Mass Storage Device Driver Programming Guide

Drivers, Kernel, & Hardware: Kernel Device Drivers

2007-04-03

Apple Inc.
© 2002, 2007 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, FireWire, Logic, Mac, Mac OS, Macintosh, Pages, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Finder is a trademark of Apple Inc.

NeXT is a trademark of NeXT Software, Inc., registered in the United States and other countries.

CDB is a trademark of Third Eye Software, Inc.

Intel and Intel Core are registered trademarks of Intel Corportation or its subsidiaries in the United States and other countries.

PowerPC and and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction **Introduction to Mass Storage Device Driver Programming Guide 7**

Who Should Read This Document? 7
Organization of This Document 7
See Also 8

Chapter 1 **Mass Storage Overview 11**

Mass Storage Devices in Mac OS X 11
Mass Storage Drivers 12
The Mass Storage Driver Stack 12
    The Transport Driver Layer 14
    The Device Services Layer 15
Mass Storage Stack Implementation 17
    Mass Storage Driver Objects 17
    The SCSI Architecture Model Family 18
    The Storage Family 21
Construction of a Mass Storage Driver Stack 24

Chapter 2 **Mass Storage Device Compliance 29**

Device Compliance 29
Available Mass Storage Drivers 29

Chapter 3 **Mass Storage Driver Matching and Loading 31**

Driver Personalities and the Matching Process 31
    Driver Personalities 31
    Driver Matching 32
    Driver Starting 33
Protocol Services Driver Matching 33
    The FireWire SBP-2 Protocol Services Driver 33
    The USB Mass Storage Class Protocol Services Driver 36
    The ATAPI Protocol Services Driver 39
Logical Unit Driver Matching 41
Filter-Scheme Driver Matching 43

Chapter 4 **Developing a Universal Binary 47**

Creating a Logical Unit or Protocol Services Driver Universal Binary 47
Creating a Filter Scheme Universal Binary 48

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Chapter 5 **Subclassing Logical Unit Drivers 49**

Setting Up Your Project 49
  Create a New Project 49
  Edit Your Driver's Property List 50
Creating Your Driver 51
  Edit the Header File 51
  Edit the C++ File 52
Testing Your Driver 53
Creating and Sending SCSI Commands 53

Chapter 6 **Subclassing Protocol Services Drivers 59**

Setting Up Your Project 59
  Create a New Project 59
  Edit Your Driver's Property List 60
Creating Your Driver 62
  Edit the Header File 62
  Edit the C++ File 62
Testing Your Driver 64

Chapter 7 **Developing a Filter Scheme 65**

Edit Your Driver's Property List 65
Creating Your Filter Scheme 67
  Edit the Header File 67
  Edit the C++ File 69
Testing Your Filter Scheme 74

Document Revision History 77

Index 79

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Figures, Tables, and Listings

Chapter 1 Mass Storage Overview 11

Figure 1-1 The mass storage driver stack 13
Figure 1-2 The transport driver layer 14
Figure 1-3 The device services layer 16
Figure 1-4 SCSI Architecture Model family inheritance 19
Figure 1-5 Storage family inheritance 22
Figure 1-6 Subclasses inheriting from a base class 24
Figure 1-7 An example mass storage stack 25
Figure 1-8 Adding a subclassed logical unit driver 27
Figure 1-9 Adding an encryption scheme 28

Chapter 3 Mass Storage Driver Matching and Loading 31

Table 3-1 USB mass storage class subclasses 36
Table 3-2 USB mass storage class protocols 36
Table 3-3 Interface-matching keys from the USB Common Class Specification 39
Table 3-4 IOMedia properties 43
Listing 3-1 The IOFireWireSerialBusProtocolTransport driver personality dictionary 34
Listing 3-2 Example FireWire protocol services driver probe method 35
Listing 3-3 One of the IOUSBMassStorageClass driver's personalities 37
Listing 3-4 Partial listing of an Info.plist file for a vendor-specific device 38
Listing 3-5 The IOATAPIProtocolTransport driver personality dictionary 39
Listing 3-6 A personality dictionary that overrides DMA and UDMA values 40
Listing 3-7 The IOSCSIPeripheralDeviceType00 driver personality dictionary 42
Listing 3-8 Example logical unit driver personality dictionary 42
Listing 3-9 Example filter-scheme driver personality 44

Chapter 4 Developing a Universal Binary 47

Listing 4-1 Byte-swapping in IOSCSIBlockCommandsDevice code 47
Listing 4-2 Byte-swapping in IOApplePartitionScheme code 48

Chapter 5 Subclassing Logical Unit Drivers 49

Table 5-1 Personality properties for MyLogicalUnitDriver 50
Table 5-2 Dependencies for MyLogicalUnitDriver 51
Listing 5-1 The MyLogicalUnitDriver header file 52
Listing 5-2 The MyLogicalUnitDriver C++ file 52
Listing 5-3 Header file for a driver that sends standard and custom SCSI commands 53
Listing 5-4 Implementation of a driver that sends standard and custom SCSI commands 54

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Chapter 6 **Subclassing Protocol Services Drivers 59**

Table 6-1 Personality properties for MyFWProtocolServicesDriver 61
Table 6-2 Dependencies for MyFWProtocolServicesDriver 61
Listing 6-1 The MyFWProtocolServicesDriver header file 62
Listing 6-2 The MyFWProtocolServicesDriver C++ file 63

Chapter 7 **Developing a Filter Scheme 65**

Table 7-1 Personality properties for MyFilterScheme 66
Table 7-2 Dependencies for MyFilterScheme 66
Listing 7-1 The MyFilterScheme header file 67
Listing 7-2 The MyFilterScheme C++ file 70

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Introduction to Mass Storage Device Driver Programming Guide

Note: This document was previously titled *Writing Drivers for Mass Storage Devices*.

This document introduces the architecture of the mass storage driver stack and describes how to write in-kernel drivers for mass storage devices and media filter schemes for content on mass storage media. It includes sample code that illustrates how to develop both in-kernel logical unit and protocol services drivers and in-kernel filter-scheme drivers.

Because this book focuses on kernel-resident drivers for mass storage devices that mount file systems or are bootable, it provides only a brief description of application-based drivers for other mass storage devices, such as tape drives. For general information on how to write drivers for such devices, see *Accessing Hardware From Applications*.

Important: This book documents the mass storage features introduced in Mac OS X version 10.1. Updates to some of this information are identified with the version of Mac OS X in which they appeared.

Who Should Read This Document?

You should read this document if you need to support a mass storage device that mounts a file system or is bootable, or if you need to develop a filter-scheme driver.

Writing drivers for Mac OS X requires the I/O Kit, Apple's object-oriented framework for driver development. Although this document presents some information on selected I/O Kit principles to provide context for the implementation of the mass storage driver stack, it does not explain these concepts in detail. If you're not familiar with the I/O Kit, you should read *I/O Kit Fundamentals* before reading this document.

In addition, if you've never written an in-kernel device driver for Mac OS X, you should read *I/O Kit Device Driver Design Guidelines* to become familiar with driver fundamentals such as driver life cycle and driver matching and loading.

Organization of This Document

This document contains the following chapters:

- "Mass Storage Overview" (page 11) describes how Mac OS X supports mass storage devices and how the mass storage driver stack is built.
- "Mass Storage Device Compliance" (page 29) describes the various device specifications with which your device must comply to work with the built-in mass storage device drivers.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

- "Mass Storage Driver Matching and Loading" (page 31) describes the driver matching process for protocol services, logical unit, and filter-scheme drivers.
- "Developing a Universal Binary" (page 47) provides some tips for developing a universal binary version of a logical unit driver, a protocol services driver, and a filter-scheme driver.
- "Subclassing Logical Unit Drivers" (page 49) describes how to subclass a built-in logical unit driver to provide device-specific support.
- "Subclassing Protocol Services Drivers" (page 59) describes how to subclass a built-in protocol services driver to provide device-specific support.
- "Developing a Filter Scheme" (page 65) describes how to create and test a filter-scheme driver.
- "Document Revision History" (page 77) lists the revisions of this document.

See Also

The ADC Reference Library contains several documents on device driver development for Mac OS X and numerous sample drivers and applications.

