HBA Device Driver Programming Guide

Hardware & Drivers > SCSI

2006-05-23

Contents

Figures and Tables

Introduction to HBA Device Driver Programming Guide

A host bus adapter (HBA) device driver transmits I/O requests between a computer and a set of storage devices or network nodes. Storage HBA devices can manage sets of parallel SCSI, Fibre Channel, SAS (Serial Attached SCSI), or SATA (Serial ATA) drives, sometimes in combination.

This document provides an overview of HBA device drivers in Mac OS X and explains how they process I/O requests. It also guides you through the development of a custom HBA driver and provides suggestions for increasing its performance.

Organization of This Document

This document contains the following chapters:

- "HBA Devices in Mac OS X" (page 9) presents an overview of the way HBA drivers work in Mac OS X and describes how an I/O request travels from an application to the hardware and back again.
- "Developing an HBA Driver" (page 15) contains guidelines for implementing a custom HBA driver.
- "Improving Performance" (page 25) describes several things you should take into account when creating an HBA driver to achieve the best performance for your hardware.
- "Document Revision History" (page 29) lists the revisions of this document.

See Also

The ADC Reference Library contains documents on device driver development for Mac OS X, including sample drivers and API reference.

- *I/O Kit Fundamentals* describes the architecture of the I/O Kit, the object-oriented framework for developing Mac OS X device drivers.
- *Mass Storage Device Driver Programming Guide* describes how to develop in-kernel device drivers for mass storage devices.
- *Kernel Framework Reference* contains API reference for I/O Kit methods and functions and for specific device families.
- Mac OS X Man Pages provides access to existing reference documentation for BSD and POSIX functions and tools in a convenient HTML format.

HBA Devices and I/O Processing

A host bus adapter (HBA) is a device that provides a connection between a set of peripheral storage devices and a computer or between nodes on a network. In addition, an HBA device may provide some I/O processing—such as segmentation and reassembly, flow control, error detection, and interrupt handling—that improves I/O throughput.

This chapter describes how HBA devices are represented and managed in Mac OS X and it introduces the SCSI Parallel family, which provides support for HBA drivers. This chapter also describes the path of an I/O request and how an HBA device driver participates in that journey.

If you need to develop a custom driver for an HBA device, you should read this chapter to learn how your driver fits into the system and how to take advantage of the services provided by the SCSI Parallel family. When you're ready to begin development, read "Developing an HBA Driver" (page 15) for specific guidelines on how to implement your driver and "Improving Performance" (page 25) for ways to improve its performance.

HBA Device Drivers in Mac OS X

Storage HBA devices provide an interface to sets of parallel SCSI, Fibre Channel, Serial ATA (SATA), or Serial Attached SCSI (SAS) devices. Some HBAs have firmware that allows them to communicate with more than one type of attached device. For example, some HBAs can handle both SAS and SATA drives.

You can develop an HBA driver to run in Mac OS X for any type of HBA that understands the SCSI commands defined in the command sets SPC-3 (SCSI Primary Commands-3), SBC-2 (SCSI Block Commands-2), MMC-3 (Multimedia Command Set-3), and RBC (Reduced Block Command Set). This includes most Fibre Channel HBA devices, in addition to parallel SCSI and SAS HBAs. For more information about the SCSI Parallel Interconnect standards, visit the SCSI Trade Association website at www.scsita.org.

An HBA Device Driver in the Mass Storage Driver Stack

In Mac OS X, HBA drivers participate in the mass storage driver stack as part of the physical interconnect layer (for more information about the objects in other layers of this stack, see *Mass Storage Device Driver Programming Guide*). The mass storage driver stack groups device communication into logical layers, each of which contains drivers and other objects that implement the appropriate protocols. In general, Apple-provided drivers and other objects implement system services, such as hot-plugging support and event handling, allowing third-party drivers to focus on supporting features unique to specific devices. A custom HBA driver's primary task is to transmit commands to the hardware, but it might also implement hardware-specific interrupt handling or device-management tasks.

Figure 1-1 shows the objects and layers in an example mass storage driver stack built up from a custom HBA driver.

Figure 1-1   A custom HBA driver participates in the mass storage driver stack
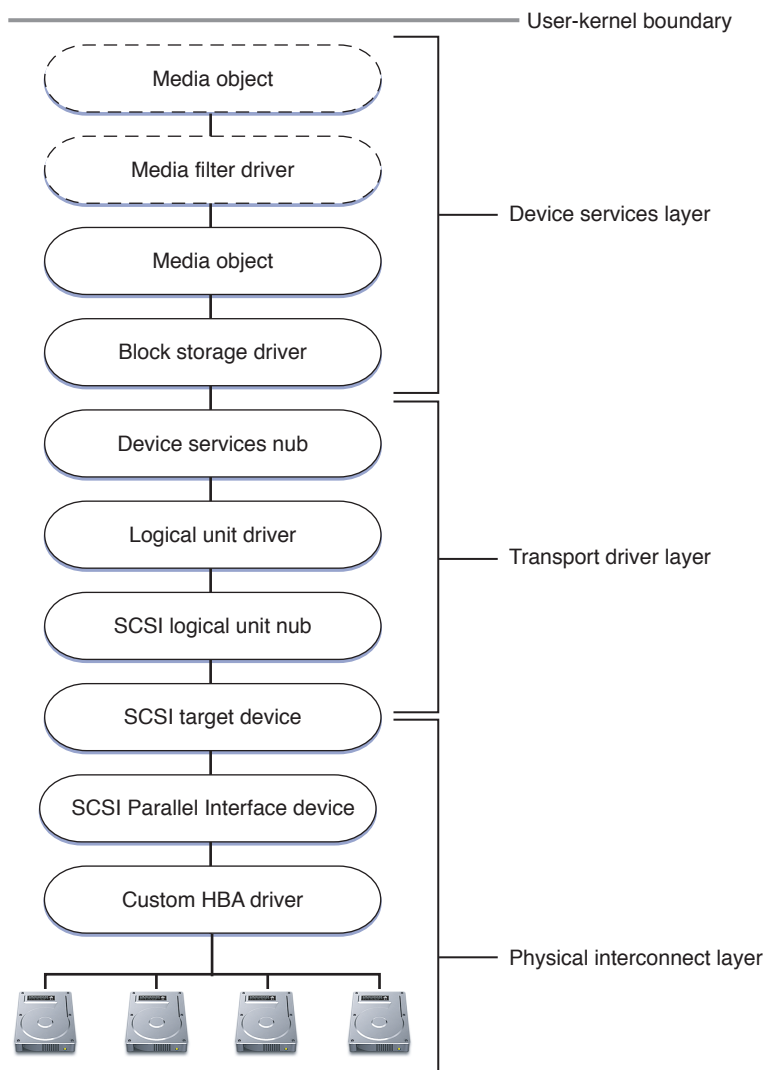
When the I/O Kit performs device matching on the object representing the HBA device, it instantiates a SCSI parallel interface controller object (an `IOSCSIParallelInterfaceController` instance). Depending on the individual HBA device, either the custom HBA driver or the SCSI parallel interface controller object then instantiates some number of SCSI parallel interface device objects (`IOSCSIParallelInterfaceDevice` instances). Although Figure 1-1 shows only one SCSI parallel interface device object, a real stack would contain a number of these objects equal to the number of devices the HBA device can support.

