INSIDE MACINTOSH

SCSI Family Reference

WWDC Release

May 1996

Apple Computer, Inc.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

SCSI Family Reference

Contents

You need to read this reference if you write device drivers for SCSI devices, if you write other software that uses SCSI services, or if you are writing a SCSI interface module (or plug-in) for a specific type of SCSI controller chip.

This chapter assumes that you have an understanding of the Mac OS 8 I/O architecture, as described in "About the I/O Architecture" **to be provided**.

If you are writing a device driver for a block-structured storage device such as a hard disk, you need to read "Block Storage Family Reference".

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*
- *SCSI-2 Common access method transport and SCSI interface module*

Draft Release Note
The information in this chapter is preliminary and subject to change. ◆

About the SCSI Family

The **SCSI family** is a completely new implementation of SCSI services for Mac OS 8. It provides

- improved performance over SCSI Manager 4.3
- a new connection-based interface for clients that doesn't require knowledge of complex parameter block structures
- enhancements to the plug-in interface, such as a simplified process for acquiring plug-in entry points
- **access control**, which allows a client to open a device with a specific status (either shared or reserved)

The SCSI family is included in Mac OS 8 and can run on any Power Macintosh or Mac-compatible computer. The SCSI family provides a client programming interface and a plug-in programming interface, and is responsible for

- routing requests to the proper plug-in

- notifying the caller when a request is complete
- maintaining compatibility with the SCSI Manager 4.3 interface
- isolating plug-ins from comprehensive knowledge of (and access to) other operating system components

Software written to the SCSI Manager 4.3 interface runs on Mac OS 8 through a compatibility layer, shown in Figure 1-1. Most SCSI Manager 4.3 functions and data structures are maintained, but no support is provided for the original SCSI Manager client API and its emulation in SCSI Manager 4.3 (as described in *Inside Macintosh: Devices*).

▲ **WARNING**
If your application or device driver calls an original SCSI Manager function or a SCSI Manager 4.3 function that provides emulation, it will get a "not supported" error. ▲

The SCSI family client interface defines communication between the SCSI family and its clients, which may include applications, other I/O families, and plug-ins from other families. For example, the block storage family and its plug-ins are the primary SCSI family clients.

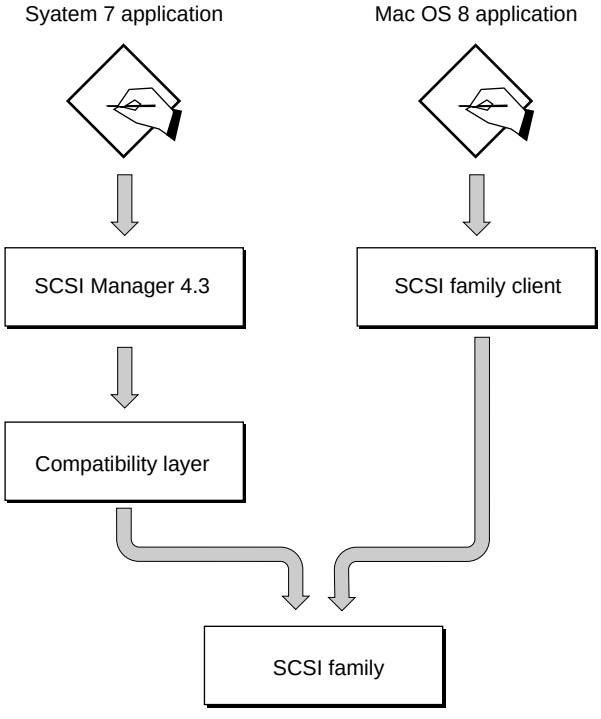
A client uses the services of the SCSI family and its plug-ins to manage a SCSI device and to transfer data to and from it. The family provides a client interface and passes on requests to the appropriate plug-in. A SCSI plug-in (also known as a **SCSI interface module**, or SIM) is responsible for managing the host bus adaptor (HBA) for a bus. "SCSI Client Constants and Data Types," beginning on page 1-11 describes the data types and constants available to SCSI family clients. "SCSI Client Functions," beginning on page 1-30, describes the available functions.

Note
This document generally refers to a SCSI interface module as a *plug-in*, rather than a *SIM*. ◆

When a client calls a SCSI family function such as `SCSIExecIOSyncCmd` (page 1-33) or `SCSIExecIOAsyncCmd` (page 1-35), the SCSI family server uses information passed in the function parameters to build a parameter block structure for use by the appropriate SCSI family plug-in. A SCSI family client is shielded from the complexity of the parameter block, and does not access its fields directly.

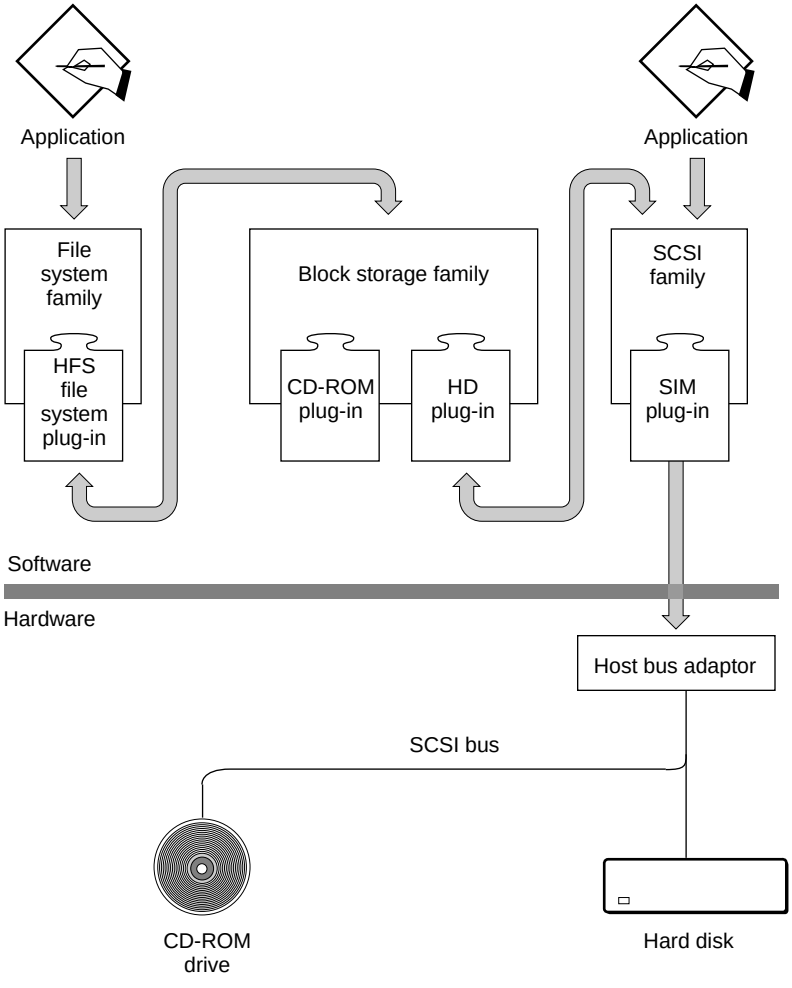
The parameter block structures used in the plug-in programming interface are nearly identical to those supported by SCSI Manager 4.3, although a few fields have been changed or are no longer supported. (For information on specific fields, see the reference sections for individual parameter block structures.) The SCSI family and plug-ins ignore any parameter block fields that are no longer used.

Figure 1-1 SCSI Manager 4.3 compatibility



Macros such as `SCSIPBHdr` and `SCSI_IO` have been replaced by similarly named structures—for more information, see "SCSI Parameter Block Header" (page 1-80) and "SCSI I/O Parameter Block" (page 1-82). The structures define fields identical to those in the original macros.

Figure 1-2 The SCSI family and a SIM plug-in in the I/O architecture



The SCSI family plug-in interface defines communication between the SCSI family and its plug-ins, which may include Apple and third-party plug-ins. "Plug-in Constants and Data Types," beginning on page 1-61, describes the data types and constants available to SCSI family plug-ins. "Exported by the SCSI Family," beginning on page 1-96, describes family functions available to

plug-ins. “SCSI Plug-in-Defined Functions,” beginning on page 1-98, describes functions a plug-in must make available to the SCSI family.

Along with the SCSI family, Apple provides plug-ins to manage hardware in Macintosh computers. Apple or third-party developers can add other plug-ins and HBA hardware at any time. For example, a PCI or NuBus expansion card can provide an additional SCSI bus that device drivers can access through the SCSI family in exactly the same way they access the internal bus. Figure 1-2 shows the relationship between applications, device drivers, the SCSI family, a SCSI plug-in (or SIM), and the SCSI controller hardware.

There are several advantages of using the new SCSI family client interface:

- It can reduce coding complexity and simplify maintenance requirements.
- The SCSI Manager 4.3 interface is not guaranteed to work with future versions of Macintosh system software.
- Using the SCSI Manager 4.3 interface may lead to reduced performance because the transitions necessary to maintain compatibility are complex and time-consuming.

SCSI Client Constants and Data Types

SCSI Connection Data Types

A SCSI *connection* is a logical path to a SCSI bus or a SCSI device. The connection controls access to its bus or device. Access to a device may be shared or reserved; access to a bus must be shared.

A SCSI *connection ID* is a value that uniquely identifies a connection. It is assigned by the Mac OS 8 when a new connection is opened. The ID remains valid from the time you open the connection until the time you close it.

For more information on connections, see “Connection-Based Services,” **to be provided**.

Note
At the present time, the SCSI family does not probe for logical units (LUNs). As a result, no LUNs exist as entries in the Name Registry, and you cannot open a direct connection to a LUN. Probing for logical units will be added in a later release. In the meantime, you can use the `SCSIExecIOControlSyncCmd` (page 1-37) or the `SCSIExecIOControlAsyncCmd` (page 1-40) functions to get access to a LUN. For more information, see the discussions associated with those functions.

ConnectionType

When you call the `SCSIOpenConnection` function (page 1-30) to open a connection, you specify a connection type. The SCSI family defines the `ConnectionType` data type and provides enumerated values for connection types.

```
typedef UInt32 ConnectionType;

enum {
    kReservedAccess     = 0x0100    /* reserved */
    kSharedAccess       = 0x0200,   /* shared */
};
```

Enumerator descriptions

`kReservedAccess` A connection that allows only one client to have access to a device. If granted, other requests to open a connection to the device will be denied. A bus connection may not be reserved.

`kSharedAccess` A connection that allows shared access to a device. If granted, other requests to open a connection to the device will be allowed. A bus connection must be shared.

ConnectionID

You obtain a connection ID by calling the `SCSIOpenConnection` function (page 1-30). You can pass the ID to SCSI client functions that read or write to a device or bus, use it to get information about a device or a bus and the plug-in associated with it, or pass it to the `SCSICloseConnection` function (page 1-32) to close the connection.

The SCSI family defines the `ConnectionID` data type for a connection ID.

```
typedef ObjectID    ConnectionID;
```

When you open a connection to a SCSI bus, you can use the `SCSIExecIOControlSyncCmd` (page 1-37) or the `SCSIExecIOControlAsyncCmd` (page 1-40) function to get limited access to devices on that bus. This capability allows you, for example, to perform a bus probe that obtains certain information about each device on the bus without having to make a connection to each device.

The SCSI Execution Tag

SCSIExecIOTag

When you call the `SCSIExecIOAsyncCmd` function (page 1-35) to make a request of a SCSI device or the `SCSIExecIOControlAsyncCmd` function (page 1-40) to make a request of a SCSI bus, the function returns a value, called a tag, that uniquely identifies the I/O request. The SCSI family defines the `SCSIExecIOTag` data type for an I/O tag.

```
typedef MessageID SCSIExecIOTag;
```

You can abort or terminate an I/O request by passing its **SCSI execution tag** to the `SCSIAbortIOCmd` function (page 1-43) or the `SCSITerminateIOCmd` function (page 1-44).

The SCSI Data Structure

SCSIDataObject

When you want to transfer data to or from a SCSI device using the `SCSIExecIOSyncCmd` function (page 1-33), the `SCSIExecIOAsyncCmd` function (page 1-35), the `SCSIExecIOControlSyncCmd` function (page 1-37), or the `SCSIExecIOControlAsyncCmd` function (page 1-40), you provide a `SCSIDataObject` structure.

The SCSI family defines the `SCSIDataObject` data type and enumerated values to specify information for a data transfer.

```
struct SCSIDataObject {
    UInt8   *scsiDataPtr;
    SInt32  scsiDataLength;
    UInt16  scsiDataType;
    UInt16  scsiSGListCount;
};
```

Field descriptions

`scsiDataPtr` A pointer to a data buffer, a **scatter/gather list**, or an I/O table that you provide to supply data to the function or receive data from it. You use the `scsiDataType` field to specify the type the pointer points to.

`scsiDataLength` The amount of data you want to transfer, in bytes.

`scsiDataType` The data type pointed to by the `scsiDataPtr` field. You specify the type using one of the constants described in "SCSI Data Type," beginning on page 1-14.

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

SCSI Data Type

You specify the data type pointed to by the `scsiDataPtr` field of the `SCSIDataObject` structure by setting the `scsiDataType` field to one of the following constants.

```
enum {
    scsiDataBuffer          = 0,    /* single contiguous buffer */
    scsiDataTIB             = 1,    /* not supported  */
    scsiDataSG              = 2,    /* scatter/gather list */
    scsiDataIOTable         = 3,    /* I/O table prepared by block
                                        storage */
    scsiDataMemList         = 4     /* a special memory list */
};
```

The constant you select provides information about the source or destination location for the data to be transferred. It also provides information about the state of the data. For example, any virtual memory used for data must be locked down to physical addresses so that no page fault occurs during data transfer, which could lead to a deadlock condition. Locking down of memory can only be performed by a task running in supervisor mode.

Enumerator descriptions

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

`scsiDataTIB` Not supported (obsolete).

`scsiDataSG` The `scsiDataPtr` field contains a pointer to a **scatter/gather list** (page 1-79). Each entry in a scatter/gather list contains the address and size of one buffer. The `scsiDataLength` field specifies the total number of bytes to be transferred (the sum of all the buffer sizes). The buffers specified by the scatter/gather list are likely to be in virtual memory, so the SCSI family must lock down memory to physical addresses before a transfer can take place. Locking down memory prevents page faults from occurring during a transfer.

`scsiDataIOTable` The `scsiDataPtr` field contains a pointer to an I/O table prepared by a client running in supervisor mode (such as the block storage family). If any virtual memory was used for the data, that memory has been locked down to physical addresses. The `scsiDataLength` field specifies the total number of bytes to be transferred.

`scsiDataMemList` The `scsiDataPtr` field contains a pointer to an I/O table prepared by a client running in either supervisor mode or user mode. Each entry in the I/O table describes a range in

memory. The memory doesn't have to be locked down—a client running in user mode can prepare the list and pass it to the SCSI family to lock it down, but a client running in supervisor mode must lock down the memory before calling on the family. The `scsiDataLength` field contains the total number of bytes to be transferred. For more information on using a **memory list**, see **to be supplied**.

The SCSI family defines an enumerated type a plug-in can use to set the bits in the `scsiDataType` field of the `SCSIDataObject` structure. Bits 0 to 15 are defined by Apple. Bits 16 to 30 are available for third parties. Bit 31 is reserved. The addressing is **little-endian**—that is, the value is read from right to left, with bit 0 on the right and bit 31 on the left.

```
enum
{
    scsiBusDataTIB      = (1<<scsiDataTIB),     /* not supported (obsolete) */
    scsiBusDataBuffer   = (1<<scsiDataBuffer),  /* single contiguous buffer supplied */
    scsiBusDataSG       = (1<<scsiDataSG),      /* scatter/gather list supplied */
    scsiBusDataIOTable  = (1<<scsiDataIOTable), /* prepare Memory for IO */
    scsiBusDataMemList  = (1<<scsiDataMemList), /* memory list */

    scsiBusDataReserved = 0x80000000            /* reserved  */
};
```

Enumerator descriptions

`scsiBusDataTIB` Not supported.

`scsiBusDataBuffer` Set bit indicating single contiguous buffer.

`scsiBusDataSG` Set bit indicating scatter/gather list.