- *Kernel Extension Programming Topics* contains tutorials that introduce you to the fundamental techniques you need to develop, debug, and package kernel extensions. This document also contains information on kernel extension loading and dependencies.
- *I/O Kit Fundamentals* describes the architecture of the I/O Kit, the object-oriented framework for developing Mac OS X device drivers.
- *I/O Kit Device Driver Design Guidelines* provides guidelines and tips for developing, debugging, and deploying kernel-resident device drivers.
- *Kernel Framework Reference* contains API reference for I/O Kit methods and functions and for specific families
- Sample Code > Hardware & Drivers > Storage includes both application-level and in-kernel code samples.
- Mac OS X Man Pages provides access to existing reference documentation for BSD and POSIX functions and tools in a convenient HTML format.
- The darwin-drivers mailing list provides a forum for discussing technical issues related to I/O Kit device driver development.

If you're ready to create a universal binary version of your device driver or filter scheme to run in an Intel-based Macintosh, see *Universal Binary Programming Guidelines, Second Edition*. The *Universal Binary Programming Guidelines* describes the differences between the Intel and PowerPC architectures and provides tips for developing a universal binary.

The Mac OS X mass storage stack supports mass storage devices that comply with the SCSI Architecture Model SCSI primary commands specification, declare peripheral device types of $00, $05, $07, or $0E, and connect to ATAPI, USB, or FireWire buses. In addition, a USB device must be compliant with the USB mass storage class specification and a FireWire device must be compliant with the FireWire Serial Bus Protocol 2 (SBP-2) specification. The following websites provide more information on these specifications:

- SCSI Architecture Model specifications (http://t10.org)—Provides computer interface and command set specifications and the FireWire Serial Bus Protocol 2 specification.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

- ATA/ATAPI standards (http://t13.org)—Provides access to the ATA/ATAPI-5 specification.
- USB specifications (http://www.usb.org)—Contains the USB Mass Storage Class Specification Overview.
- FireWire specifications (http://standards.ieee.org)—Provides access to FireWire standards.

  1394 Trade Association (http://1394ta.org)—Provides access to new and draft specifications for the IEEE 1394 standard.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Mass Storage Overview

The term mass storage encompasses a wide range of devices. Broadly defined, a mass storage device is any storage device that has media local to a machine and can support a file system. Mac OS X supports mass storage devices with a stack of drivers that manage the physical connection of the device to the bus, the translation of commands from the system to the device, and the device partitions that the user sees.

This chapter describes the construction and implementation of the mass storage driver stack. It also describes the SCSI Architecture Model family which implements the SCSI Architecture Model specifications in the transport driver layer and the Storage family which supports the device services layer.

Mass Storage Devices in Mac OS X

The Mac OS X mass storage driver stack represents a completely different model of mass storage device support from that in earlier versions of the Mac OS. Instead of a unique driver for every mass storage device, the Mac OS X mass storage driver stack separates device communication into different layers that comprise separate drivers. Apple-provided drivers provide services such as event handling and hot-plugging support, allowing you to write a driver that supports only the different or additional functionality your device requires.

In Mac OS X, ATA mass storage device drivers do not fully participate in the mass storage driver stack. A driver for an ATA mass storage device exists in a separate stack, providing the same functionality as the drivers in the lower layers of the mass storage driver stack. It does communicate with the device services layer, however, through an interface it provides.

The mass storage driver stack in Mac OS X version 10.1 and later supports SCSI devices of peripheral device types $00, $05, $07, and $0E. This includes devices such as CD-ROM and DVD-ROM drives, flash cards, and magneto-optical devices. In later versions of Mac OS X, however, the mass storage stack will include support for other SCSI peripheral devices, such as scanners. Until then, separate drivers outside the mass storage driver stack continue to provide support for these devices.

Traditionally, the term “SCSI” referred to the original parallel bus defined by the SCSI Architecture Model-1 and SCSI Architecture Model-2 specifications. With the introduction of Fibre Channel and Serial Storage Architecture (SSA), however, the SCSI Architecture Model-3 specification expanded the term to define an architecture that treats in a consistent manner devices that adhere to the SCSI Architecture Model specifications, independent of the physical bus the device is attached to.

In this document, the term “SCSI” refers to any physical bus (FireWire, ATAPI, parallel SCSI, or USB) or device that complies with the SCSI Architecture Model-2 specifications. This document refers to the original SCSI parallel bus technology as SCSI Parallel Interconnect (SPI) or parallel SCSI.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Mass Storage Drivers

In versions of the Mac OS before Mac OS X, mass storage I/O requests were serviced by a monolithic device driver that was responsible for every partition scheme, command set, and bus protocol associated with its device. There was a one-to-one correspondence between drivers and devices: A new device required a new driver, no matter how similar the new driver might be to a preexisting one. With the introduction of Mac OS X, however, this device driver model was replaced by the I/O Kit, an object-oriented driver development framework emphasizing modularity and reusability.

In Mac OS X, a driver for a mass storage device that mounts a file system or is bootable is a kernel extension, or KEXT. This KEXT inherits much of its functionality from an I/O Kit family that implements software abstractions common to all devices of a particular type.

In Mac OS X, a single mass storage device driver is therefore no longer responsible for partition information, bus protocols, or parameter block controls because these details are handled by other objects in the mass storage driver stack. In addition, the complexities introduced by features of Mac OS X such as its multithreaded kernel and hot-plugging support are handled by family code, freeing the driver to support the functions unique to its device or device type.

The I/O Kit's concept of family provides the perfect framework for the implementation of the industry standard SCSI Architecture Model. The SCSI Architecture Model specifications (http://t10.org) provide guidelines for implementing a system across all SCSI interconnect and protocol environments. The SCSI Architecture Model specifications are in no way confined to parallel SCSI devices alone. Transcending SCSI hardware, the SCSI Architecture Model defines the functional partitioning of the SCSI command sets and protocol standards for devices that understand command descriptor blocks and adhere to the SCSI Architecture Model specifications, regardless of the physical bus they are attached to.

The SCSI Architecture Model family (described in "The SCSI Architecture Model Family" (page 18)) implements the SCSI Architecture Model specifications in code abstractions that provide support for a wide range of devices across different buses. The Storage family (described in "The Storage Family" (page 21)) provides APIs that support access to the storage space represented by devices, independent of the underlying technology involved in the transport of data.

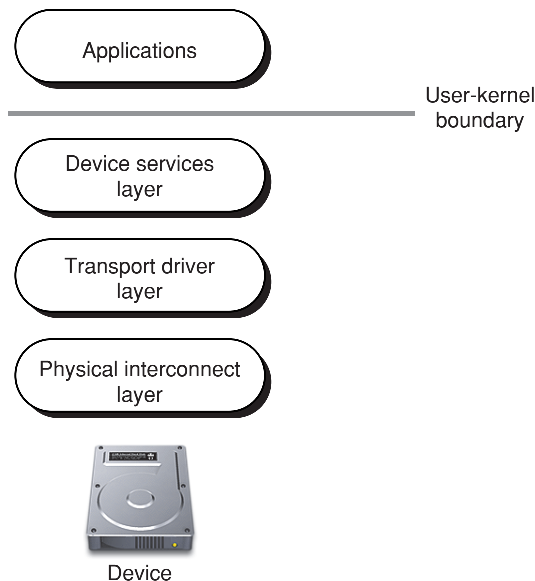
"The Mass Storage Driver Stack" (page 12) gives an overview of the mass storage driver stack, concentrating on the architecture of the stack and introducing the interaction of its layers. Although it's important to remember that the objects inhabiting these layers inherit from I/O Kit base classes and partake of functionality provided by I/O Kit families, it is equally important to avoid confusing the architecture of the driver stack with the "architecture" of the class hierarchy. To emphasize the distinction between the logical stacking of mass storage drivers and the I/O Kit class hierarchy of those drivers, the I/O Kit implementation of the driver stack is described separately in "Mass Storage Driver Objects" (page 17).

The Mass Storage Driver Stack

The mass storage driver stack consists of three fundamental layers, shown in Figure 1-1 (page 13). The stack is oriented with the physical device at the bottom and the ultimate client of that device (the application or the system) at the top.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Figure 1-1 The mass storage driver stack



At the top of the figure, above the device services layer, is the user-kernel boundary. Applications reside above this boundary along with application-based drivers, like those for scanners, tape drives, and digital cameras.

On the kernel side of the boundary is the top layer of the mass storage stack, the device services layer. This layer contains the generic block storage driver and optional filter schemes that can implement encryption or validation.

The generic block storage driver views a mass storage device as simply a storage space with no knowledge of device command sets or physical interconnect protocols. The filter schemes view mass storage devices even more abstractly: Mass storage devices can contain media objects that may represent any subset of a device, such as a disk partition, or even a set of multiple devices, such as a RAID disk controller that harnesses several disks together to appear as a single volume.

