriver`	word	driver reference number
38	`scsiDriverFlags`	word	driver flags
40	`scsiNextDevice`	4 bytes	device ID of the next device in the list

The SIM Initialization Record

0	`SIMstaticPtr`	long	SIM private data pointer
4	`staticSize`	long	SIM private data size
8	`SIMInit`	long	`SIMInit` function pointer
12	`SIMAction`	long	`SIMAction` function pointer
16	`SIM_ISR`	long	reserved
20	`SIMInterruptPoll`	long	`SIMInterruptPoll` function pointer
24	`NewOldCall`	long	`NewOldCall` function pointer
28	`ioPBSize`	word	SCSI I/O parameter block size for this SIM
30	`oldCallCapable`	byte	Boolean (`true` if SIM accepts original functions)
31	`simInfoUnused1`	byte	reserved
32	`simInternalUse`	long	SIM private data
36	`XPT_ISR`	long	reserved
40	`EnteringSIM`	long	`EnteringSIM` function pointer
44	`ExitingSIM`	long	`ExitingSIM` function pointer
48	`MakeCallback`	long	`MakeCallback` function pointer
52	`busID`	word	bus number
54	`simInfoUnused3`	word	reserved
56	`simInfoUnused4`	long	reserved

Trap Macros

Trap Macros Requiring Routine Selectors

`_SCSIAtomic`

Selector	Routine
$0001	`SCSIAction`
$0002	`SCSIRegisterBus`
$0003	`SCSIDeregisterBus`
$0004	`SCSIReregisterBus`
$0005	`SCSIKillXPT`

Result Codes

`noErr`	0	No error
`scsiRequestInProgress`	1	Parameter block request is in progress
`scsiCDBLengthInvalid`	-7863	The CDB length supplied is not supported by this SIM; typically this means it was too big
`scsiTransferTypeInvalid`	-7864	The `scsiTransferType` is not supported by this SIM
`scsiDataTypeInvalid`	-7865	SIM does not support the requested `scsiDataType`
`scsiIDInvalid`	-7866	The initiator ID is invalid
`scsiLUNInvalid`	-7867	The logical unit number is invalid
`scsiTIDInvalid`	-7868	The target ID is invalid
`scsiBusInvalid`	-7869	The bus ID is invalid
`scsiRequestInvalid`	-7870	The parameter block request is invalid
`scsiFunctionNotAvailable`	-7871	The requested function is not supported by this SIM
`scsiPBLengthError`	-7872	The parameter block length is too small for this SIM
`scsiQLinkInvalid`	-7881	The `qLink` field was not 0
`scsiNoSuchXref`	-7882	No driver has been cross-referenced with this device
`scsiDeviceConflict`	-7883	Attempt to register more than one driver to a device
`scsiNoHBA`	-7884	No HBA detected
`scsiDeviceNotThere`	-7885	SCSI device not installed or available
`scsiProvideFail`	-7886	Unable to provide the requested service
`scsiBusy`	-7887	SCSI subsystem is busy
`scsiTooManyBuses`	-7888	SIM registration failed because the XPT registry is full
`scsiCDBReceived`	-7910	The SCSI CDB was received
`scsiNoNexus`	-7911	Nexus is not established
`scsiTerminated`	-7912	Parameter block request terminated by the host
`scsiBDRsent`	-7913	A SCSI bus device reset (BDR) message was sent to the target
`scsiWrongDirection`	-7915	Data phase was in an unexpected direction
`scsiSequenceFail`	-7916	Target bus phase sequence failure
`scsiUnexpectedBusFree`	-7917	Unexpected bus free phase
`scsiDataRunError`	-7918	Data overrun/underrun error
`scsiAutosenseFailed`	-7920	Automatic `REQUEST SENSE` command failed
`scsiParityError`	-7921	An uncorrectable parity error occurred
`scsiSCSIBusReset`	-7922	Execution of this parameter block was halted because of a SCSI bus reset
`scsiMessageRejectReceived`	-7923	`REJECT` message received
`scsiIdentifyMessageRejected`	-7924	The target issued a `REJECT` message in response to the `IDENTIFY` message; the LUN probably does not exist
`scsiCommandTimeout`	-7925	The timeout value for this parameter block was exceeded and the parameter block was aborted
`scsiSelectTimeout`	-7926	Target selection timeout
`scsiUnableToTerminate`	-7927	Unable to terminate I/O parameter block request
`scsiNonZeroStatus`	-7932	The target returned non-zero status upon completion of the request
`scsiUnableToAbort`	-7933	Unable to abort parameter block request
`scsiRequestAborted`	-7934	Parameter block request aborted by the host