`scsiBusDataIOTable` Set bit indicating memory has been prepared (and locked down) by the block storage family or other supervisor mode process.

`scsiBusDataMemList`
Set bit indicating an I/O table has been prepared (but memory has not necessarily been locked down).

`scsiBusDataReserved`
Reserved.

The SCSI Command Descriptor Block Structure

SCSICDBObject

When you call the `SCSIExecIOSyncCmd` function (page 1-33), the `SCSIExecIOAsyncCmd` function (page 1-35), the `SCSIExecIOControlSyncCmd` function (page 1-37), or the `SCSIExecIOControlAsyncCmd` function (page 1-40), you provide a `SCSICDBObject` structure.

The SCSI family defines the `SCSICDBObject` data type to specify a command descriptor block (CDB).

```
struct SCSICDBObject {
    UInt16  scsiCDBLength;
    CDB     scsiCDB;
};
```

Field descriptions

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

`scsiCDB` An actual CDB. For information on the format and length of a CDB, see "CDB," beginning on page 1-78 and "Data Length Constants," beginning on page 1-77.

The SCSI Flags Structure

SCSIFlagsObject

When you call the `SCSIExecIOSyncCmd` function (page 1-33), the `SCSIExecIOAsyncCmd` function (page 1-35), the `SCSIExecIOControlSyncCmd` function (page 1-37), or the `SCSIExecIOControlAsyncCmd` function (page 1-40), you pass a `SCSIFlagsObject` structure.

The SCSIFlagsObject structure contains flag fields that help specify an I/O request.

```
struct SCSIFlagsObject {
    UInt32  scsiFlags;
    UInt16  scsiIOFlags;
    UInt16  scsiTransferType;
};
```

Field descriptions

scsiFlags	Flags that you set to indicate the transfer direction and any special handling required for this request. See "SCSI Flags," beginning on page 1-66, for flag descriptions.
scsiIOFlags	Additional I/O flags you use to describe the data transfer. See "SCSI I/O Flags," beginning on page 1-70, for flag descriptions.
scsiTransferType	The type of transfer—blind or polled—to use during the data phase. You specify the type using one of the constants described in "Transfer Types," beginning on page 1-70.

Autosense Size Value

Autosense is feature of SCSI Manager 4.3 that automatically sends a REQUEST SENSE command in response to a CHECK CONDITION status, and retrieves the sense data into the **autosense buffer**.The SCSI family provides an enumerated value, kMaxAutoSenseByteCount, for the maximum size of the buffer.

```
enum  {
kMaxAutoSenseByteCount = 255        /* max byte size of sense buffer */
};
```

The autosense size value is used in the SCSIExecIOResult structure (page 1-19)

The SCSI I/O Result Structure

SCSIExecIOResult

When you call the `SCSIExecIOSyncCmd` (page 1-33), `SCSIExecIOAsyncCmd` (page 1-35), `SCSIExecIOControlSyncCmd` (page 1-37), or `SCSIExecIOControlAsyncCmd` (page 1-40) functions, you provide a pointer to a `SCSIExecIOResult` structure to return the result of the I/O operation.

The SCSI family defines the `SCSIExecIOResult` data type to return result information from an I/O operation. The information is filled in by the appropriate plug-in.

```
struct SCSIExecIOResult {
    OSStatus    scsiResult;
    UInt16      scsiResultFlags;
    UInt16      scsiSenseLength;    /* actual sense length returned */
    SInt32      scsiDataResidual;   /* residual data length */
    SCSIExecIOTag   ioTag;
    UInt8       scsiSense[kMaxAutoSenseByteCount];
                                    /* autosense data buffer */
    UInt8       scsiSCSIstatus;
};
```

Field descriptions

`scsiResult` The result code returned by the `SCSIExecIOSyncCmd`, `SCSIExecIOAsyncCmd`, `SCSIExecIOControlSyncCmd`, or `SCSIExecIOControlAsyncCmd` function. See "SCSI Family Result Codes," beginning on page 1-101, for a list of all result codes specific to the SCSI family.

`scsiResultFlags` Flags set by the plug-in when certain conditions apply; otherwise, the plug-in sets this field to 0. The flags modify the value in the `scsiResult` field. Enumerated values for setting or testing this field are described in "Result Flags," beginning on page 1-20.

`scsiSenseLength` The number of bytes of data the plug-in placed in your autosense data buffer.

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. The plug-in sets this field.
SCSIExecIOTag	A token that uniquely identifies the I/O request this structure pertains to (page 1-13). The plug-in sets this field.
scsiSense	Your autosense data buffer. If autosense is enabled, the plug-in returns REQUEST SENSE information in this buffer. Autosense is enabled when you do not set the scsiDisableAutosense flag in the scsiFlags field of the SCSIFlagsObject structure (page 1-17).
scsiSCSIstatus	The status returned by the SCSI device. See "Data Length Constants," beginning on page 1-77, for a list of values that a SCSI device can return.

Result Flags

The SCSI family provides enumerated values for the scsiResultFlags field of the SCSIExecIOResult structure. The constant stored in the scsiResultFlags field modifies the value in the scsiResult field.

```
enum {
    scsiSIMQFrozen      = 0x0001,   /* Plug-in queue is frozen with this err*/
    scsiAutosenseValid  = 0x0002,   /* autosense data valid for target  */
    scsiBusNotFree      = 0x0004    /* at time of callback, SCSI bus is not free */
};
```

Enumerator descriptions

scsiSIMQFrozen	The plug-in queue for this logical unit (LUN) is frozen because of an error. You must call the SCSIReleaseQCmd function (page 1-46) to release the queue and resume processing requests.
scsiAutosenseValid	The plug-in performed an automatic REQUEST SENSE 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 plug-in was unable to clear the bus after an error. You may need to call the SCSIBusResetSync function (page 1-48)

or the SCSIBusResetAsync function (page 1-50) to restore operation. The choice of using a synchronous or an asynchronous call is up to the client.

The SCSI Handshake Structure

SCSIHandshakeObject

When you call the SCSISetHandshake function (page 1-54), you provide a SCSIHandshakeObject structure to specify handshaking instructions for blind transfers between a plug-in and a device. You store the handshaking instructions in an array of UInt16 (2-byte) values and terminate the array with 0.

The plug-in polls for data ready after transferring the amount of data specified in each successive scsiHandshake entry. When it encounters a 0 value, the plug-in 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.

Note
You currently set up handshaking instructions for a device globally by calling the SCSISetHandshake function. In later releases, you will be able to set up handshaking instructions for each I/O request. ◆

The SCSI family defines the SCSIHandshakeObject data type to store hand-shaking data.

```
struct SCSIHandshakeObject {
    UInt16  scsiHandshake [handshakeDataLength];
};
```

Field descriptions

scsiHandshake — The handshaking data.You store the handshaking instructions in an array of UInt16 (2-byte) values and terminate the array with 0. The constant handshakeDataLength is described in "Data Length Constants," beginning on page 1-77.

The SCSI I/O Options Structure

SCSIIOOptionsObject

The `SCSIIOOptionsObject` structure specifies options for an I/O operation.

Note
When you call the `SCSISetIOOptions` function (page 1-57), you provide a `SCSIIOOptionsObject` structure. The `SCSISetIOOptions` function is currently used to set option flags globally before calling SCSI family client functions such as `SCSIExecIOSyncCmd` (page 1-33). In a future software release, clients will be able to specify I/O options as part of the function interface for SCSI family functions. Both the `SCSISetIOOptions` function and the `SCSIIOOptionsObject` data structure will be eliminated.

```
struct SCSIIOOptionsObject {
    UInt32  scsiFlags;
    UInt32  scsiIOFlags;
};
```

Field descriptions

`scsiFlags` Flags that you set to indicate the transfer direction and any special handling required for this request. See "SCSI Flags," beginning on page 1-66, for flag descriptions.

`scsiIOFlags` Additional I/O flags you use to describe the data transfer. See "SCSI I/O Flags," beginning on page 1-70, for flag descriptions.

The SCSI Bus Information Structure

SCSIBusInfo

When you call the `SCSIBusInquiryCmd` function (page 1-60), you provide a pointer to a `SCSIBusInfo` structure. The function fills in the fields of the structure. The fields of the `SCSIBusInfo` structure are very similar to the fields of the `SCSIBusInquiryPB` structure (page 1-87)

```
struct  SCSIBusInfo
{
    UInt16  scsiEngineCount;        /* <- Number of engines on HBA */
    UInt16  scsiMaxTransferType;    /* <- Number of transfer types for this HBA */

    UInt32  scsiDataTypes;          /* <- which data types this plug-in supports */

    UInt32  scsiBIReserved4;        /* Reserved. */

    UInt32  scsiFeatureFlags;       /* <- Supported features flags field */

    UInt8   scsiVersionNumber;      /* <- Version number for the plug-in/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;     /* <- Ptr to plug-in private data area */
    UInt32  scsiSIMPrivatesSize;    /* <- Size of plug-in private data area */
    UInt32  scsiAsyncFlags;         /* <- Event cap. for Async Callback */

    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;
    UInt16  scsiMaxTarget;          /* <- maximum Target number supported */
```

```
    UInt16  scsiMaxLUN;             /* <- maximum Logical Unit number supported */

    char    scsiSIMVendor[ vendorIDLength ];
                                    /* <- Vendor ID of plug-in (or XPT if bus<FF) */
    char    scsiHBAVendor[ vendorIDLength ];
                                    /* <- Vendor ID of the HBA */
    char    scsiControllerFamily[ vendorIDLength ];
                                    /* <- Family of SCSI Controller */
    char    scsiControllerType[ vendorIDLength ];
                                    /* <- Specific Model of SCSI Controller used */
    char    scsiXPTversion[4];      /* <- version number of XPT */
    char    scsiSIMversion[4];      /* <- version number of plug-in */
    char    scsiHBAversion[4];      /* <- version number of HBA */

    UInt8   scsiHBAslotType;        /* <- type of "slot" that this HBA is in */
    UInt8   scsiHBAslotNumber;      /* <- slot number of this HBA */
    UInt16  scsiSIMsRsrcID;         /* <- resource ID of this plug-in */

    UInt16  scsiBIReserved3;        /* Reserved. */
};
```

Field descriptions

`scsiEngineCount` The number of engines on the HBA. This value is 0 for a built-in SCSI bus. See the *SCSI-2 Common access method transport and SCSI interface module* specification for information about HBA engines.

`scsiMaxTransferType` The number of transfer types supported by the plug-in. A plug-in supports all transfer types that are specified by a constant value equal to or less than the value it returns here. For example, if a plug-in returns the value `scsiTransferPolled` for its transfer type, the plug-in supports both the blind and polled transfer types. See "Transfer Types," beginning on page 1-70, for a description of the defined types.

`scsiDataTypes` A bit mask specifying the data types supported by the plug-in/HBA. See "SCSI Data Type," beginning on page 1-14, for more information.

`scsiBIReserved4` Reserved.

`scsiFeatureFlags` Flags that describe various physical characteristics of the SCSI bus. See "Feature Flags," beginning on page 1-72, for flag definitions.

`scsiVersionNumber` The version number of the plug-in/HBA.

`scsiHBAInquiry` Flags describing the capabilities of the bus. See "More Feature Flags," beginning on page 1-73, for flag definitions.

`scsiTargetModeFlags` Reserved.

`scsiScanFlags` Information about the scanning-related features supported by the plug-in/HBA. You can test for specific features using the bit masks described in "Scan Types," beginning on page 1-76.

`scsiSIMPrivatesPtr` A pointer to the plug-in's private storage.

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

`scsiAsyncFlags` Reserved.

`scsiHiBusID` The highest bus number currently registered in the Name Registry. The SCSI family provides this value. If no buses are registered, it sets this field to 0xFF.

`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 the plug-in supports. See "SCSI Flags," beginning on page 1-66, for flag definitions.

`scsiIOFlagsSupported` A bit mask that defines which `scsiIOFlags` bits the plug-in supports. See "SCSI I/O Flags," beginning on page 1-70, for flag definitions.

`scsiWeirdStuff` Flags that identify unusual aspects of a plug-in's operation. See "Unusual Features Flags," beginning on page 1-74, for flag definitions.

`scsiMaxTarget` The highest SCSI bus ID supported by the HBA. For a standard SCSI-2 HBA, the value is 7; for an HBA that supports wide transfer, the value is 15.

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

`scsiSIMVendor` A null-terminated ASCII text string that identifies the plug-in vendor. On Macintosh computers, for example, the function returns `'Apple Computer \0'` for a built-in SCSI bus.

`scsiHBAVendor` A null-terminated ASCII text string that identifies the HBA vendor. On Macintosh computers, for example, the function returns `'Apple Computer \0'` for a built-in SCSI bus.

`scsiControllerFamily`
An optional null-terminated 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`
An optional null-terminated ASCII text string that identifies the specific type of SCSI controller chip. This information is provided at the discretion of the HBA vendor.

`scsiXPTversion` Not used (obsolete).

`scsiSIMversion` An ASCII text string that identifies the version number of the plug-in. You should use the other fields of this structure to check for specific features, rather than relying on this value.

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

`scsiHBAslotType` The slot type, if any, used by this HBA. Slot types are defined in "Slot Types," beginning on page 1-76.

`scsiHBAslotNumber` Reserved.

`scsiSIMsRsrcID` Reserved.

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

Device Identification Structure

DeviceIdent

The device identification structure specifies a target device by its bus number, SCSI ID, and logical unit number (LUN). The device identification structure is defined by the `DeviceIdent` data type. You can use the `SCSIBusGetDeviceData` function (page 1-58) to get an array containing a `DeviceIdent` data structure for each device on a bus. You can then use information from the array to open a connection to any of the devices.

For information on how to get limited information about the devices on a bus, without the overhead of opening a connection to each device, see the `SCSIExecIOSyncCmd` function (page 1-33).

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

Field descriptions

`diReserved`	Reserved.
`bus`	The bus number of the plug-in/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.

The SCSI Device Iterator Structure

SCSIIOIteratorData

The SCSI family defines the `SCSIIOIteratorData` data structure to describe a device or LUN on a SCSI bus. For example, when you call the `SCSIBusGetDeviceData` function (page 1-58), you provide a pointer to an array of one or more `SCSIIOIteratorData` structures. You allocate memory for the array and the function fills in a separate element in the array with information for each device or LUN.

```
struct SCSIIOIteratorData {
    IOCommonInfo    deviceInfo;
    DeviceIdent     deviceID;
    DeviceType      deviceType;
};
```

Field descriptions

`deviceInfo` The iterator information common to all families. The `IOCommonInfo` structure (**to be provided**) specifies a device number that uniquely identifies a device within a family, and a version number that identifies the version of the plug-in's iterator structure that is in use.

`deviceID` The device's identification data. The `DeviceIdent` structure (page 1-27) specifies a bus number, a target SCSI ID, and a LUN number for a device. The default LUN number is 0, indicating the device is treated as though it has one logical unit.

`deviceType` The type of SCSI device. For a CD-ROM drive, for example, the `deviceType` field is set to the value `'CD-ROM'`. A SCSI device type is defined as a character array (page 1-29). The value stored in the SCSI device type field is returned from the device itself.

SCSI Device Type

SCSIDeviceType

The SCSI family defines the `DeviceType` data type to store a character value that identifies a SCSI device.

```
typedef char    DeviceType[kSCSIDeviceTypeSize];
```

For a CD-ROM drive, for example, the device type is specified by the value `'CD-ROM'`. When you call the `SCSIBusGetDeviceData` function (page 1-58), it returns the character device type in the `deviceType` field of the `SCSIIOIteratorData` data structure.