It is not likely that you will need to subclass the generic block storage driver to support a particular device's idiosyncrasies. A much better solution is to subclass a logical unit or protocol services driver in the transport driver layer. Creating a filter-scheme driver, however, is a good way to provide support for additional data manipulation your device may require. "The Device Services Layer" (page 15) describes this layer in more detail.

The middle layer of the mass storage stack is the transport driver layer. This layer comprises information about communicating with particular types of devices. I/O requests from the device services layer come into the transport driver layer where they are translated into commands suitable for the target device and then sent via the appropriate bus in the physical interconnect layer. The logical unit drivers and protocol services drivers from which you can subclass your device-specific and bus-specific drivers are in this layer. The transport driver layer itself is multilayered and is the focus of "The Transport Driver Layer" (page 14).

The layer above the device is the physical interconnect layer. As its name suggests, this layer consists of a collection of objects that are associated with the connection of the device to the bus. The bus controller drivers for FireWire, USB, and ATAPI are here, along with objects representing the device and, in some cases, logical portions of the device.

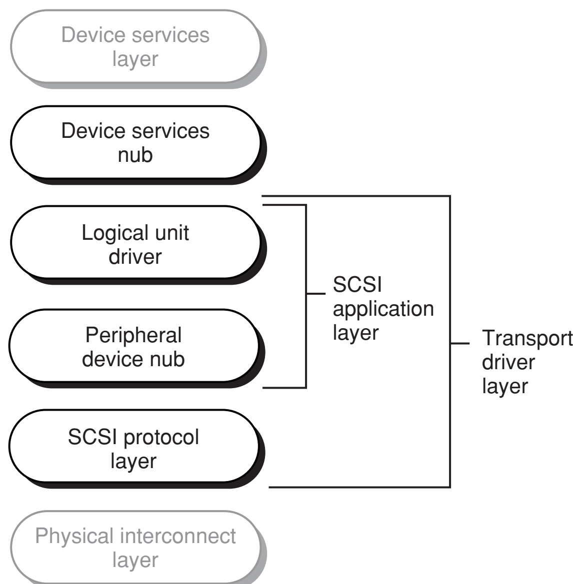
2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Unless there are bus-related issues you need to address for your device, it is not likely that you will need to subclass any physical interconnect driver.

The Transport Driver Layer

The transport driver layer transforms generic I/O requests into device-specific commands, suitable for transport on a particular bus. This layer (shown in Figure 1-2 (page 14)) consists of a link to the device services layer and two primary sublayers that accomplish the translation of I/O requests.

Figure 1-2 The transport driver layer



Between the device services layer and the transport driver layer proper is the device services nub. This nub presents the APIs the device implements and provides an attachment point between the block storage driver in the device services layer and the logical unit driver in the transport driver layer.

When a logical unit driver loads for a particular device, it publishes the corresponding device services nub which contains the device's type in the I/O Registry so that the appropriate block storage driver can be loaded. Once the block storage driver loads, it communicates with the objects in the transport driver layer through the device services nub.

The main responsibility of the SCSI application sublayer is to translate generic I/O requests into device-specific commands. This is done in the logical unit driver specific to the type of device. The SCSI Architecture Model specifications define several device categories, called peripheral device types. For example, sequential access devices such as tape drives are defined as peripheral device type $01 and printers are defined as peripheral device type $02.

Because neither tape drives nor printers are bootable or mount file systems, their logical unit drivers do not need to be in the kernel. Logical unit drivers for peripheral devices such as CD-ROM drives that are bootable and do mount file systems, however, must reside in the kernel. Apple provides four logical unit drivers corresponding to peripheral device types $00, $05, $07, and $0E:

- `IOSCSIPeripheralDeviceType00` for block storage devices such as internal disk drives

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

- IOSCSIPeripheralDeviceType05 for multimedia devices such as CD and DVD drives
- IOSCSIPeripheralDeviceType07 for magneto-optical devices
- IOSCSIPeripheralDeviceType0E for reduced block command devices such as flash cards and smart media devices

The logical unit driver uses an associated utility class called a command set builder to create an object called SCSITask. The SCSITask object (described more fully in "The SCSITask Object" (page 20)) contains the command descriptor block, or CDB, created from the generic I/O request together with all the data required during the life span of a single I/O transaction. This data includes information such as potential errors encountered, callback function pointers, and retry status. The SCSITask object is the fundamental unit of I/O transactions in the transport driver layer.

The other part of the SCSI application sublayer shown in Figure 1-2 (page 14) is the peripheral device nub. The function of the peripheral device nub is to determine what type of logical unit driver a particular device needs. When a device is discovered on the bus, the peripheral device nub queries it and publishes its device type in the I/O Registry so the appropriate logical unit driver can be loaded for it. After the peripheral device nub has fulfilled its function in the building of the mass storage driver stack, it is not involved in the subsequent processing of I/O requests.

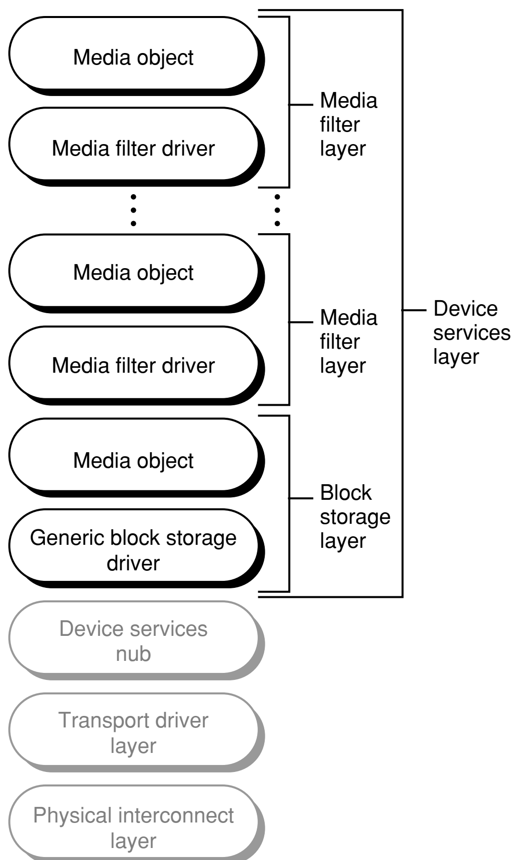
The SCSI protocol sublayer contains physical interconnect protocol–specific information. The protocol services drivers, IOUSBMassStorageClass, IOATAPIProtocolTransport, and IOFireWireSerialBusProtocolTransport, are responsible for translating the SCSITask object received from the SCSI application sublayer into a bus-specific format. For example, a hard drive plugged into a FireWire SBP-2 bus understands the CDB inside the SCSITask object but the FireWire SBP-2 bus does not. In order to process the SCSITask object, the IOFireWireSerialBusProtocolTransport driver removes the CDB and other elements from the SCSITask object and repackages them in an operation request block (or ORB) that is understood by the FireWire SBP-2 bus.

The Device Services Layer

The device services layer of the mass storage driver stack provides high-level support for random-access mass storage devices. The Storage family supports the device services layer and is responsible for sending generic I/O requests through the interface provided by the device services nub. The device services layer (shown in Figure 1-3 (page 16)) consists of the block storage driver layer and an arbitrary number of media filter layers.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Figure 1-3 The device services layer



At the top of the device services layer are the optional media filter layers. Each media filter layer comprises a filter-scheme driver and the media object it publishes. A filter-scheme driver is both a provider and a client of media objects: It receives abstract mass storage I/O requests from its client, performs the required data or offset manipulation, and passes on the modified request to its provider.

There are four basic kinds of media filter schemes:

- One-to-one—A block-level encryption or compression scheme, for example, matches against one media object and produces one media object that represents the unencrypted or uncompressed content.
- One-to-many—A partition scheme, for example, matches against one media object and produces multiple media objects each representing the content of one partition.
- Many-to-one—A RAID scheme, for example, matches against multiple media objects and produces a single media object that represents the aggregate content.
- Many-to-many—A media filter driver matches against multiple media objects and produces multiple media objects.

The block storage layer consists of the generic block storage driver and the media object it publishes. When the generic block storage driver appropriate to the device type loads, it publishes a media object representing the device in the I/O Registry. In addition to representing the device, the media object presents an interface

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

to the device with APIs implemented by the generic block storage driver. These APIs include `open`, `close`, `read`, and `write` functions that are appropriate to the device. The media object also has properties that reflect the properties of the device type it represents, such as natural block size and writability.

Between the device services layer and the transport driver layer proper, is the device services nub. This nub supplies the interface between the generic block storage driver in the device services layer and the device-specific logical unit driver in the transport driver layer.