Each SCSI parallel interface device object then tries to create a SCSI target device object (an `IOSCSITargetDevice` instance). The SCSI target device object scans the device it represents for logical units and creates a SCSI logical unit nub (an `IOSCSILogicalUnitNub` instance) for each one it finds. Figure 1-1 shows the SCSI target device object straddling the logical boundary between the transport driver and physical interconnect layers. This is because it represents a link between the SCSI Architecture Model family (in which it is defined) and the SCSI parallel family (with which it communicates).

It is important to emphasize that neither the SCSI parallel interface controller object nor a custom HBA driver object creates SCSI target device objects. This allows a controller driver (Apple-provided or custom) to avoid having to rescan the entire bus for target devices. Instead, a rescan focuses on only those SCSI parallel interface device objects that do not have a SCSI target device object attached to them. Such a rescan does not interrupt I/O transfers in progress on SCSI parallel interface device objects that do have a SCSI target device object attached to them.

The appearance of a SCSI logical unit nub causes the I/O Kit to find and match to it either an in-kernel logical unit driver (for peripheral device types $00, $05, $07, and $0E) or a SCSI task user client (for other peripheral device types). The logical unit driver creates a SCSI task object (an instance of `SCSITask`) to contain the I/O request it receives from the objects above it in the stack. A SCSI task object contains the command descriptor block (or CDB) and is the fundamental unit of I/O transactions in the transport driver layer.

Objects in the device services layer of the mass storage driver stack view devices as abstract storage spaces and have no knowledge of command sets or physical interconnect protocols. Media objects can represent a subset of a storage device (such as a disk partition) or a set of devices (such as the set controlled by an HBA device). Optional filter drivers can implement encryption or validation on the content the media objects represent. The generic block storage driver implements the open and close semantics for the device and handles tasks such as deblocking for unaligned transfers and polling for ejectable media.

The Design of the SCSI Parallel Family

The SCSI Parallel family, introduced in Mac OS X v10.2, was designed to replace the SCSI family (which was deprecated in Mac OS X v.10.3). In spite of its rather specific name, the SCSI Parallel family supports any HBA device that can present itself as a parallel SCSI HBA. Naturally, this includes serial-attached SCSI (SAS) HBA devices, but it also includes Fibre Channel and SATA HBAs, many of which use high-level SCSI APIs to handle I/O.

The SCSI Parallel family offers many advantages over the SCSI family, such as the elimination of the locking and synchronization problems that often plagued the older family. The SCSI Parallel family also emphasizes correct implementation of the SCSI parallel interface standards SPI-3 and SPI-4, which means that it handles the vast majority of conforming devices.

Of most interest to HBA driver developers, however, is the fact that the SCSI Parallel family implements most of the operating system–dependent and SPI-defined tasks required for an HBA driver to communicate with other objects in the mass storage driver stack. For example, the SCSI Parallel family handles bus scanning, device recovery, discovery of logical units, creation of target device objects, and construction of SCSI commands, among other things. This high level of support confers three significant advantages:

- It allows developers to focus on the HBA device driver's primary task of shuttling I/O to and from the hardware.

- It provides a consistent experience across physical interconnects.

- It insulates developers from potential changes in underlying implementations.

The SCSI Parallel family defines the `IOSCSIParallelInterfaceController` class, which is the base class for all custom HBA drivers that can communicate using SCSI commands. The `IOSCSIParallelInterfaceController` class handles most of the I/O Kit driver life-cycle functions, various memory-allocation tasks, and optional target-device management, among other things. As a subclass of this class, a custom HBA driver implements a few required methods to report information about itself, initialize itself, and process I/O requests. If necessary, a custom HBA driver can also implement methods to perform device-management tasks and device-specific interrupt handling.

The `IOSCSIParallelInterfaceController` class also provides a large number of methods that allow you to get information about other objects without needing to know anything about the structure of those objects. For example, there are methods to retrieve information about the SCSI parallel task objects that represent I/O requests and SCSI commands and methods to get information about SCSI target device objects. Using these methods is both convenient and safe; accessing such objects directly can lead to problems if their implementations change.

The Role of an HBA Device in the Journey of an I/O Request

One of the most important features of Mac OS X is that it is a multithreaded operating system. The advantages of multithreadedness are especially apparent in the performance of I/O. This section focuses on the roles of different threads in the processing of an I/O request. An in-depth description of multithreadedness and other ways it might affect your driver is beyond the scope of this document; if you'd like to learn more about it, see *Kernel Programming Guide*.