The SCSI family defines an enumerated value, `kSCSIDeviceTypeSize`, to set the length of the `DeviceType` character array. It also defines a value, `kSCSIAllBus`, you can pass in the `bus` parameter to the `SCSIBusGetDeviceData` function to specify that data should be returned for all devices on all available buses.

```
enum{
    kSCSIDeviceTypeSize = 9,    // number of characters for device type
    kSCSIAllBus = 0xFF          // search all buses
};
```

The SCSI-2 reference manual defines hex values for a number of SCSI device types. When you call `SCSIBusGetDeviceData`, the SCSI family obtains the hex value from the device and uses it to determine the character device type, based on the values shown in the following table:

SCSI-2 hex device type	SCSI family character device type
0x00	`DASD`
0x01	`SASD`
0x02	`PRINTER`
0x03	`PROCESSOR`
0x04	`WORM`
0x05	`CD-ROM`

SCSI-2 hex device type	SCSI family character device type
0x06	SCANNER
0x07	OPTICAL
0x08	JUKEBOX
0x09	COMM
0x0A-0x0B	ASC-IT8
0x0C-0x1E	RESERVED
0x1F	UNKNOWN

A device that wishes to specify a vendor-specific device type should return the value 0x1F.

Note
Enhancements to the SCSI family's device type handling will be provided in a future software release. ◆

SCSI Client Functions

Opening and Closing a SCSI Connection

SCSIOpenConnection

Opens a connection to a SCSI device or a SCSI bus.

```
OSStatus SCSIOpenConnection (
                    RegEntryRef * regID,
                    ConnectionType type,
                    ConnectionID * connID);
```

`regID`	On input, a pointer to the Name Registry reference for the target device or bus you want to open. When `regID` refers to a device node in the Name Registry, the connection is opened as a device connection. When `regID` refers to a bus node, it is opened as a bus connection.
`type`	The type of connection you are requesting, either shared or reserved. For a bus connection, the connection type can only be shared. "SCSI Connection Data Types," beginning on page 1-11, lists the defined connection constants.
`connID`	A pointer to a connection ID (page 1-12). On output, the function provides a new connection ID.
function result	A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

You call `SCSIOpenConnection` to open a connection and obtain a connection ID for a bus or device. You use the connection to obtain information or to make I/O requests. You can pass the connection ID as a parameter to most SCSI client functions.

The `SCSIOpenConnection` function compares the passed `RegEntryRef` to the registry references it knows about to make sure the device is actually in the family's device tree.

You can obtain a registry reference by calling the `SCSIBusGetDeviceData` function. The reference is returned as a field of the `IOCommonInfo` parameter.

IMPORTANT

When you open a device connection, you must call `SCSIExecIOSyncCmd` (page 1-33) or `SCSIExecIOAsyncCmd` (page 1-35) to send a request for that device to the SCSI family. When you open a bus connection, you must call `SCSIExecIOControlSyncCmd` (page 1-37) or `SCSIExecIOControlAsyncCmd` (page 1-40) to make a request for that bus. ▲

When you are finished using a connection, you call the `SCSICloseConnection` function (page 1-32) to close the connection and release associated system resources.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SCSICloseConnection

Closes a connection to the SCSI family.

```
OSStatus SCSICloseConnection (ConnectionID connID);
```

`connID` The connection ID to a SCSI bus or SCSI device connection you want to close.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

You should always call `SCSICloseConnection` to close a connection when you are done using it. Although a connection requires little overhead, leaving a reserved device connection open effectively ties up the entire device, preventing other clients from accessing it.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Performing I/O Operations

Before calling any SCSI family function to perform an I/O operation, a SCSI family client can call `SCSISetIOOptions` (page 1-57) to set any required option flags. When you call the `SCSISetIOOptions` function, you provide a `SCSIIOOptionsObject` structure. In a future software release, clients will be able to specify I/O options as part of the function interface for SCSI family functions. Both the `SCSISetIOOptions` function and the `SCSIIOOptionsObject` data structure will be eliminated.

SCSIExecIOSyncCmd

Initiates a synchronous I/O request to a SCSI device.

```
OSStatus SCSIExecIOSyncCmd (
                    ConnectionID connID,
                    SCSIDataObject dataObject,
                    SCSICDBObject cdbObject,
                    SCSIFlagsObject flagsObject,
                    SCSIExecIOResult *resultBuffer,
                    SCSIExecIOTag *ioTag);
```

`connID` The connection ID to a SCSI device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30). The connection cannot be to a SCSI bus—to communicate with a bus you use the `SCSIExecIOControlSyncCmd` (page 1-33).

`dataObject`	A `SCSIDataObject` structure (page 1-14) that you provide. It specifies the type and length of the data you want to transfer, and the location in memory to read from or write to.
`cdbObject`	A `SCSICDBObject` structure (page 1-17) that you provide. It specifies the length and memory location of your SCSI command descriptor block.
`flagsObject`	A `SCSIFlagsObject` structure (page 1-17) that you provide. It specifies a variety of information about the I/O request.
`resultBuffer`	A pointer to a `SCSIExecIOResult` structure (page 1-19). On output, the structure contains information about the I/O operation.
`ioTag`	A pointer to an I/O tag (page 1-13). For an asynchronous function such as `SCSIExecIOAsyncCmd` (page 1-35), the tag can be used to abort or terminate the I/O operation. The tag is not useful to `SCSIExecIOSyncCmd`, because by the time the synchronous routine returns, the request has completed.
function result	A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

When you call the `SCSIExecIOSyncCmd` function, the SCSI family forwards the request to the appropriate plug-in, which is determined from the connection ID. The plug-in performs all the actions necessary to fulfill the request, including arbitrating for the bus, selecting the device, sending the command descriptor block, retrieving or sending data, performing disconnect operations, and so on. The parameters you provide must contain all the information required by the plug-in to complete the SCSI request, including issuing a `REQUEST SENSE` command if necessary.

A client that calls `SCSIExecIOSyncCmd` is blocked until the I/O operation has completed. However, only the requesting client is blocked, and other SCSI family clients can continue to make I/O requests.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers. It can only be called with a connection ID for a SCSI device, not with a connection ID for a SCSI bus.

SCSIExecIOAsyncCmd

Initiates an asynchronous I/O request to a SCSI device.

```
OSStatus SCSIExecIOAsyncCmd (
                    ConnectionID connID,
                    KernelNotification *kernelNot,
                    SCSIDataObject dataObject,
                    SCSICDBObject cdbObject,
                    SCSIFlagsObject flagsObject,
                    SCSIExecIOResult *resultBuffer,
                    SCSIExecIOTag *ioTag);
```

`connID` The connection ID to a SCSI device that you provide. You get a connection ID by calling the `SCSIOpenConnection` function (page 1-30). The connection cannot be to a SCSI bus.

`kernelNot` A pointer to a microkernel notification structure that specifies how you want to be notified when this request completes. For more information on the `KernelNotification` structure, see "Microkernel Notification Services" in *Inside Macintosh: Microkernel and Core System Services* (**to be provided** in a later release of Mac OS 8 documentation).

`dataObject` A `SCSIDataObject` structure (page 1-14) that you provide. It specifies the type and length of the data you want to transfer, and the location in memory to read from or write to.

`cdbObject`	A `SCSICDBObject` structure (page 1-16) that you provide. It specifies the length and memory location of your SCSI command descriptor block.
`flagsObject`	A `SCSIFlagsObject` structure (page 1-17) that you provide. It specifies a variety of information about the I/O request.
`resultBuffer`	A pointer to a `SCSIExecIOResult` structure (page 1-19). On an error return, such as when the I/O request cannot be successfully queued, the `SCSIExecIOAsyncCmd` function sets the fields of this structure immediately. Otherwise, the fields are set before you are notified of the I/O completion.
`ioTag`	A pointer to an I/O tag (page 1-13). The `SCSIExecIOAsyncCmd` function sets the tag to a value that uniquely identifies this I/O request. You can use the tag value to abort (page 1-43) or terminate (page 1-44) the I/O operation.
function result	A result code. The value `noErr` indicates the request was successfully queued. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

Note
The constant `noErr`, which is defined to have the value 0, is not of type `OSStatus`, but can be used for comparison with SCSI family function results. In a future release, a constant of type `OSStatus` will be supplied.

DISCUSSION

When you call the `SCSIExecIOAsyncCmd` function, the SCSI family forwards the request to the appropriate plug-in, which is determined from the connection ID. As with the `SCSIExecIOSyncCmd` function (page 1-33), the plug-in performs all the actions necessary to fulfill the request, including arbitrating for the bus, selecting the device, and so on. However, in this case the request is queued, rather than handled immediately—the client task waits only for the function to return, not for completion of the I/O request. The parameters you provide must contain all the information required by the plug-in to complete the SCSI request, including issuing a `REQUEST SENSE` command if necessary.

The `SCSIExecIOAsyncCmd` function returns the `noErr` result code to indicate that the request was queued successfully. If the function returns an error, the

request was not queued, the notification mechanism is not invoked, and the result of the SCSI transaction is returned in the `scsiResult` field of the result buffer parameter (pointed to by `resultBuffer`).

You use the microkernel notification structure parameter (`kernelNot`) to specify the mechanism to notify your task that an asynchronous I/O request has completed:

- A microkernel queue. A notification is placed in the queue when the request completes. This is the preferred mechanism.
- A software interrupt. It is delivered to a task when an I/O operation completes execution.
- An event group and flags to be set on completion of an I/O operation.

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 `SCSIFlagsObject` structure.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SCSIExecIOControlSyncCmd

Initiates a synchronous I/O request to a SCSI bus.

```
OSStatus SCSIExecIOControlSyncCmd (
                    ConnectionID connID,
                    SCSIDataObject dataObject,
```

```
                    DeviceIdent deviceID,
                    SCSICDBObject cdbObject,
                    SCSIFlagsObject flagsObject,
                    SCSIExecIOResult *resultBuffer,
                    SCSIExecIOTag *ioTag);
```

`connID` The connection ID to a SCSI bus. You get a connection ID from the `SCSIOpenConnection` function (page 1-30). The connection cannot be to a SCSI device.

`dataObject` A `SCSIDataObject` structure (page 1-14) that you provide. It specifies the type and length of the data you want to transfer, and the location in memory to read from or write to.

`deviceID` A device identification structure that specifies a bus number, a target SCSI ID, and a LUN number. The addition of this parameter is the only difference between the parameter lists of the `SCSIExecIOControlSyncCmd` function and the `SCSIExecIOSyncCmd` function. It is supplied as a temporary mechanism for specifying a LUN, until logical unit probing is added to the SCSI family. If you do not need to specify a LUN, you do not need to use this parameter. For more information, see the discussion for this function.

`cdbObject` A `SCSICDBObject` structure (page 1-16) that you provide. It specifies the length and memory location of your SCSI command descriptor block.

`flagsObject` A `SCSIFlagsObject` structure (page 1-17) that you provide. It specifies a variety of information about the I/O request.

`resultBuffer` A pointer to a `SCSIExecIOResult` structure (page 1-19). On output, the structure contains information about the I/O operation.

`ioTag` A pointer to an I/O tag (page 1-13). For an asynchronous function such as `SCSIExecIOControlAsyncCmd` (page 1-40), the tag can be used to abort or terminate the I/O operation. The tag is not useful to `SCSIExecIOControlSyncCmd`, because by the time the synchronous routine returns, the request has completed.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

When you call the `SCSIExecIOControlSyncCmd` function, the SCSI family forwards the request to the appropriate plug-in. The plug-in performs all the actions necessary to fulfill the request, including arbitrating for the bus, selecting the device, sending the command descriptor block, retrieving or sending data, performing disconnect operations, and so on. The parameters you provide must contain all the information required by the plug-in to complete the SCSI request, including issuing a `REQUEST SENSE` command if necessary.

A client that calls `SCSIExecIOControlSyncCmd` is blocked until the I/O operation has completed. However, only the requesting client is blocked—other SCSI family clients can continue to make I/O requests.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers. It can only be called with a connection ID for a SCSI bus, not with a connection ID for a SCSI device.

SPECIAL CONSIDERATIONS

At the present time, the SCSI family does not probe for logical units (LUNs). As a result, no LUNs exist as targets in the Name Registry, and therefore you cannot open a direct connection to a LUN by calling the `SCSIOpenConnection` function. However, the SCSI family does currently provide access to LUNs through the `SCSIExecIOControlSyncCmd` function and the `SCSIExecIOControlAsyncCmd` function (page 1-40). (This feature will go away in a later release when the SCSI family adds the capability to probe the bus for LUNs.)

Suppose, for example, that a client knows that a multiple-disk CD-ROM player has four LUNs, one for each available disk. The client can access the LUN by performing the following steps:

- Call the `SCSIOpenConnection` function (page 1-30) to open a connection to the bus on which the device resides.
- Call the `SCSIExecIOControlSyncCmd` function, setting the fields of the `deviceID` parameter as follows
  - set the `diReserved` field to 0
  - set the `bus` field to the number of the bus
  - set the `targetID` field to the ID of the CD-ROM player
  - set the `LUN` field to the logical unit number of the desired CD-ROM disk (0 to 3)

Although the `SCSIExecIOControlSyncCmd` function and the `SCSIExecIOControlAsyncCmd` function (page 1-40) currently provide access to LUNs, that is only a temporary expedient. The intended purpose of these functions is to help perform diagnostics and maintenance. For example, you can use the `SCSIExecIOControlSyncCmd` function to perform a bus probe to obtain certain information about each device on the bus without having to make a connection to each device.

SCSIExecIOControlAsyncCmd

Initiates an asynchronous I/O request to a SCSI bus.

```
OSStatus SCSIExecIOControlAsyncCmd (
                    ConnectionID connID,
                    KernelNotification *kernelNot,
                    SCSIDataObject dataObject,
                    DeviceIdent deviceID,
                    SCSICDBObject cdbObject,
                    SCSIFlagsObject flagsObject,
                    SCSIExecIOResult *resultBuffer,
                    SCSIExecIOTag *ioTag);
```

`connID` The connection ID to a SCSI bus. You get a connection ID from the `SCSIOpenConnection` function (page 1-30). The connection cannot be to a SCSI device.

`kernelNot`	A pointer to a microkernel notification structure that specifies how you want to be notified when this request completes. For more information on the `KernelNotification` structure, see "Microkernel Notification Services" in *Inside Macintosh: Microkernel and Core System Services* (**to be provided** in a later release of Mac OS 8 documentation).
`dataObject`	A `SCSIDataObject` structure (page 1-14) that you provide. It specifies the type and length of the data you want to transfer, and the location in memory to read from or write to.
`deviceID`	A device identification structure that specifies a bus number, a target SCSI ID, and a LUN number. This is the only parameter used by the `SCSIExecIOControlSyncCmd` function that isn't also used by the `SCSIExecIOSyncCmd` function. It is supplied as a temporary mechanism for specifying a LUN, until logical unit probing is added to the SCSI family. If you do not need to specify a LUN, you do not need to use this parameter. For more information, see the function `SCSIExecIOControlSyncCmd` (page 1-37).
`cdbObject`	A `SCSICDBObject` structure (page 1-16) that you provide. It specifies the length and memory location of your SCSI command descriptor block.
`flagsObject`	A `SCSIFlagsObject` structure (page 1-17) that you provide. It specifies a variety of information about the I/O request.
`resultBuffer`	A pointer to a `SCSIExecIOResult` structure (page 1-19). On an error return, such as when the I/O request cannot be successfully queued, the `SCSIExecIOAsyncCmd` function sets the fields of this structure immediately. Otherwise, the fields are set before you are notified of the I/O completion.
`ioTag`	A pointer to an I/O tag (page 1-13). The `SCSIExecIOControlAsyncCmd` function sets the tag to a value that uniquely identifies this I/O request. A client can use the tag value to abort (page 1-43) or terminate (page 1-44) the I/O operation.
function result	A result code. The value `noErr` indicates the request was successfully queued. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

When you call the `SCSIExecIOControlAsyncCmd` function, the SCSI family forwards the request to the appropriate plug-in, which is determined from the connection ID. As with the `SCSIExecIOControlSyncCmd` function (page 1-37), the plug-in performs all the actions necessary to fulfill the request. However, in this case the request is queued, rather than handled immediately—the client task waits only for the function to return, not for completion of the I/O request. As with `SCSIExecIOControlSyncCmd`, the parameters you provide must contain all the information required by the plug-in to complete the SCSI request, including issuing a `REQUEST SENSE` command if necessary.

The `SCSIExecIOControlAsyncCmd` function returns the `noErr` result code to indicate that the request was queued successfully. If the function returns an error, the request was not queued, the notification mechanism is not invoked, and the result of the SCSI transaction is returned in the `scsiResult` field of the result buffer parameter (pointed to by `resultBuffer`).