When a logical unit driver loads for a mass storage device, it publishes the corresponding device services nub in the I/O Registry. The appropriate generic block storage driver matches on the device type published in the device services nub and loads. It then communicates all mass storage I/O requests across the interface the device services nub provides. This frees the generic block storage driver from all knowledge of the specific commands and mechanisms the transport driver layer objects employ to communicate with the device or bus.

Mass Storage Stack Implementation

The section "The Mass Storage Driver Stack" (page 12) describes the mass storage stack in architectural terms with only superficial regard to its implementation. This section describes the object-oriented implementation of the objects in the stack and the I/O Kit families that support the device services layer and the transport driver layer.

Mass Storage Driver Objects

The mass storage driver stack comprises I/O Kit objects that inherit from I/O Kit families. The I/O Kit defines a family as one or more C++ classes that implement software abstractions common to all devices of a particular type. A driver becomes a member of a family through inheritance: A driver's class is almost always a subclass of some class in a family. Being a member of a family means that a driver inherits the instance variables (data structures) and behaviors that are common to all members of the family.

When a device is discovered on the bus, the I/O Kit finds and loads an appropriate driver for it, using a subtractive matching process (for more information on this process, see "Driver Personalities and the Matching Process" (page 31)). Loading the driver causes the driver's family and all other objects the driver depends on to be loaded as well. The loading of the generic block storage driver, for example, causes the loading of the Storage family and all dependent classes, such as IOMedia.

The I/O Kit's object-oriented approach to driver development provides a way to separate common functionality from specific functionality and achieve modular and reusable code. For example, if your device is a FireWire SBP-2 hard drive that complies with the SCSI Architecture Model specifications for block storage devices, the Apple logical unit driver `IOSCSIPeripheralDeviceType00` is sufficient to drive it. If, however, your hard drive implements its `read` command differently than the specification, you can simply subclass the `IOSCSIPeripheralDeviceType00` driver to create a new driver whose only function is to override the `read` command implementation.

Then, by placing properties that uniquely describe your device into your driver's personality, your driver gets loaded when your device is discovered on the bus. Your driver then implements the `read` command, relying on the `IOSCSIPeripheralDeviceType00` driver to implement the remaining commands common to block storage devices. Similarly, the `IOSCSIPeripheralDeviceType00` logical unit driver relies on the SCSI Architecture Model family to implement functionality needed by all device drivers such as power management and work loops.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

If you're developing a filter-scheme driver that implements an encryption or validation scheme, the process is similar except that you do not subclass an existing filter-scheme driver. Instead, you subclass `IOStorage` and implement the appropriate methods to create a new filter scheme. Then, your disk utility program places a string that uniquely identifies your content into a partition. In your driver's personality, you place the same string and when the I/O Kit initiates matching on the new media objects the partition scheme publishes, your driver matches and loads.

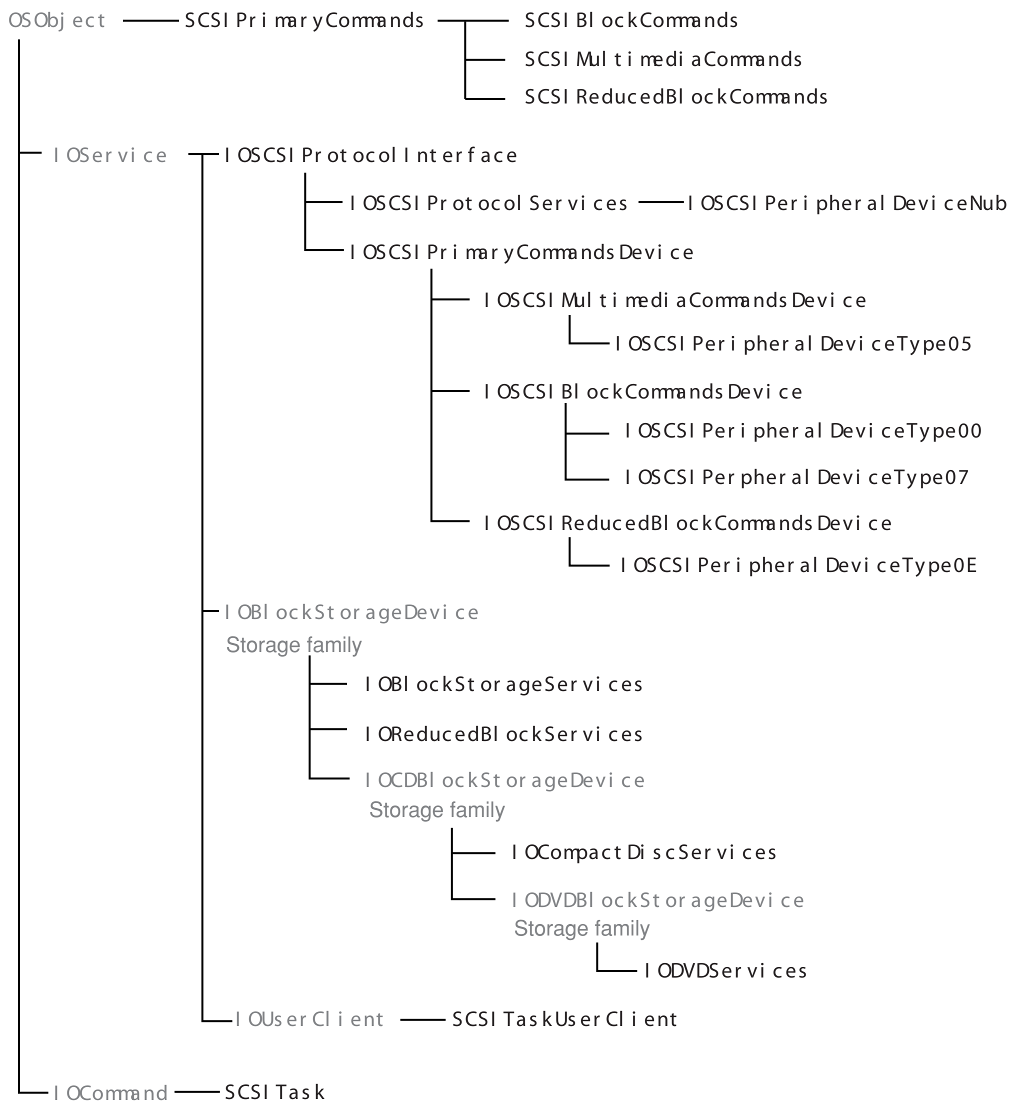
"Construction of a Mass Storage Driver Stack" (page 24) describes the building of an example mass storage driver stack and how a subclassed logical unit driver and a new filter-scheme driver fit in.

The SCSI Architecture Model Family

The SCSI Architecture Model family supports the transport driver layer of the mass storage driver stack. Figure 1-4 (page 19) illustrates the scope of the SCSI Architecture Model family by listing most of its leaf classes in a hierarchical inheritance chart.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Figure 1-4 SCSI Architecture Model family inheritance



At the top of the chart, inheriting from `SCSIPrimaryCommands`, are the three command set builder classes, `SCSIBlockCommands`, `SCSIMultimediaCommands`, and `SCSIReducedBlockCommands`. Each command set builder class corresponds to one of the shared command sets defined by the SCSI Architecture Model specifications: SCSI block commands, SCSI multimedia commands, and SCSI reduced block commands.

Each command set builder class implements every command listed in the corresponding shared command set specification. This allows the instantiated command set builder object to create a `SCSITask` object that is consistent with the shared command set specification the device is compliant with.

The base class for all logical unit drivers and protocol services drivers is `IOSCSIProtocolInterface`. This class provides the methods for sending, completing, and aborting commands.

Inheriting from `IOSCSIProtocolInterface` is the base class of the SCSI protocol sublayer, `IOSCSIProtocolServices`, and the peripheral device nub, `IOSCSIPeripheralDeviceNub`. The `IOSCSIProtocolServices` class provides the queueing model for sending commands across the physical

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

interconnect. Some of the other methods the IOSCSIProtocolServices class provides are methods for accessing the SCSITask object and its attributes and methods that get information about the CDB inside the SCSITask object.

Although the protocol services drivers such as IOFireWireSerialBusProtocolTransport and IOUSBMassStorageClass inherit from the SCSI protocol sublayer base class, they are not considered to be part of the SCSI Architecture Model family and therefore do not appear in Figure 1-4 (page 19). Instead, they are considered to be members of specific protocol families such as the FireWire family or the USB family.

Also inheriting from IOSCSIProtocolInterface is IOSCSIPrimaryCommandsDevice, the base class of the SCSI application sublayer. Some of the methods this class provides are methods to get, manipulate, and release SCSITask objects and methods to get and release objects instantiated from the command set builder classes.