Figure 1-2 shows a simplified view of an I/O path. Refer to this figure as you read the description below it.

Figure 1-2   The journey of an I/O request
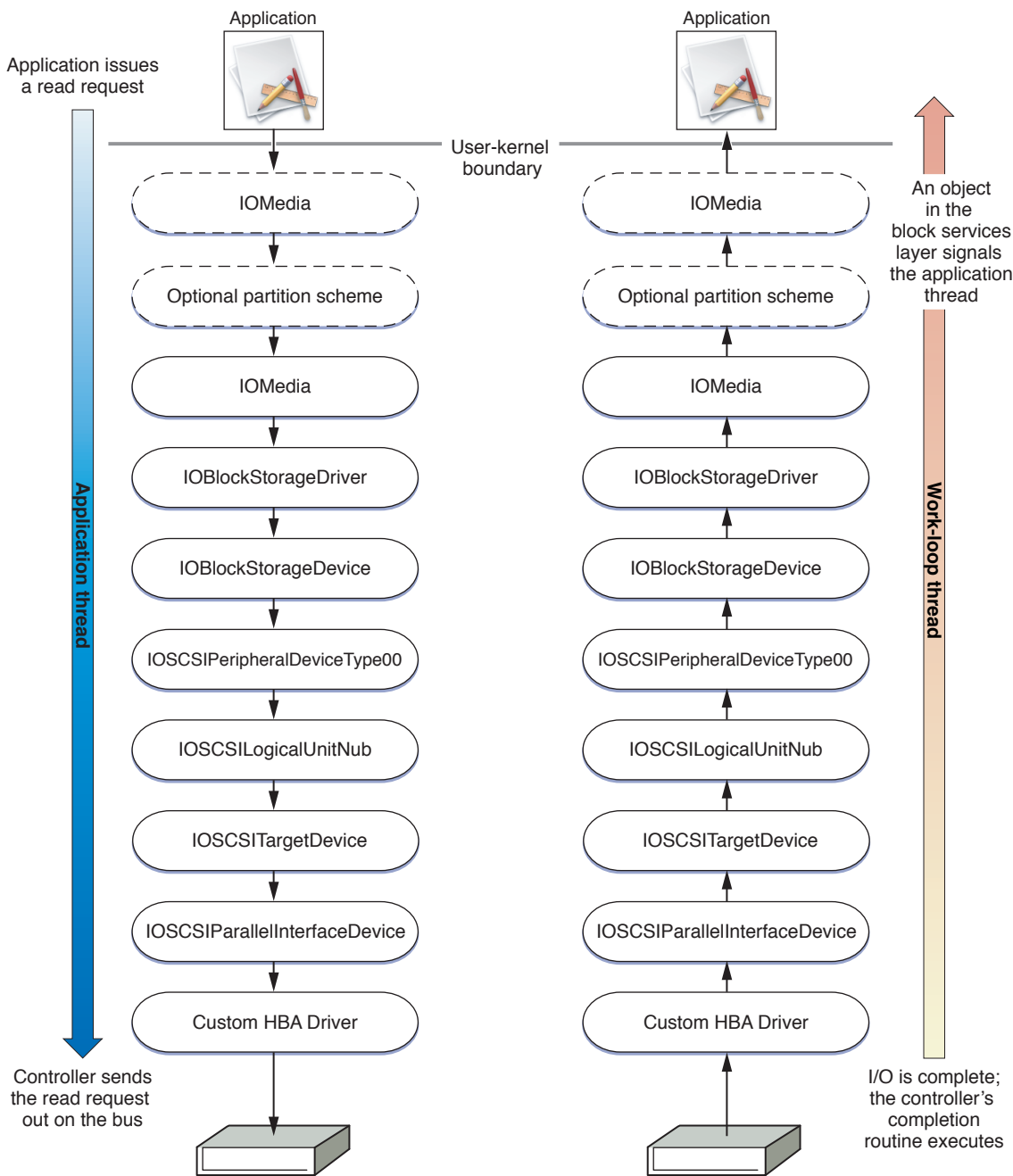
Imagine that an application makes a standard BSD `read` request and that the request (unbeknownst to the application) ultimately targets a SCSI storage device. The BSD command calls into the kernel, which performs a translation of the command's file descriptor to ensure that it targets the correct file system. After some additional handling, the read request transitions from the file system layer to the I/O Kit layer, where it enters the IOMedia object that represents the targeted content.

The request travels down the mass storage driver stack, each driver wrapping the request in the appropriate, protocol-defined format, and passing it to the driver below. Finally, the request reaches the HBA driver, which sends a command out on the peripheral bus to the appropriate target device. At this point, the application

thread blocks, waiting for the completion of its request. The exact location of the waiting thread depends on specific circumstances and is not crucial to this description. The important thing is that the read request travels on the application's thread, all the way to the HBA driver.

When the target device completes the request, it causes a hardware (direct) interrupt. The interrupt controller (running in the primary interrupt context) dispatches the interrupt to the HBA driver and signals its work loop to run. With this, the primary interrupt context goes away and the secondary interrupt context (in which the controller driver now runs) begins, with the lock held. The controller driver's read completion routine executes, which triggers a cascade of completion routines as the completed request travels back up the stack. Finally, an object in the block services layer, still on the work-loop thread with the lock held, calls a file-system completion routine that signals the application thread to wake up. The application thread returns to the application with the completed read request and the application resumes running.

The distinction between the two threads involved in the journey of an I/O request is an important one because it affects the way your HBA driver processes I/O requests and handles interrupts.

Developing an HBA Driver

This chapter describes how to develop a custom HBA driver.

If you've never developed in-kernel device driver for Mac OS X, be sure to read the documents listed in "Introduction to HBA Device Driver Programming Guide" (page 7) to learn about the I/O Kit and how to use Xcode to create a device driver.