You use the microkernel notification structure parameter (`kernelNot`) to specify the mechanism to notify your task that an asynchronous I/O request has completed:

- A microkernel queue. A notification is placed in the queue when the request completes. This is the preferred mechanism.
- A software interrupt. It is delivered to a task when an I/O operation completes execution.
- An event group and flags to be set on completion of an I/O operation.

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 `SCSIFlagsObject` structure.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers. It can only be called with a connection ID for a SCSI bus, not with a connection ID for a SCSI device.

Performing I/O Control Operations

SCSIAbortIOCmd

Cancels an existing, asynchronous I/O request.

```
OSStatus SCSIAbortIOCmd (
                    ConnectionID connID,
                    SCSIExecIOTag ioTag);
```

`connID` The connection ID to a SCSI bus or device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`ioTag` The I/O tag (page 1-13) that identifies the I/O request to be cancelled. You get the I/O tag by calling the `SCSIExecIOAsyncCmd` function (page 1-35) or the `SCSIExecIOControlAsyncCmd` function (page 1-40).

function result A result code. If the I/O request specified by the `ioTag` field is successfully cancelled, the function returns the `noErr` result code and the I/O request receives the `scsiRequestAborted` result code. If the request has already completed, `SCSIAbortIOCmd` returns `scsiUnableToAbort`.

DISCUSSION

The `SCSIAbortIOCmd` function is a synchronous command that cancels the I/O request identified by the `ioTag` field. Calling the `SCSIAbortIOCmd` function may or may not succeed in cancelling the specified I/O request before it completes. If the request has not yet been delivered to the device, the plug-in removes the request from the queue of pending requests; the request's notification mechanism is invoked with a result code of `scsiRequestAborted`. If the request has already been started, the plug-in attempts to send an `ABORT` message to the device, either by asserting the /ATN signal or by reselecting the device. If the request has already completed, `SCSIAbortIOCmd` returns `scsiUnableToAbort`.

IMPORTANT

When the `SCSIAbortIOCmd` function interrupts a data transfer, it can cause data to be lost. Data that has already been transferred may be incomplete or invalid. ▲

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SCSITerminateIOCmd

Cancels an existing asynchronous I/O request.

```
OSStatus SCSITerminateIOCmd (
                    ConnectionID connID,
                    SCSIExecIOTag ioTag);
```

`connID` The connection ID to a SCSI bus or device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`ioTag`	The I/O tag (page 1-13) that identifies the I/O request to be cancelled. You get an I/O tag from the `SCSIExecIOAsyncCmd` function (page 1-35) or the `SCSIExecIOControlAsyncCmd` function (page 1-40).
function result	A result code. If the request specified by the `ioTag` field is successfully cancelled, the function returns the `noErr` result code and the I/O request receives the `scsiTerminated` result code. If the request has already completed, `SCSITerminateIOCmd` returns `scsiUnableToTerminate`.

DISCUSSION

The `SCSITerminateIOCmd` function is a synchronous command that cancels the I/O request identified by the `ioTag` field. If the request has not yet been delivered to the device, the plug-in removes it from the queue of pending requests; the request's notification mechanism is invoked with a result code of `scsiTerminated`. If the request has already been started, the plug-in attempts to send a `TERMINATE IO PROCESS` message to the device, either by asserting the /ATN signal or by reselecting the device. If the request has already completed, `SCSITerminateIOCmd` returns `scsiUnableToTerminate`.

The `SCSITerminateIOCmd` function differs from the `SCSIAbortIOCmd` function (page 1-43) 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. For an abort command, the current request is halted immediately, if possible.

Calling the `SCSITerminateIOCmd` function may or may not succeed in terminating the specified I/O request before it completes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SCSIReleaseQCmd

Releases a frozen queue for a device specified by a connection ID.

```
OSStatus SCSIReleaseQCmd (ConnectionID connID);
```

`connID` The connection ID to a SCSI device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30). The ID must be for a device, not a bus, because the device queue is independent from the bus.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

The `SCSIReleaseQCmd` function releases a frozen I/O queue for the device associated with the connection ID you provide. If an I/O request returns with the `scsiSIMQFrozen` flag set in the `scsiResultFlags` field of the `SCSIExecIOResult` structure (page 1-18), you must call the `SCSIReleaseQCmd` function to restore normal operation.

Note
In a future release, the setting of I/O flags will be incorporated into the `SCSIReleaseQCmd` function. ◆

Queue freezing provides the opportunity to insert error-handling requests at the beginning of the queue. When an I/O request returns an error, the plug-in freezes the I/O queue for the device that caused the error. You can then issue additional I/O requests with the `scsiSIMQHead` flag set so that the requests will be inserted in front of any other requests 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 in the queue, you call the `SCSIReleaseQCmd` 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 call to `SCSIReleaseQCmd`.

Subsequent errors continue to freeze the queue, allowing you to step through the queue one request at a time without aborting any other pending requests.

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

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `SCSIReleaseQCmd` function can not be called by hardware interrupt handlers or secondary interrupt handlers. It can only be called with a connection ID for a SCSI device, not with a connection ID for a SCSI bus.

SCSIClearQueue

Issues a `CLEAR QUEUE` command to the specified device.

```
OSStatus SCSIClearQueue (ConnectionID connID);
```

`connID`	The connection ID to a SCSI device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30). The ID must be for a device, not a bus, because the device queue is independent from the bus itself.
function result	A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

The `SCSIClearQueue` function directs the device to clear its I/O request queue. Any operations on the queue are terminated. Any pending I/O requests are terminated with the `scsiRequestAborted` result code.

For more information on working with I/O request queues, see "SCSIReleaseQCmd," beginning on page 1-46.

Note
This function is not yet implemented (as of the release date of this document).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `SCSIClearQueue` function can not be called by hardware interrupt handlers or secondary interrupt handlers. It can only be called with a connection ID for a SCSI device, not with a connection ID for a SCSI bus.

SCSIBusResetSync

Resets a SCSI bus.

```
OSStatus SCSIBusResetSync (
                    ConnectionID connID,
                    OSStatus *resultBuffer);
```

`connID` The connection ID to a SCSI bus. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`resultBuffer` A pointer to an `OSStatus` variable. On output, the variable contains information about the reset operation.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

The `SCSIBusResetSync` 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. `SCSIBusResetSync` differs from the `SCSIBusResetAsync`

function (page 1-48), only in that it is synchronous, so that the calling client is blocked pending completion.

Before the reset takes place, the SCSI family invokes the notification mechanism for each request that was already sent to a device at the time `SCSIBusResetSync` was called. The appropriate LUN queue is frozen for each request that is pending at the time of the reset, unless the `scsiSIMQNoFreeze` flag is set. (A client can set this flag by calling the `SCSISetIOOptions` function, described in "SCSISetIOOptions," beginning on page 1-57.) A client with a pending request receives a result code of `scsiSCSIBusReset`.

Note
A bus reset may be generated by an external source, as well as by a call to the `SCSIDeviceResetSync` function. ◆

The `SCSIBusResetSync` function interrupts SCSI communications and can cause data loss. You should use this function only as a last resort to restore operation in the event that a device refuses to release the bus. You can use the `SCSIDeviceResetSync` function (page 1-51) or the `SCSIDeviceResetAsync` function (page 1-53) to reset a single device when the SCSI bus is operational and the device is still responding to selection.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `SCSIBusResetSync` function can not be called by hardware interrupt handlers or secondary interrupt handlers. It can only be called with a connection ID for a SCSI bus, not with a connection ID for a SCSI device.

SCSIBusResetAsync

Resets a SCSI bus.

```
OSStatus SCSIBusResetAsync (
                    ConnectionID connID,
                    KernelNotification *kernelNot,
                    OSStatus *resultBuffer);
```

`connID` The connection ID to a SCSI bus. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`kernelNot` A pointer to a microkernel notification structure that specifies how you want to be notified when this request completes. For more information on the `KernelNotification` structure, see "Microkernel Notification Services" in *Inside Macintosh: Microkernel and Core System Services* (**to be provided** in a later release of Mac OS 8 documentation).

`resultBuffer` A pointer to a `OSStatus` variable. On an error return, the `SCSIBusResetAsync` function sets this variable immediately. Otherwise, the variable is set before the client's notification mechanism is invoked.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

The `SCSIBusResetAsync` 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. `SCSIBusResetAsync` differs from the `SCSIBusResetSync` function (page 1-48), only in that it is asynchronous, so that the calling client is not blocked pending completion.

Before the reset takes place, the SCSI family invokes the notification mechanism for each request that was already sent to a device at the time `SCSIBusResetAsync` was called. The appropriate LUN queue is frozen for each request that is pending at the time of the reset, unless the `scsiSIMQNoFreeze` flag is set. (A client can set this flag by calling the `SCSISetIOOptions` function, described in "SCSISetIOOptions," beginning on page 1-57.) A client with a pending request receives a result code of `scsiSCSIBusReset`.

Note
A bus reset may be generated by an external source, as well as by a call to the `SCSIDeviceResetSync` function. ◆

The `SCSIBusResetAsync` function interrupts SCSI communications and can cause data loss. You should use this function only as a last resort to restore operation in the event that a device refuses to release the bus. You can use the `SCSIDeviceResetSync` function (page 1-51) or the `SCSIDeviceResetAsync` function (page 1-53) to reset a single device when the SCSI bus is operational and the device is still responding to selection.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `SCSIBusResetAsync` function can not be called by hardware interrupt handlers or secondary interrupt handlers. It can only be called with a connection ID for a SCSI bus, not with a connection ID for a SCSI device.

SCSIDeviceResetSync

Resets a device on a SCSI bus.

```
OSStatus SCSIDeviceResetSync (
                    ConnectionID connID,
                    OSStatus *resultBuffer);
```

`connID` The connection ID to a SCSI device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`resultBuffer` A pointer to an `OSStatus` variable. On output, the variable contains information about the reset operation.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

The `SCSIDeviceResetSync` function attempts to send a `BUS DEVICE RESET` message to the target. If the device is currently on the bus, the plug-in asserts the /ATN signal and sends the message at the next message-out phase. If the target is not on the bus, the plug-in selects it and sends an `IDENTIFY` message followed by a `BUS DEVICE RESET` message. Unlike most other SCSI commands, the `BUS DEVICE RESET` message is issued even if the device queue is already frozen.

The `SCSIDeviceResetSync` function freezes the queue for all LUNs of the target device, unless the `scsiSIMQNoFreeze` flag is set. (A client can set this flag by calling the `SCSISetIOOptions` function, described in "SCSISetIOOptions," beginning on page 1-57.) Any disconnected requests (requests sent but not yet completed) and any requests in the pending queue are terminated with the `scsiBDRSent` result code.

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.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `SCSIDeviceResetSync` function can not be called by hardware interrupt handlers or secondary interrupt handlers. It can only be called with a connection ID for a SCSI device, not with a connection ID for a SCSI bus.

SCSIDeviceResetAsync

Resets a device on a SCSI bus.

```
OSStatus SCSIDeviceResetAsync (
                    ConnectionID connID,
                    KernelNotification *kernelNot,
                    OSStatus *resultBuffer);
```

`connID` The connection ID to a SCSI device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`kernelNot` A pointer to a microkernel notification structure that specifies how you want to be notified when this request completes. For more information on the `KernelNotification` structure, see "Microkernel Notification Services" in *Inside Macintosh: Microkernel and Core System Services* (**to be provided** in a later release of Mac OS 8 documentation).

`resultBuffer` A pointer to a `OSStatus` variable. On an error return, the `SCSIDeviceResetAsync` function sets this variable immediately. Otherwise, the variable is set before the client's notification mechanism is invoked.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

The `SCSIDeviceResetAsync` function attempts to send a `BUS DEVICE RESET` message to the target. If the device is currently on the bus, the plug-in asserts the /ATN signal and sends the message at the next message-out phase. If the target is not on the bus, the plug-in selects it and sends an `IDENTIFY` message followed by a `BUS DEVICE RESET` message. Unlike most other SCSI commands, the `BUS DEVICE RESET` message is issued even if the device queue is already frozen.

The `SCSIDeviceResetAsync` function freezes the queue for all LUNs of the target device, unless the `scsiSIMQNoFreeze` flag is set. (A client can set this flag by calling the `SCSISetIOOptions` function, described in "SCSISetIOOptions," beginning on page 1-57.) Any disconnected requests (requests sent but not yet

completed) and any requests in the pending queue are terminated with the `scsiBDRSent` result code.

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.

`SCSIDeviceResetAsync` differs from the `SCSIDeviceResetSync` function (page 1-51), only in that it is asynchronous, so that the calling client is not blocked pending completion.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `SCSIDeviceResetAsync` function can not be called by hardware interrupt handlers or secondary interrupt handlers. It can only be called with a connection ID for a SCSI device, not with a connection ID for a SCSI bus.

Setting SCSI Options

SCSISetHandshake

Sets up handshaking instructions globally for a device.

```
OSStatus SCSISetHandshake (
                    ConnectionID connID,
                    SCSIHandshakeObject handshake);
```

`connID` The connection ID to a SCSI bus or device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`handshake`	An array of `UInt16` (2-byte) hand-shaking values (page 1-21), terminated with the value 0. The array values are set by the calling client.
function result	A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

When you call the `SCSISetHandshake` function, you provide a `SCSIHandshakeObject` structure to specify handshaking instructions for blind transfers between a plug-in and a device. You store the handshaking instructions in an array of `UInt16` (2-byte) values and terminate the array with 0.

Note
In a later release, you will be able to set up handshaking instructions for each I/O request. ◆

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SCSISetTimeout

Sets up the time-out parameter for an I/O request.

```
OSStatus SCSISetTimeout (
                    ConnectionID connID,
                    Duration scsiTimeout,
                    UInt16 scsiSelectTimeout);
```

`connID` The connection ID to a SCSI bus or device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`scsiTimeout` The length of time the plug-in should allow before reporting a command timeout of the SCSI bus. You provide the time value in milliseconds. A value of 0 tells the SCSI family to use the default timeout value. For more information, see the description of `scsiTimeout` in "SCSI I/O Parameter Block," beginning on page 1-82.

`scsiSelectTimeout` The length of time the plug-in should allow before reporting a selection timeout of the SCSI bus. You provide the time value in milliseconds. A value of 0 tells the SCSI family to use the default timeout value. For more information, see the description of `scsiSelectTimeout` in "SCSI I/O Parameter Block," beginning on page 1-82.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

After opening a connection to a SCSI bus, you call the `SCSISetTimeout` function to set command and selection timeout values for operations performed on that bus. If you do not set timeout values, the SCSI family uses the default values specified by the SCSI standard.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SCSISetIOOptions

Sets option flags for an I/O request.

Note
When you call the `SCSISetIOOptions` function, you provide a `SCSIIOOptionsObject` structure (page 1-22). The `SCSISetIOOptions` function is currently used to set option flags globally before calling SCSI family client functions such as `SCSIExecIOSyncCmd` (page 1-33). In a future software release, clients will be able to specify I/O options as part of the function interface for SCSI family functions. Both the `SCSISetIOOptions` function and the `SCSIIOOptionsObject` data structure will be eliminated.

```
OSStatus SCSISetIOOptions (
                    ConnectionID connID,
                    SCSIIOOptionsObject ioOptions);
```

`connID` The connection ID to a SCSI bus or device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`ioOptions` Structure for setting two kinds of SCSI flags (page 1-22). You set flags in the `scsiFlags` field to indicate the transfer direction and any special handling required for this request. (See "SCSI Flags," beginning on page 1-66, for flag descriptions.) You set flags in the `scsiIOFlags` field to provide additional information describing the data transfer. (See "SCSI I/O Flags," beginning on page 1-70, for flag descriptions.)

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

This function is currently used only to set option flags globally before calling the `SCSIReleaseQCmd` function (page 1-46).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Obtaining Device and Bus Information

SCSIBusGetDeviceData

Provides information about the devices attached to a specified SCSI bus.

```
OSStatus SCSIBusGetDeviceData (
                    UInt8 *bus,
                    UInt32 reqCount,
                    UInt32 *actCount,
                    SCSIIOIteratorData *list);
```

`bus` A pointer to an unsigned 8-bit integer that specifies a bus number (such as 0 or 1). You set this field before calling `SCSIBusGetDeviceData`. If you set the value to `kSCSIAllBus` (page 1-29), the `SCSIBusGetDeviceData` function sets the `bus` field to the number of SCSI buses and the `actCount` field to the total number of devices found on all the buses.