The three subclasses of IOSCSIPrimaryCommandsDevice, IOSCSIMultimediaCommandsDevice, IOSCSIBlockCommandsDevice, and IOSCSIReducedBlockCommandsDevice, correspond to three of the shared command sets defined in the SCSI Architecture Model specifications. The in-kernel logical unit drivers are subclasses of these three classes.

Although the device services nubs, IOBlockStorageServices, IOReducedBlockServices, IOCompactDiscServices, and IODVDServices, inherit from base classes in the Storage family, they are considered to be members of the SCSI Architecture Model family. These nubs export APIs from the device services layer to the transport driver layer. Each nub exports the API that corresponds to the device's type. For example, the IOCompactDiscServices nub exports special methods for reading a disc's table of contents, getting and setting the audio volume, and getting and setting disc speed.

Inheriting from IOUserClient, the SCSITaskUserClient class provides application-based drivers with access to devices that can be supported by SCSI Architecture Model drivers. The SCSITaskUserClient class provides device interfaces for device access (described in "SCSI Architecture Model Family Device Interfaces" (page 21)) and should not itself be subclassed.

The SCSITask class, which inherits from IOCommand, provides methods to get and set the values of the SCSITask object's attributes and populate the CDB, among many others. The SCSITask class should not be subclassed.

The SCSITask Object

The SCSI Architecture Model family provides the logical unit and protocol services drivers access to CDBs through the SCSITask object. The SCSITask object is based on the SCSI command model, described in section 5 of the SCSI Architecture Model specification. The SCSI command model defines the format of the CDB and several status indicators relating to the execution of a SCSI command. The SCSITask object encapsulates these elements, giving you access to extensive information about the status of the command in addition to access to the CDB itself.

Status attributes of the SCSITask object include the following:

- Task attribute—defines how this task should be managed when determining order for queueing and submission to the appropriate device server.
- Task state—represents the current state of the task such as new, enabled, or blocked.
- Task status—represents the completion status of the task.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

The SCSITask object also includes information such as the service response from the transport driver, the data transfer direction, and memory buffers. The header file defining the SCSITask object and its accessor methods is in /System/Library/Frameworks/IOKit.framework/Headers/scsi-commands/SCSITask.h.

SCSI Architecture Model Family Device Interfaces

The SCSI Architecture Model family provides a device interface mechanism that allows applications to send commands to mass storage devices that are controlled by SCSI Architecture Model family drivers. For more detailed information on how to use this device interface mechanism, see *SCSI Architecture Model Device Interface Guide*.

There are two access modes available to applications: Exclusive and nonexclusive. Exclusive access consists of an application acting as the logical unit driver for a device. For example, if a tape drive is discovered on the FireWire bus, the peripheral device nub (described in "The Transport Driver Layer" (page 14)) publishes device type $01 in the I/O Registry. There are no in-kernel logical unit drivers for that peripheral device type so the matching and loading process comes to a halt.

When an application-based driver for peripheral device type $01 launches, however, it finds the nub representing the tape drive in the I/O Registry and instantiates a SCSITaskDeviceInterface object. This gives the application unrestricted access to the device—in short, the application becomes the logical unit driver.

Nonexclusive access is available for authoring applications. If a CD or DVD drive is discovered, the I/O Kit finds and loads the in-kernel logical unit driver for peripheral device type $05 and the rest of the stack is built as usual. As long as no other client currently holds exclusive access to the device, an authoring application can gain nonexclusive access to the device by instantiating an MMCDeviceInterface object. It can then use device interface APIs to get information such as the amount of free space on the device.

If the authoring application later requires exclusive access to the device, for example, to burn a CD or DVD, it first reserves the media in the drive. Then it instantiates a SCSITaskDeviceInterface object and requests exclusive access. At this point, the in-kernel logical unit driver yields control to the application and the device services layer of the mass storage stack is torn down. The application then serves as the logical unit driver until it is closed, at which point the in-kernel logical unit driver regains control and the rest of the stack is rebuilt.

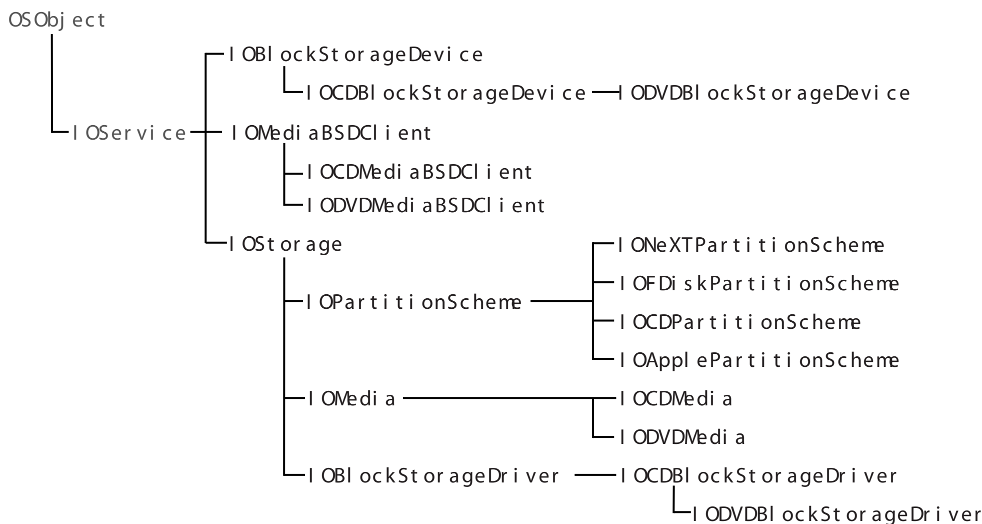
When an application has exclusive access to a device, it uses objects provided by a third device interface, the SCSITaskInterface, to manipulate the in-kernel SCSITask objects. Each SCSITaskInterface object corresponds to exactly one SCSITask object, allowing the application the same access to commands enjoyed by in-kernel logical unit drivers.

The Storage Family

The Storage family supports the device services layer of the mass storage driver stack. Figure 1-5 (page 22) shows the Storage family in a hierarchical inheritance chart.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Figure 1-5 Storage family inheritance



At the top of the chart is the abstract class `IOBlockStorageDevice`. This class declares the interface to the underlying mechanisms of the transport driver layer that transport data to and from the represented storage space. Mass storage I/O requests pass through this interface without any involvement in the actual commands the transport driver layer objects use to communicate with the device. A transport family or driver subclasses `IOBlockStorageDevice`, implements the interface APIs, and instantiates a device services nub object. The generic block storage driver then drives this object, unconcerned with the device-specific and bus-specific objects below it.

The SCSI Architecture Model family declares two subclasses of `IOBlockStorageDevice`, the device services nubs `IOBlockStorageServices` and `IOReducedBlockServices`. These device services nubs relay generic requests from the generic block storage driver to the device-specific logical unit driver in the transport driver layer (for more information on this family, see "The SCSI Architecture Model Family" (page 18)).

There are other families and drivers that subclass `IOBlockStorageDevice` to create nubs that provide the interface between device-specific transport drivers and the generic block storage driver. These include the ATA transport layer (`IOATABlockStorage`), the USB UFI transport layer, and the Disk Images transport layer.

The Storage family declares two subclasses of `IOBlockStorageDevice`: `IOCDBlockStorageDevice` and `IODVDBlockStorageDevice`. These subclasses provide the protocol for generic CD and DVD functionality, independent of the underlying physical interconnect protocol. The SCSI Architecture Model family subclasses `IOCDBlockStorageDevice` for the device services nub `IOCompactDiscServices` and subclasses `IODVDBlockStorageDevice` for the device services nub `IODVDServices`.

The `IOStorage` class is the common base class for both driver and media objects. It is an abstract class that declares the basic `open`, `close`, `read`, and `write` interfaces that its subclasses implement. The `read` and `write` interfaces provide byte-level access to the storage space. The `IOStorage` class also establishes the protocol media objects use to communicate with driver objects without needing media objects to be subclassed for each driver.

The `IOMedia` subclass of `IOStorage` is a random-access disk device abstraction. It provides a consistent interface for both real and virtual disk devices, for subdivisions of disks, such as partitions, and for supersets of disks, such as RAID volumes. The `IOMedia` class implements the appropriate `open`, `close`, `read`, `write`, and matching semantics for media objects. It has properties that reflect the properties of actual media, such as its total size in bytes and whether it is ejectable.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

The subclasses of IOMedia provide properties, methods, and advanced interfaces that are specific to CD and DVD media objects. The IOCDMedia subclass includes properties such as the type of CD media object and its table of contents, and implements methods that read special areas of the CD. The IODVDMedia subclass includes properties that describe the type of DVD media object, such as DVD-ROM or DVD-R/W, and implements additional methods specific to DVDs.