Create a Matching Dictionary for Your Driver

To ensure that your driver matches and loads for your device, you need to edit your driver's information property list (`Info.plist` file). The I/O Kit uses the information in this file to match a device with the driver best suited to drive it. Because HBA hardware is represented by one or more `IOPCIDevice` nubs, a custom HBA driver must include the following key-value pair in its `Info.plist` file:

```
<key>IOProviderClass</key>
<string>IOPCIDevice</string>
```

If you're developing an HBA driver to run in a PowerPC-based Macintosh computer, you can use Open Firmware matching to match on some combination of the Open Firmware device properties `name`, `compatible`, `device_type`, or `model`. To do this, use the `IONameMatch` key and supply the appropriate property or properties in a string (or array of strings) as the value, as in this example:

```
<key>IONameMatch</key>
<string>pci1000,626</string>
```

If you're developing an HBA driver to run in an Intel-based Macintosh computer, use standard PCI matching (Open Firmware matching is not available in Intel-based Macintosh computers). PCI matching is based on the values in the PCI configuration space registers listed below (note that some registers, such as revision ID and header type, are not available for matching):

- Vendor and device ID (offsets $00 and $02)
- Subsystem vendor and device ID (offsets $2C and $2E)
- Class code (offset $09)

The PCI family defines four matching keys that you can use to specify combinations of values on which to match. The table below shows these keys and their associated behavior.

Key	Matching behavior
IOPCIMatch	Matches against the primary vendor and device ID registers or the subsystem vendor and device ID registers. The values of the primary registers are checked first; if there is no match, then the values of the subsystem registers are checked.

Key	Matching behavior
IOPCIPrimaryMatch	Matches against only the primary vendor and device ID registers.
IOPCISecondaryMatch	Matches against only the subsystem vendor and device ID registers.
IOPCIClassMatch	Matches against the class code register.

For several examples of PCI matching, see the "PCI Matching" section in Writing a Driver for a PCI Device. For information on other matching keys, such as `kIOPropertyPhysicalInterconnectLocationKey`, see the documentation for `IOStorageProtocolCharacteristics.h` in *Kernel Framework Reference*.

After your driver is matched to your device and started, you have the opportunity to add to the property list other properties that contain information about the HBA device itself or about its target devices. These properties (like all properties in the `Info.plist` file) are stored in the I/O Registry during the driver's lifetime. This allows other I/O Kit entities to get information about the capabilities of your device. The `IOSCSIParallelInterfaceController` class provides methods your driver can call to add or remove properties (these methods are described in "Set and Remove HBA and Target Device Properties (Optional)" (page 20)). It's important to use only these methods to add or remove properties because they protect you from the synchronization issues that can arise when different entities attempt to make changes in the I/O Registry at the same time.

Implement an IOSCSIParallelInterfaceController Subclass

The SCSI Parallel family's `IOSCSIParallelInterfaceController` class contains all the methods you need to develop a custom HBA driver. It's important to emphasize that the SCSI Parallel family handles a great many lower-level details for you, such as the creation of a work loop and various event sources, basic interrupt handling, and, if necessary, the creation and destruction of target device objects.

For the most part, your job consists of subclassing the `IOSCSIParallelInterfaceController` class and implementing its pure virtual methods to process I/O and report information specific to your hardware. The `IOSCSIParallelInterfaceController` class also contains methods you can implement to perform tasks, such as managing target devices and handling interrupts, and methods you can call to get information and set various properties. The following sections describe the required and optional tasks HBA drivers perform and the `IOSCSIParallelInterfaceController` methods associated with them.

After you've succeeded in developing a driver that communicates with your device, be sure to read "Improving Performance" (page 25) to learn about ways to improve your driver's performance in Mac OS X.

Initialize Your HBA Device Driver (Required)

Custom HBA drivers do not implement the I/O Kit `init` method. Instead, the `IOSCSIParallelInterfaceController` object does some initial setup work (such as getting a work loop and updating I/O Registry properties) and then it calls the `InitializeController` method on your subclass. In your implementation of this method, you should perform any required hardware initialization, resource allocation, and hardware interrogation.

As you design the implementation of the `InitializeController` method, be aware that you do not need to allocate memory in your subclass to hold per-command or per-target information. For example, some HBA devices keep track of the mapping between the worldwide node name and the target device or information about negotiation speeds. Instead of allocating memory to hold such information in your `InitializeController` method, you should use the report methods to tell the `IOSCSIParallelInterfaceController` superclass how much space you need. Then, when the `IOSCSIParallelInterfaceController` object creates target-device and task objects, it will also allocate enough memory to hold the information you need to keep. (For more information on the report methods, see "Report Device-Specific Information (Required)" (page 17).)

If there is information you need to store that is specific to the HBA device itself, however, you can allocate memory for it in the `InitializeController` method. Your driver might also need to allocate a small amount of memory to use for non-task communication with target devices. For example, some HBA drivers send to target devices auxiliary requests or negotiation requests that can't be encapsulated in I/O request objects.

In your implementation of the `InitializeController` method you should also query the hardware to get information you need, such as the highest supported device ID and the initiator ID. Your driver might need other data as well, but it will be required to report the initiator ID and the highest supported device ID and logical unit number (LUN) to its superclass (see "Report Device-Specific Information (Required)" (page 17) for the methods you use to supply this data).

Report Device-Specific Information (Required)

After you've initialized your HBA device, the `IOSCSIParallelInterfaceController` object requests some information from your driver. To provide this information, your driver must implement the following methods:

- `ReportInitiatorIdentifier`

  The `IOSCSIParallelInterfaceController` object calls this method to get the SCSI identifier of the initiator (your HBA device). The initiator ID is considered reserved and no target devices can be created with that ID.

- `ReportHighestSupportedDeviceID`

  In this method, your HBA driver must return the highest device ID it supports. The `IOSCSIParallelInterfaceController` object uses the value you return to determine the last ID to process when, for example, it scans for target devices.

- `ReportMaximumTaskCount`

  In this method, return the maximum number of outstanding tasks your HBA device can process. The `IOSCSIParallelInterfaceController` object uses this value both to allocate the appropriate number of task objects and to avoid sending a new task when your driver is already processing the maximum number of tasks.