`reqCount` A 32-bit unsigned integer that specifies the number of items for which you have allocated memory. Each item is a `SCSIIOIteratorData` structure. The first item is pointed to by the data iterator list parameter.

actCount	A pointer to an unsigned 32-bit integer. The function sets the actCount field to the number of devices it found attached to the specified bus. If you set the value of the bus field to kSCSIAllBus, the SCSIBusGetDeviceData function sets the actCount field to the total number of devices found on all SCSI buses.
list	A pointer to an array of one or more contiguous SCSIIOIteratorData structures (page 1-28), each of which describes one device on the specified bus.
function result	A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

The SCSIBusGetDeviceData function returns the actual number of devices (including the initiator) it found on the specified bus (or on all buses, if you set the bus field to the value kSCSIAllBus) in the actCount field. Before you call SCSIBusGetDeviceData, you allocate memory for a list of SCSIIOIteratorData structures and set the reqCount field to the number of items you have allocated. If you don't allocate enough memory for the number of devices on the bus, SCSIBusGetDeviceData will fill in information just for the number you allocate.

You can take one of several approaches for deciding how much memory to allocate for the device list:

- Allocate enough memory for the highest possible number of devices that can reside on the specified bus. If the actual number of devices returned is smaller (as it will be in most cases), you can deallocate the unused memory.

  This may be practical in the case where at most 8 or 16 devices are present. However, if the devices include LUNs and if you ask for all devices on all buses (which could include network devices), the highest possible number could theoretically be in the thousands, requiring a large initial allocation.

- Allocate enough memory for a reasonable number of devices (say 8 or 16). Since the allocation in this case is relatively small, you may choose not to deallocate unused memory. Always check the actCount field—if the actual count exceeds the number of structures you allocated memory for, increase your memory allocation to match the actual number of devices and call the SCSIBusGetDeviceData function a second time.

- Allocate no memory. Set the reqCount field to 0 and call SCSIBusGetDeviceData once to determine the number of devices. Allocate

enough memory for that number of devices and call `SCSIBusGetDeviceData` a second time to get the device data.

You can then use information from the list of `SCSIIOIteratorData` structures to open a connection to any of the devices found on the bus.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SCSIBusInquiryCmd

Retrieves configuration and capability information about a plug-in and its HBA.

```
OSStatus SCSIBusInquiryCmd (
                    ConnectionID connID,
                    SCSIBusInfo *resultBuffer);
```

`connID` The connection ID to a SCSI bus or device. You get a connection ID from the `SCSIOpenConnection` function (page 1-30).

`resultBuffer` A pointer to a `SCSIBusInfo` structure (page 1-23). The plug-in that handles the inquiry sets the fields of the structure.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

You can use this function to determine precisely what optional features a particular plug-in supports, such as synchronous mode or wide transfer mode.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Plug-in Constants and Data Types

A plug-in can support one or more data types for transferring data. When you call the `SCSIBusInquiryCmd` function (page 1-60), the plug-in returns the data types it supports in the `scsiDataTypes` field of the `SCSIBusInfo` parameter (`resultBuffer`). For more information, see "SCSI Data Type," beginning on page 1-14.

Plug-in Control Block Structure

PluginControlBlock

After a plug-in is loaded into memory and prepared for execution, the SCSI family calls its initialization function, `MySCSIPluginInitFunc` (page 1-98), passing a pointer to a `PluginControlBlock` structure to exchange initialization information with the plug-in. The SCSI family defines the `PluginControlBlock` data type as follows:

```
struct PluginControlBlock {
    UInt16  ioPBSize;               /* <- size of SCSI_IO_PBs required by plug-in */
    UInt16  oldCallCapable;         /* Not used */
    UInt16  busID;                  /* <- bus number for the registered bus */
    UInt8   simSlotNumber;          /* <- reserved */
```

```
    UInt8   simSRsrcID;             /* <- reserved */
    Ptr     simRegEntry;            /* -> The SIM's RegEntryIDPtr */
    UInt32  maxTargetID;            /* <- max Target ID of this bus */
    UInt32  initiatorID;            /* <- comes from the NVRAM */
    UInt32  scsiTimeout;            /* <- bus time out period */
    UInt32  scsiFlagsSupported;     /* <- scsiFlags supported by this plug-in */
    SInt16  scsiSelectTimeout;      /* <- selection time out period */
    UInt16  scsiIOFlagsSupported;   /* <- scsiIOFlags supported by this plug-in */
    UInt32  scsiDataTypes;          /* <- scsiDataType supported by this plug-in */
};
```

Field descriptions

`ioPBSize` The minimum size, in bytes, of the SCSI parameter block required by this plug-in. The plug-in returns this value.

`oldCallCapable` Not used.

`busID` The bus number of the SCSI bus controlled by this plug-in. The SCSI family generates the ID and sets this field. A bus number remains valid from system startup until either the system is shut down or the plug-in is removed. The plug-in returns this value.

`simSlotNumber` Reserved.

`simSRsrcID` Reserved.

`simRegEntry` A pointer, supplied by the family, to a `RegEntryRef` data structure describing the bus. The plug-in needs the data supplied by this field. For information on how the family acquires the pointer, see "Driver and Family Matching".

`maxTargetID` Maximum Target ID of this bus.

`initiatorID` The SCSI ID of the HBA managed by this plug-in. A plug-in obtains the ID from NVRAM and returns it.

`scsiTimeout` Bus time out period.

`scsiFlagsSupported` SCSI flags supported by this plug-in (page 1-66).

`scsiSelectTimeout` Selection time out period.

`scsiIOFlagsSupported` SCSI I/O flags (page 1-70) supported by this plug-in.

`scsiDataTypes` SCSI data types (page 1-14) supported by this plug-in.

Plug-in-Defined Function Types

SCSIPluginInitEntry

After a plug-in is instantiated, the SCSI family calls the initialization function provided by the plug-in. The plug-in then prepares itself to handle requests.

A plug-in uses the `SCSIPluginDispatchTable` structure (page 1-64) to export a pointer to its initialization function (and other functions). The function pointer is defined by the SCSI family as follows:

```
typedef OSStatus (*SCSIPluginInitEntry)(PluginControlBlock *pcb);
```

For information about creating your own initialization function, see the description of the `MySCSIPluginInitFunc` function (page 1-98). For more information on the `PluginControlBlock` structure, see "PluginControlBlock," beginning on page 1-61.

SCSIPluginActionEntry

When a SCSI client makes an I/O request by calling a connection-based function, the SCSI family passes the request to a plug-in by calling the action function provided by the plug-in. The plug-in then processes the request.

A plug-in uses the `SCSIPluginDispatchTable` structure (page 1-64) to export a pointer to its action function (and other functions). The function pointer is defined by the SCSI family as follows:

```
typedef void (*SCSIPluginActionEntry)(SCSI_PB *scsiPB);
```

For information about creating your own action function, see the description of the `MySCSIPluginActionFunc` function (page 1-99). For more information on the `SCSI_PB` structure, see "SCSI I/O Parameter Block," beginning on page 1-82.

SCSIPluginHandleBusEventEntry

The SCSI family calls a plug-in's bus event function to notify the plug-in that a bus event has occurred. The plug-in then processes the request.

A plug-in uses the `SCSIPluginDispatchTable` structure (page 1-64) to export a pointer to its bus event function (and other functions). The function pointer is defined by the SCSI family as follows:

```
typedef void (*SCSIPluginHandleBusEventEntry)(void *busEvent);
```

For information about creating your own bus event function, see the description of the `MySCSIPluginHandleBusEventFunc` function (page 1-100).

Plug-in Dispatch Table

SCSIPluginDispatchTable

Each SCSI family plug-in must export a dispatch table to make specific functions available to the family. The SCSI family dispatch table contains references to the plug-in's action function (page 1-99), its handle bus event function (page 1-100), and its initialization function (page 1-98).

The plug-in also uses the dispatch table to specify version information so that the family can verify its ability to work with the plug-in.

The SCSI family calls the Driver and Family Matching (DFM) Software to load each plug-in. Subsequently, the DFM returns a pointer to the dispatch table. For more information on the DFM, see "Driver and Family Matching" **to be provided**.

The SCSI family defines the `SCSIPluginDispatchTable` data type for plug-in dispatch tables.

```
struct  SCSIPluginDispatchTable
{
    SCSIPluginInfo                  header;
    SCSIPluginActionEntry           scsiPluginAction;
```

```
    SCSIPluginHandleBusEventEntry   scsiPluginHandleBusEvent;
    SCSIPluginInitEntry             scsiPluginInit;
};
```

Field descriptions

`header` Version information for the plug-in. A plug-in includes the SCSI family header file and uses the `kSCSIPluginVersion` constant from that file for its version number. The SCSI family can then check the version number to verify that it can work with the plug-in.

`scsiPluginAction` Address of plug-in action routine (page 1-99).

`scsiPluginHandleBusEvent` Address of plug-in routine to handle bus events (page 1-100).

`scsiPluginInit` Address of plug-in initialization routine (page 1-98).

The SCSI family defines the `SCSIPluginInfo` structure to supply version information for a SCSI plug-in.

```
struct  SCSIPluginInfo
{
    UInt32  version;
    UInt32  reserved1;
    UInt32  reserved2;
    UInt32  reserved3;
};
```

Field descriptions

`version` Version information for the plug-in.

`reserved1` Reserved.

`reserved2` Reserved.

`reserved3` Reserved.

The SCSI family defines the `kSCSIPluginVersion` enumerated type to specify a plug-in version number.

```
enum
{
    kSCSIPluginVersion = 0x02019600     /* date and version */
};
```

SCSI Flags

The SCSI family defines enumerated values for setting flag bits to provide information for a data transfer request. The SCSI flags specify a variety of information about the request. You set the flags in the `scsiFlags` field of the `SCSIFlagsObject` parameter (page 1-17) when calling the `SCSIExecIOSyncCmd` function (page 1-33), the `SCSIExecIOAsyncCmd` function (page 1-35), the `SCSIExecIOControlSyncCmd` function (page 1-37), or the `SCSIExecIOControlAsyncCmd` function (page 1-40).

You can determine which SCSI flags a plug-in supports by calling the `SCSIBusInquiryCmd` function (page 1-60) and checking the bits in the `scsiFlagsSupported` field.

```
enum {
    scsiDirectionMask       = 0xC0000000,       /* data direction mask  */
    scsiDirectionNone       = 0xC0000000,       /* no data transfer */
    scsiDirectionReserved   = 0x00000000,       /* reserved */
    scsiDirectionOut        = 0x80000000,       /* data out */
    scsiDirectionIn         = 0x40000000,       /* data in      */
    scsiDisableAutosense    = 0x20000000,       /* disable autosense feature */
    scsiFlagReservedA       = 0x10000000,       /* reserved */
    scsiFlagReserved0       = 0x08000000,       /* reserved */
    scsiCDBLinked           = 0x04000000,       /* not supported */
    scsiQEnable             = 0x02000000,       /* target queue actions are enabled */
    scsiCDBIsPointer        = 0x01000000,       /* CDB field contains a pointer */
    scsiFlagReserved1       = 0x00800000,       /* reserved */
    scsiInitiateSyncData    = 0x00400000,       /* attempt Sync data xfer and SDTR */
    scsiDisableSyncData     = 0x00200000,       /* disable sync, go to async */
    scsiSIMQHead            = 0x00100000,       /* place PB at the head of SIM Q */
    scsiSIMQFreeze          = 0x00080000,       /* return the SIM Q to frozen state */
    scsiSIMQNoFreeze        = 0x00040000,       /* disallow SIM Q freezing */
    scsiDoDisconnect        = 0x00020000,       /* definitely do disconnect */
    scsiDontDisconnect      = 0x00010000,       /* definitely don't disconnect */
    scsiDataReadyForDMA     = 0x00008000,       /* data buffer(s) are ready for DMA */
    scsiFlagReserved3       = 0x00004000,       /* reserved */
    scsiDataPhysical        = 0x00002000,       /* SG/buffer data ptrs are physical */
    scsiSensePhysical       = 0x00001000,       /* autosense buffer ptr is physical */
    scsiFlagReserved5       = 0x00000800,       /* reserved */
    scsiFlagReserved6       = 0x00000400,       /* reserved */
```

```
    scsiFlagReserved7       = 0x00000200,       /* reserved */
    scsiFlagReserved8       = 0x00000100        /* reserved */
};
```

The descriptions define the meaning of setting specific bits.

Flag descriptions

`scsiDirectionMask` Data direction mask.

`scsiDirectionNone` No data transfer associated with this request.

`scsiDirectionOut` A write request. The data transfer direction is from the CPU to the device.

`scsiDirectionIn` A read request. The data transfer direction is from the device to the CPU.

`scsiDisableAutosense` Disable the autosense feature (whereby the plug-in automatically sends a REQUEST SENSE command in response to a CHECK CONDITION status from the device).

`scsiCDBLinked` Not supported. Do not use this flag.

`scsiQEnable` Enable target queue actions (command queuing). This option may not be supported by all plug-ins. For more information on queueing of I/O requests, see "SCSIExecIOAsyncCmd," beginning on page 1-35 and "SCSIExecIOControlAsyncCmd," beginning on page 1-40

`scsiCDBIsPointer` The `scsiCDB` field of the `SCSIExecIOPB` parameter block (page 1-82) contains a pointer to a command descriptor block. If the bit is not set, the `scsiCDB` field contains the actual command descriptor block. In either case, the `scsiCDBLength` field contains the number of bytes in the command descriptor block.
The SCSI family connection-based interface never sets this bit because the family interface never uses a pointer.

`scsiInitiateSyncData` The plug-in 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. Not all plug-ins (or buses) support this capability.

`scsiDisableSyncData`
Disable synchronous data transfer. The plug-in 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. Not all plug-ins (or buses) support this capability.

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

`scsiSIMQFreeze` Freeze the plug-in queue after completing this transaction. See "SCSIReleaseQCmd," beginning on page 1-46 for information about using this flag.

`scsiSIMQNoFreeze` Disable plug-in queue freezing for this transaction.

`scsiDoDisconnect` Explicitly allow the device to disconnect.

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

`scsiDataReadyForDMA`
Data buffer is locked and non-cacheable. This option may not be supported by all plug-ins.

`scsiDataPhysical` Data buffer address is physical. This option may not be supported by all plug-ins.

`scsiSensePhysical` Autosense data pointer is physical. This option may not be supported by all plug-ins.

SCSI Function Codes

For compatibility with SCSI Manager 4.3, the SCSI family defines enumerated values for specifying SCSI operations. As indicated below, many of these values are no longer supported by the SCSI family. For more information on these values, see "SCSI Manager 4.3" in *Inside Macintosh: Devices*.

```
enum {
    SCSINop                 = 0x00, /* do nothing */
    SCSIExecIO              = 0x01, /* execute the specified IO */
```

```
    SCSIBusInquiry          = 0x03, /* get parameters for entire path of HBAs */
    SCSIReleaseQ            = 0x04, /* release frozen SIM queue for particular LUN */
    SCSIAbortCommand        = 0x10, /* abort the selected Control Block */
    SCSIResetBus            = 0x11, /* reset the SCSI bus */
    SCSIResetDevice         = 0x12, /* reset the SCSI device */
    SCSITerminateIO         = 0x13, /* terminate any pending IO  */
    SCSIGetVirtualIDInfo    = 0x80, /* find out which bus old ID is on */
    SCSILoadDriver          = 0x82, /* not supported */
    SCSIOldCall             = 0x84, /* not supported */
    SCSICreateRefNumXref    = 0x85, /* not supported */
    SCSILookupRefNumXref    = 0x86, /* not supported */
    SCSIRemoveRefNumXref    = 0x87, /* not supported */
    SCSIRegisterWithNewXPT  = 0x88, /* not supported */
    vendorUnique            = 0xC0  /* 0xC0 thru 0xFF */
};
```

Constant descriptions

`SCSINop` Null request. Provided for compatibility with the ANSI Common Access Method specification and as a debugging aid. For more information, refer to the *SCSI-2 Common access method transport and SCSI interface module* specification.