The IOMediaBSDClient class publishes a BSD interface for all media objects. The IOMediaBSDClient driver represents IOMedia objects as device files in the Mac OS X BSD execution environment. You can access these device files using BSD APIs such as open, close, read, write, and ioctl. The ioctl system call provides methods to determine various media properties and control various aspects of the media. The subclasses of IOMediaBSDClient, IOCDMediaBSDClient and IODVDMediaBSDClient, extend ioctl behavior to include CD-specific and DVD-specific functionality.

Partition-scheme drivers inherit from IOPartitionScheme, an abstract subclass of IOStorage. Apple provides the following partition schemes:

- IOApplePartitionScheme, the standard Apple partition-scheme driver
- IOFDiskPartitionScheme, the standard PC partition-scheme driver
- IONeXTPartitionScheme, the NeXT partition-scheme driver
- IOCDPartitionScheme, the partition-scheme driver for CD tracks that require treatment as partitions

The IOPartitionScheme class provides a basic framework for a partition-scheme driver that implements the appropriate open and close semantics for partition objects, and the default read and write interfaces. Although the open, close, read, and write implementations IOPartitionScheme provides are sufficient for simple partition schemes, more complicated schemes may need to perform more processing.

The IOBlockStorageDriver subclass of IOStorage is the common base class for generic block storage drivers. It extends the IOStorage protocol by implementing methods such as deblocking for unaligned transfers, polling for ejectable media, and statistics gathering and reporting. Because the IOBlockStorageDriver functions independently of the underlying device and bus transport protocols, you should not subclass it to handle device idiosyncrasies. A new type of generic device, however, might require the subclassing of the IOBlockStorageDriver.

The IOCDBlockStorageDriver subclass of IOBlockStorageDriver implements methods that support CD drives, such as getting information related to the table of contents and reading special areas of the disc. The IODVDBlockStorageDriver subclass of IOCDBlockStorageDriver implements methods that support DVD drives such as getting information related to the encryption and key for the drive.

Filter Schemes

A mentioned in “The Device Services Layer” (page 15), a filter scheme driver is both the client of an IOMedia object and the provider of IOMedia objects. If you’re interested in developing a filter-scheme driver you might assume that you need to subclass an existing partition scheme. This is unnecessary, however, because you can access your content within an existing partition. A partition-scheme driver, such as IOApplePartitionScheme, publishes a distinct IOMedia object for each partition’s contents. If you’ve placed your content within a partition, your filter-scheme driver can match on your unique identifier, contained in the content hint property of the IOMedia object representing that partition (see Figure 1-9 (page 28) for an example of how this might look). As a subclass of IOStorage, therefore, your filter-scheme driver can match directly on your content within a partition and avoid the I/O overhead and potential stale data issues associated with actively probing the media for your signature.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

It's important to realize that a filter-scheme driver should never produce an IOCDMedia or IODVDMedia object, because these objects have provider requirements a filter-scheme driver would be unable to meet. For example, an IODVDMedia object has requirements specific to DVD media that only an IODVDBlockStorageDriver (or subclass) can meet. An IOMedia object, on the other hand, has more generic requirements that an IOStorage subclass (such as a custom filter-scheme driver) can meet. See "Developing a Filter Scheme" (page 65) for information on how to implement a filter-scheme driver.

Accessing IOMedia Objects From Applications

The Storage family provides a device interface to access IOMedia objects from applications using the BSD device interface. Each IOMedia object has a BSD client driver that produces a device node (in the form of /dev/disk) in the Mac OS X BSD execution environment. Applications can use the read and write system calls to access the data represented by an IOMedia object and ioctl system calls to manipulate the special characteristics of devices.

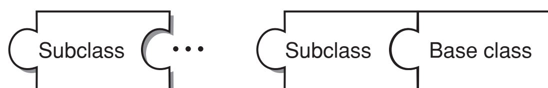
Applications can use standard I/O Kit search and notification APIs to find specific IOMedia objects. An application searching for a CD, for example, can create a matching dictionary for the subclass IOCDMedia and, using properties the IOMedia object publishes, narrow the search to ejectable media only. An example of this process is in *Device File Access Guide for Storage Devices*.

Construction of a Mass Storage Driver Stack

This section describes how the mass storage driver stack is built up from the discovery of a specific device. Then, it shows how a subclassed logical unit driver and a new filter-scheme driver fit into the stack.

The illustrations in this section use a "puzzle piece" shape (shown in Figure 1-6 (page 24)) to show the inheritance chain of each subclassed driver.

Figure 1-6 Subclasses inheriting from a base class

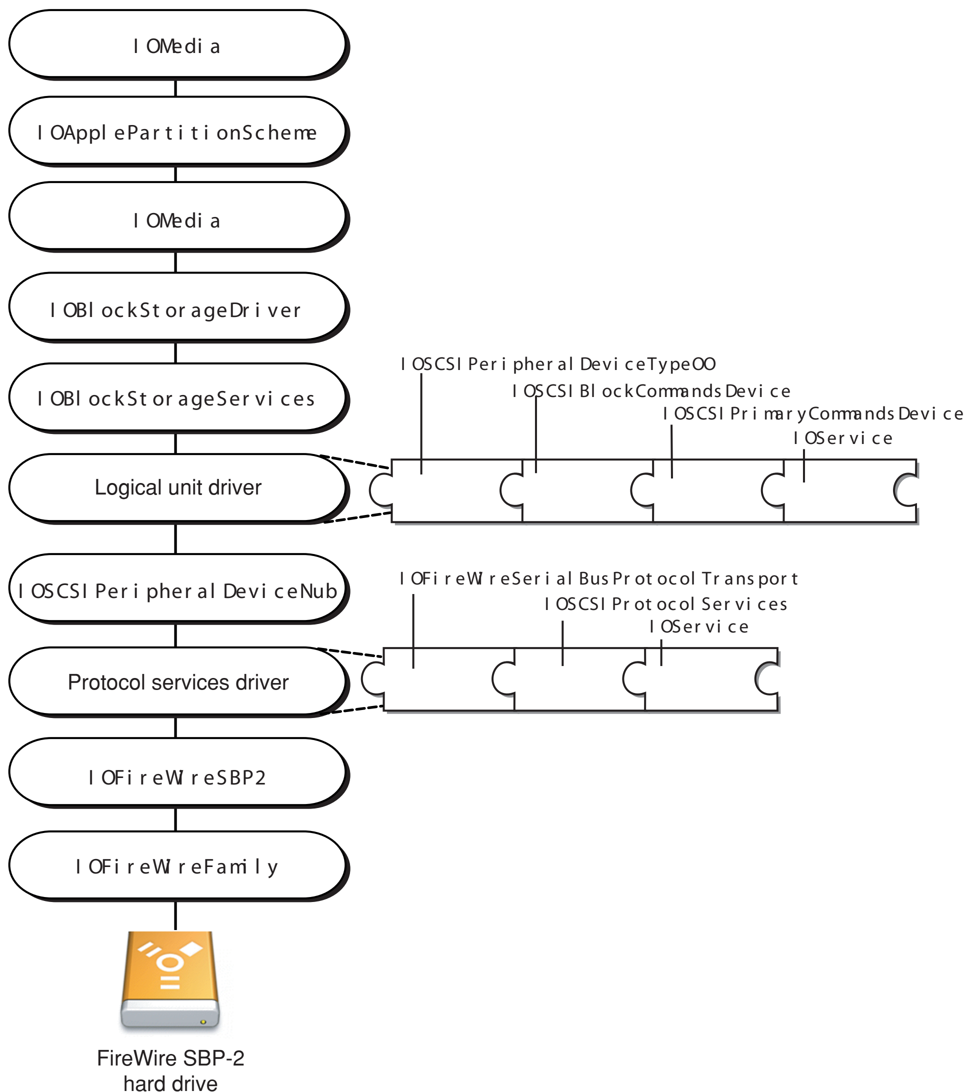


Inheritance runs from right to left: Each subclass locks onto its ancestor on its right edge and provides a projection for another (potential) subclass on its left edge.