  > **Important:** You must report a number greater than 0 in this method or your driver will not match and load. However, the number you report should balance your driver's performance with overall system performance. For some advice on how to arrive at a reasonable value for this quantity, see "Report Appropriate Maximum Task Count" (page 26).

- `ReportHBASpecificTaskDataSize`

If your HBA driver needs to keep track of some data for each task it processes, it can tell its superclass to allocate space for that data in each task object. This relieves your driver of the responsibility of creating and maintaining the memory to hold per-task data, such as a scatter-gather list.

- `ReportHBASpecificDeviceDataSize`

  If your HBA driver needs to keep track of some data for each target device attached to it, it can tell its superclass to allocate space for that data in each `IOSCSIParallelInterfaceDevice` object. Your driver might want to store data such as negotiation speeds, the worldwide port name, or a destination ID.

- `DoesHBAPerformDeviceManagement`

  Many HBAs, typically Fibre Channel and SAS HBAs, are capable of discovering and managing the devices attached to them. Other HBAs, such as generic parallel SCSI HBAs, know how many devices they can support, but do not manage those devices in any way. All HBA drivers must implement the `DoesHBAPerformDeviceManagement` method to state whether they intend to manage the target devices attached to them.

  If your HBA driver returns `true` in this method, it means that target device objects will be created (and destroyed) only by your HBA driver, not by the superclass. Your HBA driver must then call the `IOSCSIParallelInterfaceController` methods that create and destroy target device objects (described in "Create, Initialize, and Destroy Target Device Objects (Optional)" (page 18)).

  If your HBA does *not* perform device management, you should return `false` in the `DoesHBAPerformDeviceManagement` method, and you do not have to call the methods described in "Create, Initialize, and Destroy Target Device Objects (Optional)" (page 18). For these HBAs, the `IOSCSIParallelInterfaceController` takes responsibility for discovering and probing attached devices at boot time. The superclass creates the target device objects and attaches them to the custom HBA driver object in the I/O Registry.

- `ReportHBAHighestLogicalUnitNumber`

  In this method, your driver should return the highest logical unit number your HBA device can address.

- `DoesHBASupportSCSIParallelFeature`

  The superclass calls this method to find out if your HBA device supports a specific SCSI Parallel Interface feature (as defined in the SPI-3 and SPI-4 specifications). Your implementation of this method should examine the specified feature and return `true` if you support it or `false` if you don't.

Create, Initialize, and Destroy Target Device Objects (Optional)

A driver for an HBA device that manages the target devices attached to it must create and destroy the I/O Kit objects that represent them in the mass storage driver stack. (If your driver manages target devices, be sure to return `true` in your implementation of the required `DoesHBAPerformDeviceManagement` method, described in "Report Device-Specific Information (Required)" (page 17).) If your driver does not manage target devices, you can skip the rest of this section and go on to the next section, "Set and Remove HBA and Target Device Properties (Optional)" (page 20).

To create target device objects, your driver must call one of the variants of the `CreateTargetForID` method. You cannot override the `CreateTargetForID` methods because the `IOSCSIParallelInterfaceController` object uses them to perform many low-level tasks, such as updating internal states and structures.

The `IOSCSIParallelInterfaceController` class allows a custom HBA driver to create a target device object by passing either the device ID of the target device and a dictionary of its properties or just the device ID. In general, Fibre Channel HBA devices need to supply a dictionary of properties that describe the target device, but most other HBA devices do not. Table 2-1 (page 19) lists the property keys and values that Fibre Channel HBA devices can provide in the `CreateTargetForID` method. For more information about these and other keys used to describe the characteristics of physical interconnect protocols, see the documentation for `IOStorageProtocolCharacteristics.h` in *Kernel Framework Reference*.

Table 2-1   Property keys and values for Fibre Channel device objects

Property key	Required	Value	Type
`kIOPropertyFibre-ChannelNodeWorldWideNameKey`	Yes	Unique 64-bit worldwide name for the device node	`OSData`
`kIOPropertyFibre-ChannelPortWorldWideNameKey`	No	Unique 64-bit worldwide name for the port	`OSData`
`kIOPropertyFibreChannelAddress-IdentifierKey`	No	The 24-bit address identifier as defined in the Fibre Channel FS (FC-FS) specification	`OSData`
`kIOPropertyFibreChannelALPAKey`	No	The 8-bit Arbitrated Loop Physical Address value	`OSData`

> **Note:** You can also use the `SetTargetProperty` method (described in "Set and Remove HBA and Target Device Properties (Optional)" (page 20)) to add the optional key-value pairs to the device's property dictionary after it is created.

When you call the `CreateTargetForID` method, the `IOSCSIParallelInterfaceController` object creates a new `IOSCSIParallelInterfaceDevice` object to represent the device. This device object performs the I/O Kit `init`, `attach`, and `start` routines. In the mass storage driver stack, the `IOSCSIParallelInterfaceDevice` object transitions between objects in the SCSI Parallel family and objects in the SCSI Architecture Model family.

Because device-specific data is stored in the target device object, the memory for this data is not allocated until the target device object is created. If your HBA driver needs to access that memory to perform set-up tasks for a target object it creates, it should implement the `InitializeTargetForID` method. This method is called by the `IOSCSIParallelInterfaceDevice` object in its `start` method (before the object is registered for matching), when the target device–specific memory is available.

If your driver calls the `CreateTargetForID` method to create target device objects, it must also call the `DestroyTargetForID` method to destroy them. As with the `CreateTargetForID` method, you cannot override the `DestroyTargetForID` method; otherwise you might interfere with the tasks the `IOSCSIParallelInterfaceDevice` object performs in its implementation of the method. When you call the `DestroyTargetForID` method, the specified `IOSCSIParallelInterfaceDevice` object terminates.