`SCSIExecIO` Execute a SCSI I/O transaction.

`SCSIBusInquiry` Bus inquiry.

`SCSIReleaseQ` Release a frozen plug-in queue.

`SCSIAbortCommand` Abort a SCSI command.

`SCSIResetBus` Reset the SCSI bus.

`SCSIResetDevice` Reset a SCSI device.

`SCSITerminateIO` Terminate an I/O transaction.

`SCSIGetVirtualIDInfo` Return the device identification structure of a virtual SCSI ID.

`SCSILoadDriver` Not supported.

`SCSIOldCall` Not supported.

`SCSICreateRefNumXref` Not supported.

`SCSILookupRefNumXref` Not supported.

`SCSIRemoveRefNumXref`
Not supported.

`SCSIRegisterWithNewXPT`
Not supported.

`vendorUnique` Requests in this range (0xC0 through 0xFF) are currently reserved.

Transfer Types

When you call the `SCSIExecIOSyncCmd` function (page 1-33), the `SCSIExecIOAsyncCmd` function (page 1-35), the `SCSIExecIOControlSyncCmd` function (page 1-37), or the `SCSIExecIOControlAsyncCmd` function (page 1-40), you specify a transfer type in the `scsiTransferType` field of the `SCSIFlagsObject` parameter (page 1-17).

Two transfer types are defined. A third-party plug-in can define additional transfer types.

```
enum {
    scsiTransferBlind   = 0,
    scsiTransferPolled
};
```

Enumerator descriptions

`scsiTransferBlind` Use DMA, if available; otherwise, perform a blind transfer. For a blind transfer, you must have previously supplied handshaking information by calling the `SCSISetHandshake` function (page 1-54). You specify the handshake data in the `SCSIHandshakeObject` (page 1-21) parameter passed to `SCSISetHandshake`.

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

SCSI I/O Flags

When you call the `SCSIExecIOSyncCmd` function (page 1-33), the `SCSIExecIOAsyncCmd` function (page 1-35), the `SCSIExecIOControlSyncCmd`

function (page 1-37), or the SCSIExecIOControlAsyncCmd function (page 1-40), you can instruct a plug-in to handle the transfer in specific ways by setting bits in the scsiIOFlags field of the SCSIFlagsObject parameter (page 1-17). SCSI I/O flags specify hardware-dependent features. The following constants are defined.

```
enum {  /* values for the scsiIOFlags field */
    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,       /* reserved */
    scsiIOFlagReserved8000      = 0x8000        /* reserved */
};
```

Enumerator descriptions

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 plug-in normally throws away the excess and the functions returns the scsiDataRunError result code. If this

flag is set, the plug-in 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. It may not be supported by all plug-ins.

`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 plug-in normally sends an arbitrary number of meaningless bytes (0xEE) until the target releases the bus. If this flag is set, the plug-in 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. It may not be supported by all plug-ins.

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

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

`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.

Feature Flags

The `SCSIBusInquiryCmd` function (page 1-60) returns information about a plug-in/HBA in the `scsiFeatureFlags` field of its `SCSIBusInfo` parameter (page 1-23). You can test for specific features using the following bit masks.

```
enum {
    scsiBusInternalExternalMask     = 0x000000C0,   /* bus internal/external mask */
    scsiBusInternalExternalUnknown  = 0x00000000,   /* unknown if bus inside/outside */
    scsiBusInternalExternal         = 0x000000C0,   /* bus goes 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,   /* not supported */
    scsiBusDifferential             = 0x00000004,   /* single-ended or differential */
    scsiBusFastSCSI                 = 0x00000002,   /* HBA supports fast SCSI */
    scsiBusDMAavailable             = 0x00000001    /* DMA is available         */
};
```

Enumerator descriptions

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

`scsiBusInternalExternal`
: The bus is both internal and external.

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

`scsiBusExternal`
: The bus extends outside of the computer.

`scsiBusCacheCoherentDMA`
: DMA is cache coherent.

`scsiBusOldCallCapable`
: Not supported (obsolete). A plug-in never sets this bit. If a client attempts to call an original SCSI Manager function, the function returns an error.

`scsiBusDifferential`
: The bus uses a differential SCSI interface. If the bit is clear, the bus uses a single-ended SCSI interface.

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

`scsiBusDMAavailable`
: DMA is available.

More Feature Flags

The `SCSIBusInquiryCmd` function (page 1-60) returns information about additional HBA features by setting fields of its `SCSIBusInfo` parameter (page 1-23), including bits in the `scsiHBAInquiry` field. You can test for these features using the following bit masks.

```
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 Sync Data Transfer Req message */
    scsiBusLinkedCDB    = 0x08,     /* not supported */
    scsiBusTagQ         = 0x02,     /* supports tag queue message */
    scsiBusSoftReset    = 0x01      /* supports soft reset */
};
```

Enumerator descriptions

`scsiBusMDP`	Supports the `MODIFY DATA POINTER` message.
`scsiBusWide32`	Supports 32-bit wide transfers.
`scsiBusWide16`	Supports 16-bit wide transfers.
`scsiBusSDTR`	Supports synchronous transfers.
`scsiBusLinkedCDB`	Not supported (obsolete). A plug-in never sets this bit.
`scsiBusTagQ`	Supports tagged queuing.
`scsiBusSoftReset`	Supports soft reset.

Unusual Features Flags

The `SCSIBusInquiryCmd` function (page 1-60) returns information about unusual hardware-dependent configuration features of a plug-in and its HBA in the `scsiWeirdStuff` field of its `SCSIBusInfo` parameter. These flags give a plug-in a way to tell a client that the plug-in cannot handle certain conditions. If so, it is the obligation of the client not to cause those conditions to occur.

You can test for these features using the following bit masks.

```
enum {
    scsiOddDisconnectUnsafeRead1    = 0x0001,
    scsiOddDisconnectUnsafeWrite1   = 0x0002,
    scsiBusErrorsUnsafe             = 0x0004,
    scsiRequiresHandshake           = 0x0008,
    scsiTargetDrivenSDTRSafe        = 0x0010,
    scsiOddCountForPhysicalUnsafe   = 0x0020
    scsiAbortCmdFixed               = 0x0040
};
```

Enumerator descriptions

`scsiOddDisconnectUnsafeRead1`

Indicates that a disconnect or other phase change on an odd byte boundary during a read operation results in inaccurate residual counts or data loss. If your device can disconnect on odd bytes, use polled transfers instead of blind. Note that some devices can only handle a transfer that starts on an address that is a multiple of 8—any other start address is considered odd.

`scsiOddDisconnectUnsafeWrite1`

Indicates that a disconnect or other phase change on a odd byte boundary during a write operation results in inaccurate residual counts or data loss. If your device can disconnect on odd bytes, use polled transfers instead of blind. Note that some devices can only handle a transfer that starts on an address that is a multiple of 8—any other start address is considered odd.

`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 plug-in 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 in the `SCSIExecIOPB` parameter block (page 1-82).

`scsiOddCountForPhysicalUnsafe`

Indicates that if you are using physical addresses, all counts must be even, and disconnects must occur on even byte boundaries. Because of the virtual memory system in

Mac OS 8, this error car rarely occur, and the use of this flag may eventually be discontinued.

`scsiAbortCmdFixed` Set if abort command is fixed to properly make callbacks

Slot Types

In the `scsiHBAslotType` field of its `SCSIBusInfo` parameter, the `SCSIBusInquiryCmd` function (page 1-60) returns the type of physical location for a plug-in's HBA. The following constants define types of physical location.

```
enum {
    scsiMotherboardBus      = 0x00,     /* built-in SCSI bus */
    scsiNuBus               = 0x01,     /* a NuBus card */
    scsiPDSBus              = 0x03,     /* a PDS card        */
    scsiPCIBus              = 0x04,     /* a PCI bus card */
    scsiPCMCIABus           = 0x05,     /* a PCMCIA card */
    scsiFireWireBridgeBus   = 0x06      /* connected through
                                           Firewire bridge */
};
```

Constant descriptions

`scsiMotherboardBus` HBA is on a built-in SCSI bus.

`scsiNuBus` HBA is on an expansion card in a NuBus slot.

`scsiPDSBus` HBA is on an expansion card in a processor-direct slot.

`scsiPCIBus` HBA is on an expansion card in a PCI slot.

`scsiPCMCIABus` HBA is on an expansion card in a PC Card slot.

`scsiFireWireBridgeBus`
HBA is connected through a **Firewire** bridge.

Scan Types

When you call the `SCSIBusInquiryCmd` function (page 1-60), you provide a pointer to a `SCSIBusInfo` structure (page 1-23). The `SCSIBusInfo` structure contains a `scsiScanFlags` field, as does the `SCSIBusInquiryPB` structure (page 1-87).

Note
The fields of the `SCSIBusInfo` structure are set by whoever does the scanning. The SCSI family currently performs the scan, but in a future software release, the plug-in will perform the scan. ◆

The SCSI family defines the following enumerated constants to test for values in the `scsiScanFlags` field.

```
enum
{
     scsiBusScansDevices    = 0x80, /* bus scans for and maintains device list */
     scsiBusScansOnInit     = 0x40, /* bus scans performed at power-up/reboot */
     scsiBusLoadsROMDrivers = 0x20  /* may load ROM drivers to support targets */
};
```

Enumerator descriptions

`scsiBusScansDevices`
bus scans for and maintains device list

`scsiBusScansOnInit`
bus scan performed at power-up/reboot

`scsiBusLoadsROMDrivers`
bus may load ROM drivers to support targets

Data Length Constants

The following constants define the length of certain fields in SCSI data structures.

```
enum {
    handshakeDataLength     = 8,    /* Handshake data length */
    maxCDBLength            = 16,   /* Space for the CDB bytes/pointer */
    vendorIDLength          = 16    /* ASCII string len for Vendor ID  */
};
```

Constant descriptions

`handshakeDataLength`
The number of `UInt16` (2-byte) elements in the `scsiHandshake` field in the `SCSIExecIOPB` parameter block

	(page 1-82) or the `scsiHandshake` field of the `SCSIHandshakeObject` structure.
`maxCDBLength`	The size, in bytes, of the `cdbBytes` field in the `CDB` union structure (page 1-78).
`vendorIDLength`	The size, in bytes, of the `scsiSIMVendor`, `scsiHBAVendor`, `scsiControllerFamily`, and `scsiControllerType` fields in the `SCSIBusInquiryPB` parameter block (page 1-87) and the `SCSIBusInfo` structure (page 1-23).

Command Descriptor Block Structure

CDB

You use the command descriptor block structure to pass SCSI commands to the `SCSIExecIOSyncCmd` function (page 1-33), the `SCSIExecIOAsyncCmd` function (page 1-35), the `SCSIExecIOControlSyncCmd` function (page 1-37), or the `SCSIExecIOControlAsyncCmd` function (page 1-40).

The command descriptor block structure is defined by the `CDB` data type. The value `maxCDBLength` is described in "Data Length Constants," beginning on page 1-77.

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

Field descriptions

`cdbPtr`	A pointer to a buffer containing a command descriptor block.
`cdbBytes`	A buffer in which you place one command descriptor block.

Scatter/Gather List Structure

SGRecord

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.

When you want to transfer data to or from a SCSI device using the `SCSIExecIOSyncCmd` function (page 1-33), the `SCSIExecIOAsyncCmd` function (page 1-35), the `SCSIExecIOControlSyncCmd` function (page 1-37), or the `SCSIExecIOControlAsyncCmd` function (page 1-40), you provide a pointer to a `SCSIDataObject` structure (page 1-14). The `scsiDataPtr` field in the `SCSIDataObject` structure may point to a scatter/gather list.

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 Parameter Block Header

SCSIHdr

When a client calls a SCSI family function such as `SCSIExecIOSyncCmd` (page 1-33) or `SCSIExecIOAsyncCmd` (page 1-35), the SCSI family server uses information passed in the function parameters to build a parameter block structure for use by the appropriate SCSI family plug-in. A SCSI family client is shielded from the workings of the parameter block—only developers of plug-ins need to know about the structure of a parameter block.

The parameter block structures used are nearly identical to those supported by SCSI Manager 4.3, although a few fields have been changed or are no longer supported. (For information on specific fields, see the reference sections for individual parameter block structures.) The SCSI family and plug-ins ignore any parameter block fields that are no longer used.

Macros such as `SCSIPBHdr` and `SCSI_IO` have been replaced by similarly named structures. The structures define fields identical to those in the original macros.

The parameter block structures used by the SCSI family consist of a common header (`SCSIHdr`) followed by function-specific fields, if any. This section describes the parameter block header common to all SCSI parameter block structures.

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

```
struct SCSIHdr
    SCSIHdr         *qLink;
    short           scsiReserved1;
    UInt16          scsiPBLength;
    UInt8           scsiFunctionCode;
    UInt8           scsiReserved2;
    OSErr           scsiResult;
    DeviceIdent     scsiDevice;
    SCSICallbackUPP scsiCompletion;
    UInt32          scsiFlags;
    BytePtr         *scsiDriverStorage;
```

```
    Ptr             scsiXPTprivate;
    long            scsiReserved3;
};
```

Field descriptions

`qLink`	A pointer to the next entry in the request queue. This field is used internally by the plug-in.
`scsiReserved1`	Reserved for use of plug-in.
`scsiPBLength`	The size of the parameter block, in bytes, including the parameter block header. The plug-in uses this value to verify that the parameter block has the correct length.
`scsiFunctionCode`	A function code that specifies the requested service. "SCSI Function Codes," beginning on page 1-68, lists these codes.
`scsiReserved2`	Reserved for use of plug-in
`scsiResult`	Set to the final result when the parameter block is returned to the SCSI family. "SCSI Family Result Codes," beginning on page 1-101, lists all result codes specific to the SCSI family.
`scsiDevice`	A 4-byte value that uniquely identifies the target device for a request. The `DeviceIdent` data type (page 1-27) designates the bus number, target SCSI ID, and LUN.
`scsiCompletion`	A pointer to a completion routine.
`scsiFlags`	Flags that indicate the transfer direction and any special handling required for an I/O request. See "SCSI Flags," beginning on page 1-66 for flag descriptions.
`scsiDriverStorage`	Reserved for driver or client use. For example, a client may use this field to store a pointer to its private storage.
`scsiXPTprivate`	Reserved for use by family.
`scsiReserved3`	Reserved for use by plug-in.

SCSI Parameter Block

SCSI_PB

The SCSI parameter block is defined by the `SCSI_PB` data type. Its fields are identical to those of the `SCSIHdr` data type (page 1-80).

```
struct SCSIAbortCommandPB {
    SCSIHdr *       qLink;              /* (internal use, must be nil on entry) */
    short           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 Identifier (bus+target+lun) */
    SCSICallbackUPP scsiCompletion;     /* -> Callback on completion function  */
    UInt32          scsiFlags;          /* -> assorted flags */
    BytePtr         scsiDriverStorage;  /* <> Ptr for driver private use */
    Ptr             scsiXPTprivate;     /* private field for use in XPT */
    long            scsiReserved3;      /* reserved */
};
```

SCSI I/O Parameter Block

SCSI_IO

The SCSI I/O parameter block is defined by the `SCSI_IO` data type.

```
struct SCSI_IO {
    struct SCSIHdr      *qLink;
    short               scsiReserved1;
    UInt16              scsiPBLength;
    UInt8               scsiFunctionCode;
```

```
UInt8               scsiReserved2;
OSErr               scsiResult;
DeviceIdent         scsiDevice;
SCSICallbackUPP     scsiCompletion;
UInt32              scsiFlags;
UInt8               *scsiDriverStorage;
Ptr                 scsiXPTprivate;
long                scsiReserved3;

UInt16              scsiResultFlags;
UInt16              scsiReserved3pt5;
UInt8               *scsiDataPtr;
SInt32              scsiDataLength;
UInt8               *scsiSensePtr;
SInt8               scsiSenseLength;
UInt8               scsiCDBLength;
UInt16              scsiSGListCount;
UInt32              scsiReserved4;
UInt8               scsiSCSIstatus;
SInt8               scsiSenseResidual;
UInt16              scsiReserved5;
long                scsiDataResidual;
CDB                 scsiCDB;
long                scsiTimeout;
UInt8               *scsiReserved5pt5;
UInt16              scsiReserved5pt6;
UInt16              scsiIOFlags;
UInt8               scsiTagAction;
UInt8               scsiReserved6;
UInt16              scsiReserved7;
UInt16              scsiSelectTimeout;
UInt8               scsiDataType;
UInt8               scsiTransferType;
UInt32              scsiReserved8;
UInt32              scsiReserved9;
UInt16              scsiHandshake[handshakeDataLength];
UInt32              scsiReserved10;
UInt32              scsiReserved11;
struct SCSI_IO      *scsiCommandLink;
UInt8               scsiSIMpublics[8];
UInt8               scsiAppleReserved6[8];
```

```
    UInt16              scsiCurrentPhase;
    short               scsiSelector;
    OSErr               scsiOldCallResult;
    UInt8               scsiSCSImessage;
    UInt8               XPTprivateFlags;
    UInt8               XPTextras[12];
};

typedef SCSI_IO SCSIExecIOPB;
```

Field descriptions

`qLink` A pointer to the next entry in the request queue. This field is used internally by the plug-in.