The device in this example is a FireWire SBP-2 hard drive. Figure 1-7 (page 25) shows the stack layers built up above the hard drive, with inheritance chains shown for the logical unit driver and the protocol services driver.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Figure 1-7 An example mass storage stack



When you plug in the hard drive, the FireWire bus controller in the physical interconnect layer discovers it and instantiates an `IOFireWireDevice` object. The `IOFireWireDevice` object scans the device's configuration ROM and produces an `IOFireWireUnit` object for each unit directory in the device. The I/O Kit performs matching on the `IOFireWireUnit` and, since the hard drive is a FireWire SBP-2 device, instantiates an `IOFireWireSBP2Target` object.

The `IOFireWireSBP2Target` object scans the device's configuration ROM and produces an `IOFireWireSBP2LUN` object for each logical unit it finds. After performing matching on the `IOFireWireSBP2Target` object, the I/O Kit instantiates the `IOFireWireSerialBusProtocolTransport` driver object.

The instantiation of the protocol services driver causes the instantiation of the peripheral device nub, which sends an inquiry command to the device. The response to this inquiry describes the device's type and the peripheral device nub publishes a nub containing the key "peripheral device type 00" in the I/O Registry.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

The I/O Kit performs matching on this nub, ultimately finding and loading the `IOSCSIPeripheralDeviceType00` driver. When the `IOSCSIPeripheralDeviceType00` driver object instantiates, the corresponding device services nub, `IOBlockStorageServices`, instantiates and initiates the matching process for the block storage driver.

When the matching block storage driver is found, it loads and publishes an `IOMedia` object that represents the whole device. The I/O Kit then finds and loads a partition-scheme driver that matches on the whole device. For Apple-formatted disks, this will probably be `IOApplePartitionScheme`. `IOApplePartitionScheme` then publishes an `IOMedia` object for each partition it finds.

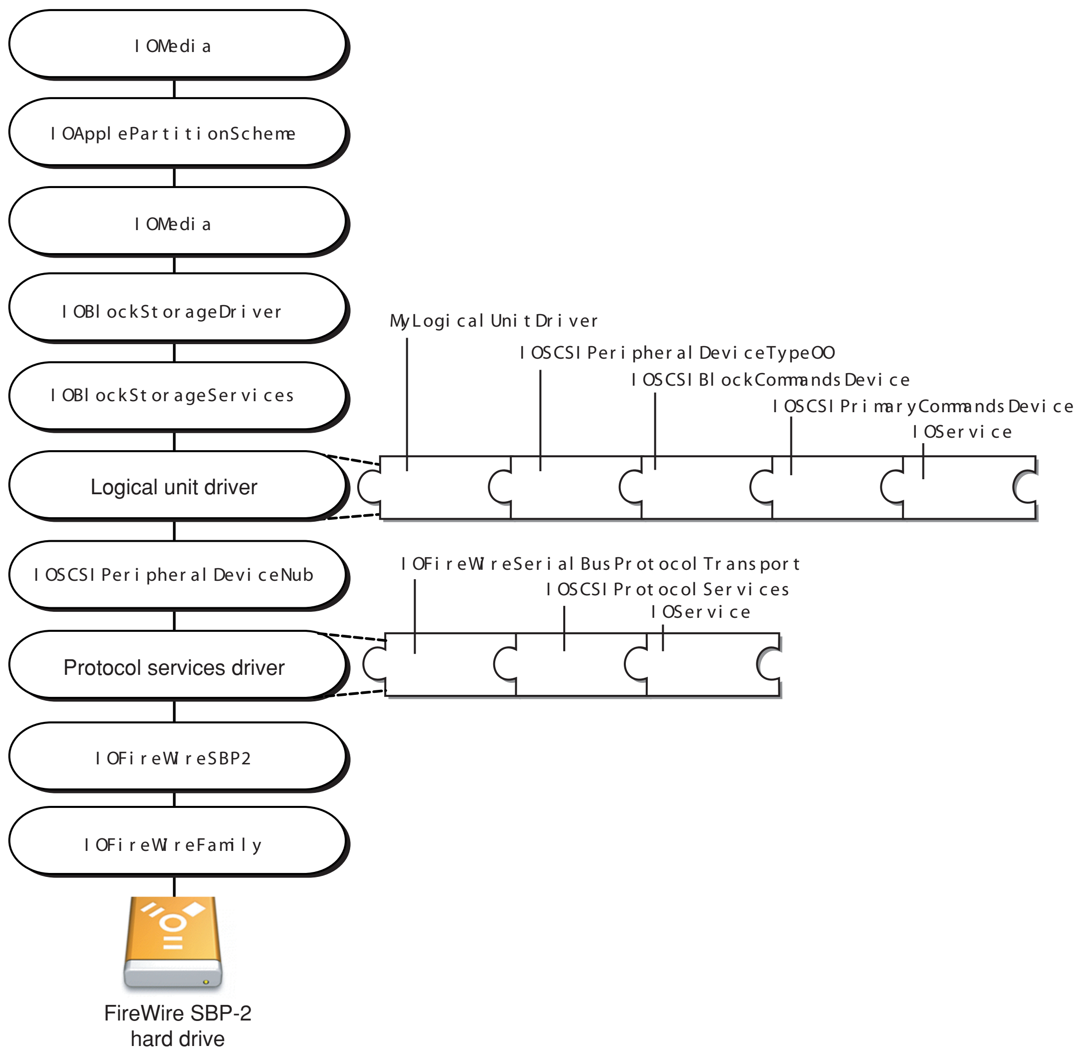
Now, imagine you need to support a FireWire SBP-2 hard drive that implements the `read` command differently than as defined by the SCSI Architecture Model specifications. You subclass the `IOSCSIPeripheralDeviceType00` driver and override the `read` method. Because your driver should be loaded only for your device, you place matching information that uniquely identifies your device in your driver's personality dictionary.

When the I/O Kit discovers your device, it builds the mass storage driver stack as before until it searches for a logical unit driver. The I/O Kit matching process gives the driver with the most matching keys the first chance to drive a device. Since your driver matches on vendor, product, and perhaps even software revision values associated with your device, the I/O Kit favors your driver over the `IOSCSIPeripheralDeviceType00` driver which matches only on the peripheral device type. Your driver loads, along with the SCSI Architecture Model family and other dependencies.

Figure 1-8 (page 27) shows the mass storage stack after your logical unit driver (named "MyLogicalUnitDriver") matches and loads.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Figure 1-8 Adding a subclassed logical unit driver



Now suppose you want to implement an encryption scheme that works over any transport. This new encryption-scheme driver would match on an `IOMedia` object and produce another `IOMedia` object. To create the driver, you subclass `IOStorage` and implement the encryption and decryption behavior in the driver's `read` and `write` methods. Because your filter-scheme driver should be loaded only for a partition containing your content, you place a content-hint string, of the form *MyCompany_MyContent*, in your driver's personality.

Then, you make a disk utility program available so a user can format the disk to contain your encryption scheme. When the disk utility program reformats the disk, it also places your content-hint string in one of the partitions.

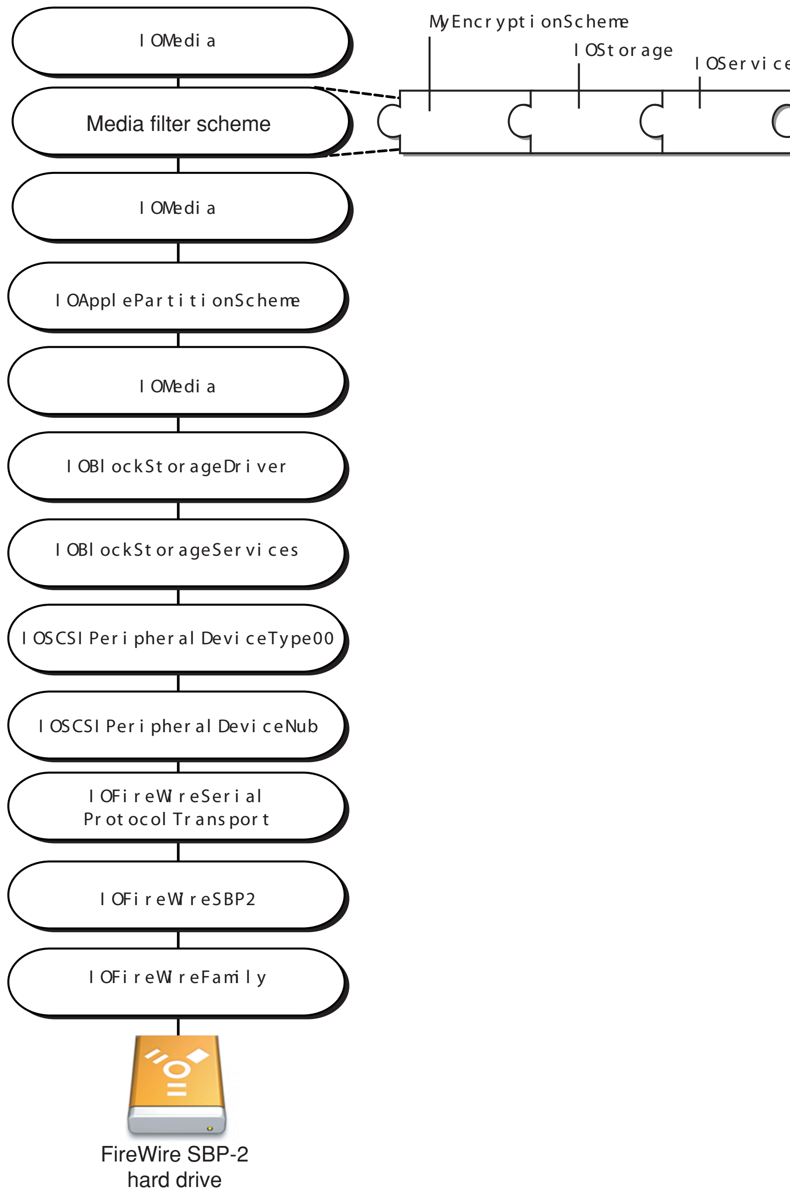
When the disk utility program completes its task, the partition-scheme driver publishes an `IOMedia` object for each partition. This causes an update to the I/O Registry. The I/O Kit then searches for a filter-scheme driver to match on the content-hint property in the new `IOMedia` object and it finds just one—your encryption-scheme driver.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Because your driver matches on the same content-hint string your disk utility program placed in the partition, there is no doubt that this is your content, so implementing the `probe` method is optional. If you don't implement the `probe` method, by default it returns `true`.