Start Your HBA Driver (Required)

To ready your HBA device to accept I/O requests and SCSI commands, you must implement the `StartController` method. The superclass calls the `StartController` method in its own `start` method, after it has retrieved HBA-specific information from your driver and allocated a pool of SCSI parallel task objects.

In your `StartController` method, you might need to enable ports on your hardware, clear pending interrupts, and perform other tasks that put your device in a running state. If your HBA performs device management, you might also scan for target devices in the `StartController` method. If you do this, be aware that the superclass is holding a command gate while in its `start` method. You might find that this prevents you from receiving asynchronous, interrupt-based device-discovery notifications. If you experience this, you can create a separate thread on which to call `CreateTargetForID`.

The most important thing to remember about the `StartController` method is that your driver must be prepared to receive and process I/O requests and commands as soon as the method completes.

Set and Remove HBA and Target Device Properties (Optional)

During the initialization and starting phases of your driver, you might need to set or remove I/O Registry properties associated with target devices or with the HBA device itself. Instead of using `IORegistryEntry` methods to do this directly, you should use the methods the `IOSCSIParallelInterfaceController` class provides. One advantage of using the family-specific methods is that they perform some data validation on the properties you pass in. A more important advantage is that these methods protect you from serialization problems that can occur when different entities attempt to manipulate properties in the I/O Registry at the same time.

Both the `SetTargetProperty` and `SetHBAProperty` methods expect a string containing the property key name and an `OSObject`-derived object containing the value. Some of the valid target property keys are listed in Table 2-1 (page 19); a list of valid HBA property keys is in `SetHBAProperty`. For more information about the values appropriate for these properties, see the header files `IOStorageProtocolCharacteristics.h` and `IOStorageDeviceCharacteristics.h`.

> **Note:** You can pass a property's value in the appropriate `OSString`, `OSData`, or `OSNumber` container objects; `OSDictionary` and `OSArray` objects are not allowed because they can cause serialization problems.

Process Parallel Tasks (Required)

The primary task of a custom HBA driver is to receive SCSI parallel tasks and pass them along to the target devices. To do this your HBA subclass must implement the `ProcessParallelTask` method.

In the `ProcessParallelTask` method, your HBA driver can call `IOSCSIParallelInterfaceController` accessor methods to extract information from the `SCSIParallelTask` object that represents the task. Some of these accessor methods are listed here:

```
GetCommandDescriptorBlock
GetDataBuffer
GetHBADataPointer
GetSCSIParallelFeatureNegotiationCount
```

> **Important:** You should always use the `IOSCSIParallelInterfaceController` accessor methods to extract information from the SCSI task object (and all other objects in the stack). This preserves the opacity of these objects and protects you from any changes that might be made to their internal implementations.

After you've examined the information in the command and processed it as required, you need to send the command out on the bus. Some HBA devices require strict synchronization between sending commands out on the bus and processing the interrupts generated from completed commands; others do not. The `IOSCSIParallelInterfaceController` class supports both types of HBA device by providing mechanisms an HBA driver can use to ensure the level of synchronization its hardware needs. For more information on how to synchronize these events, see "Synchronize Hardware Access" (page 25).

You must return a valid SCSI service response value from your implementation of the `ProcessParallelTask` method. The following is a list of the SCSI service responses (defined in the header file `SCSITask.h` in the Kernel framework) you can return:

- `kSCSIServiceResponse_Request_In_Process`

    Although this is not one of the standard service responses defined in the SCSI Architecture Model specification, it is the response you should return when an asynchronous command is not yet completed.

- `kSCSIServiceResponse_SERVICE_DELIVERY_OR_TARGET_FAILURE`

- `kSCSIServiceResponse_TASK_COMPLETE`

- `kSCSIServiceResponse_LINK_COMMAND_COMPLETE`

- `kSCSIServiceResponse_FUNCTION_COMPLETE`

- `kSCSIServiceResponse_FUNCTION_REJECTED`

To learn how to handle timeouts, see "Handle Timeouts (Optional)" (page 22).

Perform SCSI Task Management (Required)

Clients of your HBA driver use the SCSI task management functions (defined in the SCSI Architecture Model 2 specification) to request specific actions. The `IOSCSIParallelInterfaceController` class defines the following pure virtual methods your subclass must implement to perform these actions:

```
AbortTaskRequest
AbortTaskSetRequest
ClearACARequest
ClearTaskSetRequest
LogicalUnitResetRequest
TargetResetRequest
```

In your implementation of these methods, you can complete them either synchronously or asynchronously. If you choose to complete them synchronously, your implementation should return the appropriate SCSI service response. You can return any of the service responses listed at the end of "Handle Parallel Tasks (Required)" (page 20) *except* `kSCSIServiceResponse_Request_In_Process`.

If you decide to complete these functions asynchronously, your implementation should return only the `kSCSIServiceResponse_Request_In_Process` service response and then call the appropriate completion method. The `IOSCSIParallelInterfaceController` class defines the following completion methods your driver should implement if it processes SCSI task management functions asynchronously:

```
CompleteAbortTask
CompleteAbortTaskSet
CompleteClearACA
CompleteClearTaskSet
CompleteLogicalUnitReset
CompleteTargetReset
```

Because both command completion and timeout management are serialized on the work loop thread, you should serialize the task management your driver performs on the work loop, too. One way to do this is to use a command gate object associated with the work loop to take the work-loop lock before you send the task management request to the target device (you can accomplish this using the `IOCommandGate::runAction` function). In this way, you can avoid receiving a command or task management completion at the same time you send a task management request. The `IOSCSIParallelInterfaceController` class provides the convenient accessor method `GetCommandGate` to get the command gate object associated with the work loop.

> **Note:** There are no timers enabled on task management requests. It is up to your HBA driver to deal with failed task management requests and notify the requesting client of the failure by returning `kSCSIServiceResponse_SERVICE_DELIVERY_OR_TARGET_FAILURE`.