`scsiReserved1` Reserved for input.

`scsiPBLength` The size of the parameter block, in bytes, including the parameter block header. The plug-in uses this value to verify that the parameter block has the correct length.

`scsiFunctionCode` A function code that specifies the requested service. "SCSI Function Codes," beginning on page 1-68, lists these codes.

`scsiReserved2` Reserved for output.

`scsiResult` Set to the final result when the parameter block is returned to the SCSI family. "SCSI Family Result Codes," beginning on page 1-101, lists all result codes specific to the SCSI family.

`scsiDevice` A 4-byte value that uniquely identifies the target device for a request. The `DeviceIdent` data type (page 1-27) designates the bus number, target SCSI ID, and LUN.

`scsiCompletion` A pointer to a completion routine.

`scsiFlags` Flags that indicate the transfer direction and any special handling required for an I/O request. See "SCSI Flags," beginning on page 1-66 for flag descriptions.

`scsiDriverStorage` Reserved for plug-in use. For example, a plug-in may use this field to store a pointer to its private storage.

`scsiXPTprivate` Reserved.

`scsiReserved3` Reserved.

`scsiResultFlags` Flags set by the plug-in when certain conditions apply; otherwise the plug-in sets this field to 0. The flags modify the value in the `scsiResult` field. See "SCSI Family Result

	Codes," beginning on page 1-101, for a list of all result codes specific to the SCSI family.
`scsiReserved3pt5`	Reserved for use of plug-in.
`scsiDataPtr`	A pointer to a data buffer, scatter/gather list, or I/O table that you provide. You specify the data type in the `scsiDataType` field.
`scsiDataLength`	The amount of data you want to transfer, in bytes.
`scsiSensePtr`	A pointer to the autosense data buffer that you provide. If autosense is enabled, the plug-in returns `REQUEST SENSE` information in this buffer. (Autosense is enabled when you do not set the `scsiDisableAutosense` flag in the `scsiFlags` field of the parameter block header).
`scsiSenseLength`	The size of your autosense data buffer, in bytes.
`scsiCDBLength`	The length of your SCSI command descriptor block, in bytes.
`scsiSGListCount`	The number of elements in your scatter/gather list.
`scsiReserved4`	Reserved for use of plug-in.
`scsiSCSIstatus`	The status returned by the SCSI device. See "Data Length Constants" (page 1-77) for a list of values that a SCSI device can return.
`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. The plug-in sets this field.
`scsiReserved5`	Reserved for use of plug-in.
`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. The plug-in sets this field.
`scsiCDB`	An actual CDB or a pointer to a CDB. You provide one or the other depending on how you set the `scsiCDBIsPointer` flag in the `scsiFlags` field in the parameter block header.
`scsiTimeout`	The length of time the plug-in should allow before reporting a timeout of the SCSI bus. You provide the time value 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 plug-in 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 plug-in. The default timeout for the Apple-provided plug-in is infinite (that is, no timeout).
`scsiReserved5pt5`	Reserved for use of plug-in.
`scsiReserved5pt6`	Reserved for use of plug-in.
`scsiIOFlags`	Additional I/O flags you use to describe the data transfer. See "SCSI I/O Flags" (page 1-70) for flag descriptions.
`scsiTagAction`	Must be filled in by family if `scsiQEnable` flag is set. Used with tagged queueing.
`scsiReserved6`	Reserved for use of plug-in.
`scsiReserved7`	Reserved for use of plug-in.
`scsiSelectTimeout`	An optional `SELECT` timeout value, in milliseconds, that you can provide (see "SCSISetTimeout," beginning on page 1-55). 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 plug-ins 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 constants described in "SCSI Data Type," beginning on page 1-14.
`scsiTransferType`	The type of transfer—blind or polled—to use during the data phase. You specify the type using one of the constants described in "Transfer Types," beginning on page 1-70.
`scsiReserved8`	Reserved for use of plug-in.
`scsiReserved9`	Reserved for use of plug-in.
`scsiHandshake`	Handshaking instructions for blind transfers. You provide an array of 2-bytes values, terminated by 0. The plug-in polls for data ready after transferring the amount of data specified in each successive `scsiHandshake` entry. When it encounters a 0 value, the plug-in 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. For more information, see "SCSIHandshakeObject," beginning on page 1-21, "SCSISetHandshake," beginning

	on page 1-54, and "Data Length Constants," beginning on page 1-77.
`scsiReserved10`	Reserved for use of plug-in.
`scsiReserved11`	Reserved for use of plug-in.
`scsiCommandLink`	Not supported.
`scsiSIMpublics`	An additional input field available for use by plug-in developers.
`scsiCurrentPhase`	Reserved for use of plug-in.
`scsiSelector`	Reserved for use of plug-in.
`scsiOldCallResult`	Reserved for use of plug-in.
`scsiSCSIMessage`	Reserved for use of plug-in.
`XPTprivateFlags`	Reserved for use of SCSI family.
`XPTextras[12]`	Reserved for use of SCSI family.

SCSI Bus Inquiry Parameter Block

SCSIBusInquiryPB

The SCSI bus inquiry parameter block is defined by the `SCSIBusInquiryPB` data type.

```
struct SCSIBusInquiryPB
{
    SCSIHdr *       qLink;              /* (internal use, must be nil on entry) */
    short           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 Identifier (bus+target+lun) */
    SCSICallbackUPP scsiCompletion;     /* -> Callback on completion function  */
    UInt32          scsiFlags;          /* -> assorted flags */
    BytePtr         scsiDriverStorage;  /* <> Ptr for driver private use */
    Ptr             scsiXPTprivate;     /* private field for use in XPT */
```

```
long            scsiReserved3;      /* reserved */

UInt16          scsiEngineCount;        /* <- Number of engines on HBA */
UInt16  scsiMaxTransferType;    /* <- Number of transfer types for this HBA */
UInt32  scsiDataTypes;          /* <- which data types this plug-in supports */
UInt16  scsiIOpbSize;           /* <- Size of SCSI_IO PB for this SIM/HBA */
UInt16  scsiMaxIOpbSize;        /* <- Size of max SCSI_IO PB for all SIM/HBAs */
UInt32  scsiFeatureFlags;       /* <- Supported features flags field */
UInt8   scsiVersionNumber;      /* <- Version number for the plug-in/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;     /* <- Ptr to plug-in private data area */
UInt32  scsiSIMPrivatesSize;    /* <- Size of plug-in private data area */
UInt32  scsiAsyncFlags;         /* <- Event cap. for Async Callback */

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;
UInt16  scsiMaxTarget;          /* <- maximum Target number supported */
UInt16  scsiMaxLUN;             /* <- maximum Logical Unit number supported */
char    scsiSIMVendor[ vendorIDLength ];
                                /* <- Vendor ID of plug-in (or XPT if bus<FF) */
char    scsiHBAVendor[ vendorIDLength ];
                                /* <- Vendor ID of the HBA */
char    scsiControllerFamily[ vendorIDLength ];
                                /* <- Family of SCSI Controller */
char    scsiControllerType[ vendorIDLength ];
                                /* <- Specific Model of SCSI Controller used */
char    scsiXPTversion[4];      /* <- version number of XPT */
char    scsiSIMversion[4];      /* <- version number of plug-in */
char    scsiHBAversion[4];      /* <- version number of HBA */

UInt8   scsiHBAslotType;        /* <- type of "slot" that this HBA is in */
UInt8   scsiHBAslotNumber;      /* <- slot number of this HBA */
UInt16  scsiSIMsRsrcID;         /* <- resource ID of this plug-in */
```

```
    UInt16  scsiBIReserved3;        /* Reserved. */
    UInt16  scsiAdditionalLength;   /* <- additional BusInquiry PB len */
};
```

Field descriptions

`qLink`	A pointer to the next entry in the request queue. This field is used internally by the plug-in. The `qLink` pointer points to a `SCSIHdr` structure, a common header structure included in all SCSI family parameter block structures (page 1-82).
`scsiReserved1`	Reserved for input.
`scsiPBLength`	The size of the parameter block, in bytes, including the parameter block header. The plug-in uses this value to verify that the parameter block has the correct length. (The `SCSIBusInquiryCmd` function (page 1-60) returns the minimum size in the `scsiIOpbSize` field of its parameter block.)
`scsiFunctionCode`	A function code that specifies the requested service. "SCSI Function Codes," beginning on page 1-68, lists these codes.
`scsiReserved2`	Reserved for output.
`scsiResult`	Set to the final result when the parameter block is returned to the SCSI family. "SCSI Family Result Codes," beginning on page 1-101, lists all result codes specific to the SCSI family.
`scsiDevice`	A 4-byte value that uniquely identifies the target device for a request. The `DeviceIdent` data type (page 1-27) designates the bus number, target SCSI ID, and LUN.
`scsiCompletion`	A pointer to a completion routine.
`scsiFlags`	Flags that indicate the transfer direction and any special handling required for an I/O request. See "SCSI Flags," beginning on page 1-66 for flag descriptions.
`scsiDriverStorage`	Reserved for plug-in use. For example, a plug-in may use this field to store a pointer to its private storage.
`scsiXPTprivate`	Reserved.
`scsiReserved3`	Reserved.
`scsiEngineCount`	The number of engines on the HBA. This value is 0 for a built-in SCSI bus. See the ANSI Common Access Method specification for information about HBA engines.

`scsiMaxTransferType`
: The number of transfer types supported by the plug-in. A plug-in supports all transfer types that are specified by a constant value equal to or less than the value it returns here. For example, if a plug-in returns the `scsiTransferPolled` constant here, it means that the plug-in supports both the blind and polled transfer types. See"Transfer Types," beginning on page 1-70 for a description of the defined types.

`scsiDataTypes`
: A bit mask specifying the data types supported by the plug-in/HBA. See "A plug-in can support one or more data types for transferring data. When you call the `SCSIBusInquiryCmd` function (page 1-60), the plug-in returns the data types it supports in the `scsiDataTypes` field of the SCSIBusInfo parameter (resultBuffer). For more information, see "SCSI Data Type," beginning on page 1-14.," beginning on page 1-61 for more information.

`scsiIOpbSize`
: The minimum size of a SCSI I/O parameter block for this plug-in.

`scsiMaxIOpbSize`
: The maximum size of a SCSI I/O parameter block for all currently registered plug-ins. In other words, the largest parameter block size currently registered.

`scsiFeatureFlags`
: Flags that describe various physical characteristics of the SCSI bus. See "Feature Flags" (page 1-72) for flag definitions.

`scsiVersionNumber`
: The version number of the plug-in/HBA.

`scsiHBAInquiry`
: Flags describing the capabilities of the bus. See "More Feature Flags" (page 1-73) for flag definitions.

`scsiTargetModeFlags`
: Reserved.

`scsiScanFlags`
: On input, scan related feature flags (page 1-72).

`scsiSIMPrivatesPtr`
: A pointer to the plug-in's private storage.

`scsiSIMPrivatesSize`
: The size of the plug-in's private storage, in bytes.

`scsiAsyncFlags`
: Reserved.

`scsiHiBusID`
: The highest bus number currently registered in the Name Registry. The SCSI family provides this value. If no buses are registered, it sets this field to 0xFF.

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

`scsiBIReserved0` Reserved.

`scsiBIReserved1` Reserved.

`scsiFlagsSupported`
A bit mask that defines which `scsiFlags` bits the plug-in supports.

`scsiIOFlagsSupported`
A bit mask that defines which `scsiIOFlags` bits the plug-in supports.

`scsiWeirdStuff` Flags that identify unusual aspects of a plug-in's operation. See "Unusual Features Flags," beginning on page 1-74, for flag definitions.

`scsiMaxTarget` The highest SCSI bus ID supported by the HBA. For a standard SCSI-II HBA, the value is 7; for an HBA that supports wide transfer, the value is 15.

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

`scsiSIMVendor` A null-terminated ASCII text string that identifies the plug-in vendor. On Macintosh computers, for example, the function returns `'Apple Computer \0'` for a built-in SCSI bus.

`scsiHBAVendor` A null-terminated ASCII text string that identifies the HBA vendor. On Macintosh computers, for example, the function returns `'Apple Computer \0'` for a built-in SCSI bus.

`scsiControllerFamily`
An optional null-terminated 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`
An optional null-terminated ASCII text string that identifies the specific type of SCSI controller chip. This information is provided at the discretion of the HBA vendor.

`scsiXPTversion` Not used.

`scsiSIMversion` A 4-byte `NumVersion` data structure that identifies the version number of the plug-in. (Formerly an ASCII text

string. For data stored in the old style, the first byte will be a printable ASCII character.)

`scsiHBAversion` A 4-byte `NumVersion` data structure that identifies the version number of the HBA. (Formerly an ASCII text string. For data stored in the old style, the first byte will be a printable ASCII character.)

`scsiHBAslotType` The slot type, if any, used by this HBA. Slot types are defined in "Slot Types" (page 1-76).

`scsiHBAslotNumber` Reserved.

`scsiSIMsRsrcID` Reserved.

`scsiBIReserved3` Reserved.

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

SCSI Abort Command Parameter Block

SCSIAbortCommandPB

The abort command parameter block is defined by the `SCSIAbortCommandPB` data type. Except for the `scsiIOptr` field, its fields are identical to those of the `SCSIHdr` data type (page 1-80).

```
struct SCSIAbortCommandPB {
    SCSIHdr *       qLink;              /* (internal use, must be nil on entry) */
    short           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 Identifier (bus+target+lun) */
    SCSICallbackUPP scsiCompletion;     /* -> Callback on completion function  */
    UInt32          scsiFlags;          /* -> assorted flags */
```

```
    BytePtr         scsiDriverStorage;  /* <> Ptr for driver private use */
    Ptr             scsiXPTprivate;     /* private field for use in XPT */
    long            scsiReserved3;      /* reserved */
    SCSI_IO         scsiIOptr;          /* Pointer to PB for request to abort */
};
```

Field descriptions

`scsiIOptr` A pointer to a parameter block for the I/O command to be aborted.

Terminate I/O Parameter Block

SCSITerminateIOPB

The terminate command parameter block is defined by the `SCSITerminateIOPB` data type. Its fields are identical to those of the `SCSIAbortCommandPB` data type (page 1-92).

```
struct SCSITerminateIOPB {
    SCSIHdr *       qLink;              /* (internal use, must be nil on entry) */
    short           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 Identifier (bus+target+lun) */
    SCSICallbackUPP scsiCompletion;     /* -> Callback on completion function  */
    UInt32          scsiFlags;          /* -> assorted flags */
    BytePtr         scsiDriverStorage;  /* <> Ptr for driver private use */
    Ptr             scsiXPTprivate;     /* private field for use in XPT */
    long            scsiReserved3;      /* reserved */
    SCSI_IO         scsiIOptr;          /* Pointer to PB for request to terminate */
};
```

Field descriptions

`scsiIOptr` A pointer to a parameter block for the I/O command to be terminated.