Handle Timeouts (Recommended)

SCSI commands usually have a timeout value associated with them. Your driver does not set the command's timeout value (it is set by the object that sent you the command), but it can enable the timer by calling the `SetTimeoutForTask` method. (If you choose, you can also use the `SetTimeoutForTask` method to override the timeout value provided in the task.)

Although some HBA drivers may not need to do anything when a command times out, many drivers need to perform some cleanup and communicate with the hardware. If your driver needs to handle command timeouts in a hardware-specific way, you can implement the `HandleTimeout` method.

When a task times out, the SCSI Parallel family calls the `HandleTimeout` method. The `IOSCSIParallelInterfaceController` class's default implementation of this method merely completes the command with a SCSI task status of `kSCSITaskStatus_TaskTimeoutOccurred` and a SCSI service response of `kSCSIServiceResponse_SERVICE_DELIVERY_OR_TARGET_FAILURE`. If this does not meet your driver's needs, you can implement the `HandleTimeout` method to, for example, clean up HBA-specific data structures associated with the command and abort the command. If the command is already out on the bus, you may have to send an `AbortTaskRequest` command; if not, you may be able to simply remove it from the queue of commands waiting to be sent to the target device.

> **Note:** The `HandleTimeout` method is called on the work loop with the lock already held, so you do not have to get the lock in this method.

Handle Interrupt Requests (Required)

As described in "The Role of an HBA Device in the Journey of an I/O Request" (page 12), the completion of an I/O request triggers an interrupt that an HBA driver must handle. When the `IOSCSIParallelInterfaceController` object is instantiated, it creates a work loop and registers with it a filter interrupt event source (among other event sources). The interrupt event source includes an action routine that runs on the work-loop thread to handle interrupts. When an interrupt occurs, the action routine calls your HBA driver's `HandleInterruptRequest` method while in the secondary interrupt context (on the work-loop thread with the lock held). Your driver must implement the `HandleInterruptRequest` method to service the interrupt.

Your implementation of the `HandleInterruptRequest` method should be as efficient as possible and never block indefinitely. In particular, this means that you must not allocate memory or create objects in your `HandleInterruptRequest` method (or any methods it calls), because allocation can block for unrestricted periods of time. Be sure to call the `CompleteParallelTask` method when your driver completes the processing of a parallel task.

Stop and Terminate Your HBA Driver (Required)

In your implementation of this required method you should stop accepting commands, but you should not release any resources you may have acquired. This is because you may be called upon to resume accepting commands (with a call to your `StartController` method) at any time.

When the `IOSCSIParallelInterfaceController` object calls your `TerminateController` method, however, it means your subclass object will be destroyed. You must implement this method to shut down all hardware services and release all resources you've acquired.

Improving Performance

This chapter describes a few things you can do to enhance the performance of your custom HBA device driver in Mac OS X. You should read this chapter if your hardware has any of the following features:

■ Serialization requirements for accessing registers

■ An optimum I/O transaction size or a maximum number of tasks it can handle

■ Support for interrupt coalescing

■ A shared interrupt line

Minimize Hardware-Access Synchronization

In "The Journey of an I/O Request Through the Mass Storage Stack" (page 12), you learned how an I/O request travels down the mass storage driver stack on the client's thread and how it is completed later by your HBA driver on its work-loop thread. In this model, it's possible for one I/O request to come into your driver at the same time another I/O request is completing. When this is the case, an HBA driver may find itself sending a command out on the bus at the same time it is attempting to handle an incoming interrupt. For some hardware, this situation can cause synchronization problems.

Some HBA devices require special synchronization because, for example, they do not allow simultaneous access to their registers. If your hardware has such a requirement, be sure to synchronize the sending and completing of I/O requests on your driver's work loop.

As described in "Handle Interrupt Requests (Required)" (page 23), the `IOSCSIParallelInterfaceController` object calls your `HandleInterruptRequest` method in the secondary interrupt context, with the work-loop lock held. To synchronize the issuing of a command with your interrupt handling, therefore, you need to execute the most sensitive code with the lock held. One way to do this is to put the sensitive, register-access code in a separate method and pass it to a command-gate object to run in its `runAction` routine. (For more information on the `IOCommandGate::runAction` function, see the documentation for `IOCommandGate` in *Kernel Framework Reference*.) Because a command gate object takes the work-loop lock before running its action routine, no other event sources on the same work loop (such as an interrupt event source) can run at the same time. Note that you do not need to create your own command gate. Instead, you can use the `GetCommandGate` accessor method to get a pointer to the command gate created by the `IOSCSIParallelInterfaceController` object.

If you do need to synchronize these tasks, be sure to streamline all code that executes while a lock is held. In particular, you should never allocate memory or create objects while the command gate holds the lock, because these tasks may block. The gated code should do as little work as possible to reduce potential contention on the work loop and to enhance the performance of your driver.

Report Appropriate Transaction Size

To achieve the best performance, you should specify the maximum I/O transaction size your hardware can handle. Given this information, the system can send to your hardware I/O requests of the appropriate size and this can increase your driver's throughput.

Information about maximum I/O transaction size is static and belongs in the I/O Registry, where other I/O Kit objects, such as other entities in the mass storage driver stack, can find it. The `IOKitKeys.h` header file in the Kernel framework defines eight keys you can use to add this information to your driver's `Info.plist` file (each key requires a value of type `OSNumber`). Table 3-1 lists these keys and whether they are required.