Reset Bus Parameter Block

SCSIResetBusPB

The SCSI reset bus parameter block is defined by the `SCSIResetBusPB` data type. Its fields are identical to those of the `SCSIHdr` data type (page 1-80).

```
struct SCSIResetBusPB
    struct SCSIHdr  *qLink;
    short           scsiReserved1;
    UInt16          scsiPBLength;
    UInt8           scsiFunctionCode;
    UInt8           scsiReserved2;
    OSErr           scsiResult;
    DeviceIdent     scsiDevice;
    SCSICallbackUPP scsiCompletion;
    UInt32          scsiFlags;
    UInt8           *scsiDriverStorage;
    Ptr             scsiXPTprivate;
    long            scsiReserved3;
};
```

Reset Device Parameter Block

SCSIResetDevicePB

The SCSI reset device parameter block is defined by the `SCSIResetDevicePB` data type. Its fields are identical to those of the `SCSIHdr` data type (page 1-80).

```
struct SCSIResetDevicePB
    struct SCSIHdr  *qLink;
    short           scsiReserved1;
    UInt16          scsiPBLength;
    UInt8           scsiFunctionCode;
    UInt8           scsiReserved2;
    OSErr           scsiResult;
    DeviceIdent     scsiDevice;
    SCSICallbackUPP scsiCompletion;
    UInt32          scsiFlags;
    UInt8           *scsiDriverStorage;
    Ptr             scsiXPTprivate;
    long            scsiReserved3;
};
```

Release Queue Parameter Block

SCSIReleaseQPB

The SCSI release queue parameter block is defined by the `SCSIReleaseQPB` data type. Its fields are identical to those of the SCSI parameter block header (page 1-80).

```
struct SCSIReleaseQPB
    struct SCSIHdr  *qLink;
    short           scsiReserved1;
    UInt16          scsiPBLength;
    UInt8           scsiFunctionCode;
    UInt8           scsiReserved2;
    OSErr           scsiResult;
    DeviceIdent     scsiDevice;
    SCSICallbackUPP scsiCompletion;
    UInt32          scsiFlags;
    UInt8           *scsiDriverStorage;
```

```
    Ptr             scsiXPTprivate;
    long            scsiReserved3;
};
```

Plug-in Functions

The SCSI family server exports the `SCSIFamBusEventForSIM` (page 1-96) and `SCSIFamMakeCallback` (page 1-97) functions for a SCSI plug-in to call. A SCSI family plug-in in turn uses the plug-in dispatch table (page 1-64) to provide the SCSI family server with references to its action function (page 1-99), its handle bus event function (page 1-100), and its initialization function (page 1-98).

Exported by the SCSI Family

SCSIFamBusEventForSIM

Allows a plug-in's hardware interrupt handler to notify the plug-in that a bus event has occurred.

```
extern OSStatus SCSIFamBusEventForSIM (UInt32 busID, void *busEvent);
```

`busID` The bus ID of the bus controlled by the plug-in. The plug-in originally obtains the ID when the SCSI family calls its initialization function, `SCSIPluginInitEntry` (page 1-98). The plug-in is responsible for making the ID available to its hardware interrupt handler.

`busEvent` A pointer to a private structure, defined and allocated by the plug-in, in which the hardware interrupt handler records information about the bus event. The SCSI family does not interpret its contents.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

A plug-in notifies its bus event handler function of bus events by calling the SCSI family's `SCSIFamBusEventForSIM` function. The family responds by calling the plug-in's `MySCSIPluginHandleBusEventFunc` function (page 1-100) at task level, passing a pointer to the plug-in's private bus event structure.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
No	Yes	Yes

CALLING RESTRICTIONS

The `SCSIFamBusEventForSIM` function can be called at hardware or secondary interrupt level. However, a plug-in should only call this function from one or the other interrupt level—it should not mix calls from both levels because unprotected data may be corrupted.

SCSIFamMakeCallback

Informs the SCSI family that an I/O request is complete.

```
extern void SCSIFamMakeCallback (SCSI_PB *req);
```

`req` A pointer to the SCSI parameter block for the completed request. The SCSI family passes this parameter block to the plug-in when it calls the plug-in's action function, `SCSIPluginActionEntry` (page 1-99), to start a request. Before calling the `SCSIFamMakeCallback` function, the plug-in provides an output value for each field in the parameter block that requires one. See the individual parameter block descriptions for the output requirements of each type of request.

DISCUSSION

When a plug-in has finished processing an I/O request, it must notify the SCSI family that the request is complete by calling `SCSIFamMakeCallback`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
No	No	No

CALLING RESTRICTIONS

The `SCSIFamMakeCallback` function is not reentrant. It can be called only by a plug-in's main code (running at task level). It cannot be called from a plug-in's hardware interrupt handler.

SCSI Plug-in-Defined Functions

MySCSIPluginInitFunc

Initializes a plug-in.

```
OSStatus MySCSIPluginInitFunc (PluginControlBlock *pcb);
```

`pcb` A pointer to a plug-in control block structure. On input, the SCSI family provides values for the structure's `simRegEntry` and `busID` fields. On output, the plug-in provides values for many fields in the structure. For more information on the fields of the `PluginControlBlock` structure, see "PluginControlBlock," beginning on page 1-61.

function result A result code. See "SCSI Family Result Codes" (page 1-101) for a list of possible result codes.

DISCUSSION

The SCSI family calls this function to initialize a plug-in after the plug-in is loaded into memory and prepared for execution. The plug-in must respond by initializing its software structures and the HBA.

The `SCSIPluginInitEntry` type (page 1-63) defines the plug-in's initialization function.

SPECIAL CONSIDERATIONS

Only the SCSI family calls `MySCSIPluginInitFunc`. This function is not reentrant. It is always called at task level.

MySCSIPluginActionFunc

Passes a client request to a plug-in.

```
void MySCSIPluginActionFunc (SCSI_PB *scsiPB);
```

`scsiPB` A pointer to a SCSI parameter block for a client request. The request types are listed in "SCSI Function Codes," beginning on page 1-68. The parameter blocks corresponding to different request types are described individually. For a given type of request, see the appropriate parameter block description for input values a client provides and output values a plug-in provides.

DISCUSSION

When you call the `SCSIExecIOSyncCmd` function (page 1-33), the `SCSIExecIOAsyncCmd` function (page 1-35), and other SCSI family client functions, the SCSI family server uses information passed in the function parameters to build a parameter block structure to pass to the appropriate SCSI family plug-in. The type of parameter block used depends on the specific family function called and the parameters passed to that function. The parameter blocks are nearly identical to those supported by SCSI Manager 4.3, although some fields have been changed or are no longer supported. (For information on specific fields, see the reference sections for individual parameter block structures.) The SCSI family and plug-ins ignore any parameter block fields that are no longer used.

The SCSI family calls the `MySCSIPluginActionFunc` function to notify the plug-in of an I/O request. The action function defined by the plug-in must be of type

SCSIPluginActionEntry (page 1-63). The plug-in exports the function in its dispatch table (page 1-64).

The MySCSIPluginActionFunc function is responsible for handling the I/O request. It determines which bus the request is directed to and adds the request to the HBA's pending queue. When the SCSI bus is free and there's a request in the pending queue, the request is moved to the active queue and a SCSI selection is started for the request.

In executing the request, the plug-in must conform to the behavior defined for each type of service available through the SCSI family.

When request processing is complete, the plug-in stores a result code in the scsiResult field of the parameter block. The code should be appropriate for the request being processed. Then the plug-in notifies the SCSI family that processing is complete by calling the SCSIFamMakeCallback function (page 1-97).

SPECIAL CONSIDERATIONS

Only the SCSI family calls SCSIPluginAction. This function is not reentrant. It is always called at task level.

MySCSIPluginHandleBusEventFunc

Notifies a plug-in that a bus event has occurred.

```
void MySCSIPluginHandleBusEventFunc (void *busEvent);
```

busEvent — A pointer to a private structure, defined and allocated by the plug-in, that contains information about the bus event. The SCSI family obtains the pointer when the plug-in's hardware interrupt handler calls the SCSIFamBusEventForSIM function (page 1-96).

DISCUSSION

The SCSI family calls the MySCSIPluginHandleBusEventFunc function to notify a plug-in about a bus event. Typically, the bus event is an I/O completion, but it can also be an error condition on the bus.

When an interrupt occurs, the plug-in hardware interrupt handler calls the `SCSIFamBusEventForSIM` function, alerting the SCSI family. The SCSI family responds by calling `SCSIPluginHandleBusEvent`. The SCSI family passes the bus event parameter provided by the handler to the plug-in without interpreting its content.

If an I/O request completes as a result of the bus event, the plug-in needs to set all necessary output fields in the request's parameter block, and call the `SCSIFamMakeCallback` function to tell the SCSI family that the request is complete.

The `SCSIPluginHandleBusEventEntry` type (page 1-100) defines the plug-in's bus event function.

SPECIAL CONSIDERATIONS

Only the SCSI family calls `MySCSIPluginHandleBusEventFunc`. This function is not reentrant. It is always called at task level.

SCSI Family Result Codes

The result codes specific to the SCSI family are listed below. In addition, SCSI family client functions may also return result codes from microkernel services. You can read about those result codes in *Inside Macintosh: Microkernel and Core System Services*, **to be provided** in a later release of Mac OS 8 documentation.

The value of each SCSI family result code is computed by adding a value to the enumerated constant `scsiErrorBase`, which is defined as follows:

```
enum {
scsiErrorBase= -7936
};
```

For example, the result code `scsiRequestAborted` is defined as follows:

```
scsiRequestAborted = scsiErrorBase + 2
                            /* -7934 = PB request aborted by the host */
```

noErr	0	No error
scsiRequestInProgress	1	Parameter block request is in progress
scsiRequestAborted	-7934	Parameter block request aborted by the host
scsiUnableToAbort	-7933	Unable to abort parameter block request
scsiNonZeroStatus	-7932	The target returned non-zero status upon completion of the request
scsiUnableToTerminate	-7927	Unable to terminate I/O parameter block request
scsiSelectTimeout	-7926	Target selection timeout
scsiCommandTimeout	-7925	The timeout value for this parameter block was exceeded and the parameter block was aborted
scsiIdentifyMessageRejected	-7924	The target issued a REJECT message in response to the IDENTIFY message; the LUN probably does not exist
scsiMessageRejectReceived	-7923	REJECT message received
scsiSCSIBusReset	-7922	Execution of this parameter block was halted because of a SCSI bus reset
scsiParityError	-7921	An uncorrectable parity error occurred
scsiAutosenseFailed	-7920	Automatic REQUEST SENSE command failed
scsiDataRunError	-7918	Data overrun/underrun error
scsiUnexpectedBusFree	-7917	Unexpected bus free phase
scsiSequenceFailed	-7916	Target bus phase sequence failure
scsiWrongDirection	-7915	Data phase was in an unexpected direction
scsiBDRsent	-7913	A SCSI bus device reset (BDR) message was sent to the target
scsiTerminated	-7912	Parameter block request terminated by the host
scsiNoNexus	-7911	Nexus is not established
scsiCDBReceived	-7910	The SCSI CDB was received
scsiTooManyBuses	-7888	plug-in registration failed because the XPT registry is full

scsiBusy	-7887	SCSI subsystem is busy
scsiProvideFail	-7886	Unable to provide the requested service
scsiDeviceNotThere	-7885	SCSI device not installed or available
scsiNoHBA	-7884	No HBA detected
scsiDeviceConflict	-7883	Attempt to register more than one driver to a device (obsolete)
scsiNoSuchXref	-7882	No driver has been cross-referenced with this device (obsolete)
scsiQLinkInvalid	-7881	The qLink field was not 0
scsiPBLengthError	-7872	The parameter block length is too small for this plug-in
scsiFunctionNotAvailable	-7871	The requested function is not supported by this plug-in
scsiRequestInvalid	-7870	The parameter block request is invalid
scsiBusInvalid	-7869	The bus ID is invalid
scsiTIDInvalid	-7868	The target ID is invalid
scsiLUNInvalid	-7867	The logical unit number is invalid
scsiIDInvalid	-7866	The initiator ID is invalid
scsiDataTypeInvalid	-7865	plug-in does not support the requested scsiDataType
scsiTransferTypeInvalid	-7864	The scsiTransferType is not supported by this plug-in
scsiCDBLengthInvalid	-7863	The CDB length supplied is not supported by this plug-in; typically this means it was too big
kSCSITargetProbeInvalidState	-7862	Internal bus prober software error
scsiBadObjectID	-7861	Object ID on microkernel message to family does not reference a known message port
scsiBadDataLength	-7860	On a read request, the scsiDataLength field of SCSI I/O parameter block contains 0
scsiPartialPrepared	-7859	Cannot lock down enough memory for I/O transfer
scsiBadPBSize	-7872	Incorrect parameter block size

`scsiInvalidMsgType`	-7858	Invalid message type; internal error
`scsiInvalidRegID`	-7857	Invalid registry entry ID; internal error
`scsiBadConnID`	-7856	Bad connection ID; internal error
`scsiBadIOTag`	-7855	Bad I/O tag; internal error
`scsiIOInProgress`	-7854	Cannot close connection, I/O in progress; internal error
`scsiTargetReserved`	-7853	Target ID is reserved; cannot make connection
`scsiTargetInUse`	-7852	Target is in use by another client
`scsiNoReserveOnBus`	-7851	Cannot make reserved connection to a bus
`scsiBadConnType`	-7850	Invalid connection type

Glossary

access control The ability to open a SCSI connection to a bus or device with a specified access, either shared (other clients may also open a connection) or reserved (no other client may open a connection). A bus connection must be shared; a device connection may be shared or reserved.

autosense A feature of SCSI Manager 4.3 that automatically sends a REQUEST SENSE command in response to a CHECK CONDITION status, and retrieves the sense data.

autosense buffer Buffer in which sense data generated by the autosense feature is stored.

big-endian Used to describe data formatting in which each field is addressed by referring to its most significant byte. Compare **little-endian**, **mixed-endian**.

bit-bucketing The practice of throwing away excess data bytes when a target tries to supply more data than the initiator expects.

connection When used with the SCSI family, a logical path to a SCSI bus or a SCSI device. A connection controls access to its bus or device. Access to a device may be shared or reserved; access to a bus must be shared.

connection ID A value that uniquely identifies a connection. It is assigned by the Mac OS when a new connection is opened.

FireWire A high-speed peripheral bus using IEEE 1394—a fast serial port protocol.

little-endian Used to describe data formatting in which each field is addressed by referring to its least significant byte. Compare **big-endian**, **mixed-endian**.

memory list A table of values in which each entry describes a range in memory to serve as a source or destination in an I/O operation. When a memory list is prepared by a client running in user mode, the client must pass the list to the SCSI family to lock down the specified memory to physical addresses. Locking the memory prevents a page fault from occurring during an I/O operation. When a memory list is prepared by a client running in supervisor mode, the client must lock down the memory before passing the list to the SCSI family.

mixed-endian The ability of a computer system, such as Power Macintosh, to support both **big-endian** and **little-endian** data formats.

scatter/gather list A data type consisting of one or more elements, each of which describes the location and size of one data buffer.

SCSI execution tag A value, returned by Mac OS 8, that uniquely identifies an I/O request. The tag can later be used to abort or terminate the I/O request it identifies. The SCSI family uses the `SCSIExecIOTag` data type to store an execution tag.

SCSI interface module (SIM) A software module between the transport (XPT) and the host bus adapter (HBA) in SCSI Manager 4.3. The SIM processes and executes SCSI requests, and provides a hardware-independent interface to the HBA. In the Mac OS 8 SCSI family, a SIM is a SCSI family plug-in.

SCSI Family That part of the I/O system that manages the transfer of data between a Macintosh computer and peripheral devices connected through the Small Computer System Interface (SCSI). The SCSI family provides a connection-based interface for clients and a parameter block-based interface for plug-ins. The SCSI family is responsible for routing I/O requests to the proper plug-in, notifying the caller when a request is complete, maintaining compatibility with the SCSI Manager 4.3 interface, and isolating plug-ins from comprehensive knowledge of (and access to) other operating system components.

Index

T

U

V