Table 3-1   I/O transaction-size keys

Key	Required	Description
`IOMaximumSegmentCountRead`	Yes	Maximum number of physically disjoint (non-contiguous) segments that can be processed on a per read I/O basis.
`IOMaximumSegment-` `CountWrite`	Yes	Maximum number of physically disjoint (non-contiguous) segments that can be processed on a per write I/O basis.
`IOMaximumSegment-` `ByteCountRead`	Yes	Maximum size in bytes for each physically disjoint (non-contiguous) segment.
`IOMaximumSegment-` `ByteCountWrite`	Yes	Maximum size in bytes for each physically disjoint (non-contiguous) segment.
`IOMaximumByteCountRead`	No	For most devices, this value is equal to the product of the values of the `IOMaximumSegmentByteCountRead` and `IOMaximumSegmentCountRead` keys.
`IOMaximumByteCountWrite`	No	For most devices, this value is equal to the product of the values of the `IOMaximumSegmentByteCountWrite` and `IOMaximumSegmentCountWrite` keys.
`IOMaximumBlockCountRead`	No	If your hardware does not require this value, do not use this key.
`IOMaximumBlockCountWrite`	No	If your hardware does not require this value, do not use this key.

Report Appropriate Maximum Task Count

To achieve the best overall system performance while your HBA driver is running, you should report a reasonable maximum task count in the `ReportMaximumTaskCount` method. As described in "Report Device-Specific Information (Required)" (page 17), you use this method to report the maximum number of outstanding tasks your HBA device can process. Although you should never return a number greater than your device's supported maximum task count (if it defines one), you may be able to improve system performance by reporting a number that balances the performance of your driver with that of the system as a whole.

The `IOSCSIParallelInterfaceController` class uses the number you return in the `ReportMaximumTaskCount` method to allocate an appropriate number of `SCSIParallelTask` objects. It's a good idea to run I/O tests on your HBA driver to find out if this number of preallocated objects is too large (making your driver's footprint larger than necessary and degrading system performance) or too small (causing your driver to block on I/O completions too often and degrading driver performance). Because overall system performance affects your driver's users, it's recommended that you find an optimum maximum task count value that allows your driver to be a good citizen in Mac OS X.

Turn Off Interrupt Coalescing

On other platforms, you may be accustomed to performing interrupt coalescing in your HBA driver. If, for example, your driver tends to perform numerous very small I/O transactions, it might make sense to wait for several completion interrupts instead of processing each completion as it occurs. Although batch-processing of I/O completions can increase the number of I/O operations per second, it can degrade I/O throughput. This may be a trade-off you're willing to accept, but you should be aware of the advantages and disadvantages of interrupt coalescing in Mac OS X.

Mac OS X is more likely to process I/O transactions synchronously than asynchronously. When the system does process I/O transactions asynchronously, it is more likely to process large transactions (for example, a megabyte in size), which are not good candidates for batch processing. Although it may seem counterintuitive, it's usually better to avoid interrupt coalescing in your Mac OS X HBA driver and instead handle I/O transactions separately. As with all performance decisions, however, you should test this with your specific hardware and driver and design accordingly.

Handle Shared Interrupts

It's important not to assume that your device has its own, dedicated interrupt line. For example, an expansion chassis may have a shared interrupt line and sometimes two PCI card slots can share an interrupt line. Another common example is an HBA card that is a multifunction PCI card. This type of card may have only one chip on it, but that chip can have multiple functions (also called devices). When the I/O Kit discovers such hardware, it creates an independent `IOPCIDevice` object to represent each function in the I/O Registry. Each of these `IOPCIDevice` objects is matched by a separate `IOSCSIParallelInterfaceDevice` object (and, if one exists, a separate instance of a custom HBA driver). Although each function has its own driver, these functions might share a single interrupt line on the card itself. When this is the case, more than one function can assert the shared interrupt line when it needs work to be done. This causes the interrupt controller to call each driver's interrupt-handling routine in turn to find the right one to handle the interrupt.

The I/O Kit provides the `IOFilterInterruptEventSource` class (a subclass of `IOInterruptEventSource`) to allow drivers to handle shared interrupts in a safe way. In addition to the action completion routine used to handle the interrupt, the `IOFilterInterruptEventSource` class defines a callback function that gets called in each driver sharing the interrupt line. Each driver implements this function as a filter, checking to see if the interrupt is indeed for it and responding accordingly. If a driver returns `true` in its filter routine, the I/O Kit automatically starts that driver's interrupt handler on its work loop and the interrupt remains disabled in hardware until the interrupt handler returns.

When the interrupt controller disables the interrupt line after finding and scheduling the correct driver's interrupt handling routine, no other device sharing that line can get its work done until the line is re-enabled. This can cause large interrupt latencies. You may be able to reduce these latencies if your hardware allows a driver to disable interrupts at the source (in other words, on the card).

The `IOSCSIParallelInterfaceController` class defines the `FilterInterruptRequest` method to allow the drivers of such hardware to disable interrupts and schedule their own interrupt handlers. If your hardware does not allow a driver to disable interrupts at the source, you should not implement the `FilterInterruptRequest` method. Instead, you should rely on the default implementation of this method, which returns `true`. This causes the interrupt controller to schedule the appropriate interrupt handler on the work loop and disable the interrupt line, as described above.

> ⚠️ **Warning:** The `FilterInterruptRequest` method is called at direct (primary) interrupt time. It is essential that your implementation of this method be as efficient as possible and that you defer all interrupt handling to your `HandleInterruptRequest` method. Also, you must be absolutely certain to clear the interrupting condition from the hardware; if you do not, an infinite loop of filter-interrupt requests will occur.

In your implementation of the `FilterInterruptRequest` method, you must first determine if the interrupt is for you. If it is not for you, you must allow the interrupt controller to call other drivers and allow them the chance to handle the interrupt. In this case, your implementation should immediately return `false`.

If the interrupt is for you, you must do the following in your `FilterInterruptRequest` method:

1. Disable interrupts for the device.

2. Call the `SignalInterrupt` method. This method schedules your driver's `HandleInterruptRequest` method on the work loop without disabling the interrupt line.

3. Return `false`.

Then, in your `HandleInterruptRequest` method, you must:

1. Clear the hardware condition that raised the interrupt.

2. Process the interrupt and complete the I/O request.

3. Re-enable interrupts for the device.

Document Revision History

This table describes the changes to *HBA Device Driver Programming Guide*.

Date	Notes
2006-05-23	New document that describes how to develop an in-kernel HBA driver.