When your encryption-scheme driver loads, it publishes an `IOMedia` object representing the unencrypted content on the media. Figure 1-9 (page 28) shows the mass storage driver stack after your driver, called "MyEncryptionScheme," loads.

Figure 1-9 Adding an encryption scheme



2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Mass Storage Device Compliance

Apple provides mass storage device drivers in the transport driver layer that support various device specifications. In order for your device to work with these drivers, it must comply with the appropriate specifications. This chapter describes device compliance and lists the logical unit and protocol services drivers Apple provides.

The concept of device compliance has no meaning in the device services layer. The generic block storage driver treats the device as a storage space and media filter-scheme drivers work with media present in the device; neither makes any assumptions about underlying transport specifications or implementation. For more information about how to develop your own filter-scheme driver, see "Filter-Scheme Driver Matching" (page 43) and "Developing a Filter Scheme" (page 65).

Device Compliance

Apple provides logical unit and protocol services drivers at the transport driver layer of the mass storage driver stack (shown in Figure 1-2 (page 14)). These drivers will drive any mass storage device that complies with the supported specifications.

There are two areas in which a device must be compliant in order to partake of the services of the provided drivers:

- SCSI command set implementation
- Physical interconnect transport protocol

SCSI command set implementation compliance means that a device's firmware processes commands as documented in a SCSI Architecture Model shared command set specification. For example, if a multimedia device processes commands as defined by the SCSI multimedia command set specification, it is considered compliant and the Apple-provided `IOSCSIPeripheralDeviceType05` driver will drive it successfully.

Compliance with a physical interconnect transport protocol means that a device sends and receives commands according to the protocol defined by the bus it's on. For example, in order for a USB device to be compliant with the USB mass storage class, it must comply with one of the subclasses defined by the USB Mass Storage Class Specification. The Apple-provided `IOUSBMassStorageClass` protocol services driver will drive such a device successfully.

Available Mass Storage Drivers

For SCSI command set implementation-compliant devices, Apple provides four logical unit drivers that support the following specifications:

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

- The `IOSCSIPeripheralDeviceType00` driver supports block storage devices that comply with the SCSI block commands specification.
- The `IOSCSIPeripheralDeviceType05` driver supports multimedia devices that comply with the SCSI multimedia commands specification.
- The `IOSCSIPeripheralDeviceType07` driver supports magneto-optical devices that comply with the SCSI block commands specification.
- The `IOSCSIPeripheralDeviceType0E` driver supports reduced block command devices that comply with the SCSI reduced block commands specification.

For physical interconnect transport protocol-compliant devices, Apple provides protocol services drivers that support the following bus transport protocols:

- The `IOFireWireSerialBusProtocolTransport` driver supports FireWire Serial Bus Protocol 2 (SBP-2) mass storage devices defined in the SCSI Architecture Model specifications (http://t10.org).
- The `IOUSBMassStorageClass` driver supports USB mass storage devices that comply with the USB Mass Storage Class specification (http://www.usb.org). For a listing of the supported USB Mass Storage Class subclasses and protocols, see "The USB Mass Storage Class Protocol Services Driver" (page 36).
- The `IOATAPIProtocolTransport` driver supports ATAPI mass storage devices that comply with the ATA/ATAPI-5 specification (http://t13.org).

If your device is compliant with both a SCSI Architecture Model shared command set specification and a physical interconnect transport protocol, you will not have to write your own driver for it. If, however, your device is not compliant with these specifications or protocols, you will need to subclass the appropriate Apple-provided driver to address the difference. Similarly, if your device provides additional functionality at the command set implementation or bus transport level, you will need to develop a subclass that supports the new feature.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Mass Storage Driver Matching and Loading

Before a mass storage device can be used, the I/O Kit must find and load several drivers for it. In the transport driver layer, the required drivers are a protocol services driver and a logical unit driver. The only required driver in the device services layer is the generic block storage driver but if there is media present in the device, there may be partition and filter-scheme drivers, as well. Like all I/O Kit drivers, these drivers must declare what devices they are suited to drive by placing device-specific or device type-specific information in special dictionaries called personalities. In a process called driver matching, the I/O Kit compares this information to values reported by the device to find the most suitable driver.

This chapter first describes the driver matching process in general and then focuses on the matching semantics of the protocol services drivers, the logical unit drivers, and the optional filter-scheme drivers.

Driver Personalities and the Matching Process

The I/O Kit finds and loads device drivers in a three-stage matching process that excludes unsuitable drivers from the pool of candidates until one or more eligible drivers are left. The most eligible of the remaining drivers is then given the first opportunity to drive the device.

This process makes use of matching dictionaries that are in every driver's information property list. Each dictionary, consisting of XML key-value pairs, specifies a personality of the driver, which declares its suitability for a particular device or device type. A driver may have more than one personality if it can drive different devices or device types.

This section presents a brief overview of driver personalities and matching (for a more in-depth description of these topics, see *I/O Kit Fundamentals*.) The subsequent sections, "Protocol Services Driver Matching" (page 33), "Logical Unit Driver Matching" (page 41), and "Filter-Scheme Driver Matching" (page 43) describe how this process is implemented by specific mass storage drivers.

Driver Personalities

Every device driver must declare one or more personalities that define the types of devices it can support. These personalities are in the form of XML dictionaries contained in the information property list (`Info.plist` file) in the driver's kernel extension (KEXT) bundle.

Each entry in the matching dictionary is made up of a key-value pair in which the XML tags `<key>` and `</key>` enclose the key and the associated value is enclosed by XML tags that indicate its data type.

At minimum, all driver personalities contain the following two keys:

- The `IOClass` key declares the name of the class the I/O Kit instantiates when probing. For example,

```
<key>IOClass</key>
<string>IOATAPIProtocolTransport</string>
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

tells the I/O Kit to instantiate the IOATAPIProtocolTransport driver when probing the device for this personality.

- The IOProviderClass key declares the name of the nub class a driver personality attaches to. For example,

```
<key>IOProviderClass</key>
<string>IOFireWireSBP2LUN</string>
```

  tells the I/O Kit that this driver personality attaches to an IOFireWireSBP2LUN nub.

  The provider class defines the family-specific matching keys used in the passive matching step, described in "Driver Matching" (page 32).

Most driver personalities also contain the IOProbeScore key, which declares the initial probe score for a personality. For example,

```
<key>IOProbeScore</key>
<integer>5000</integer>
```

declares a base probe score of 5000, which can be increased or decreased during the matching process, reflecting the driver's suitability for the device.

Driver Matching

Driver matching occurs when a device is discovered. Each candidate driver has a probe score that reflects how well suited it is to drive the device. During the matching process, the family can increase the probe score with each property match. The driver with the highest probe score is given the first opportunity to drive the device.

The driver matching process consists of three phases:

1. In the class matching phase, the I/O Kit eliminates drivers of the wrong class. For example, the I/O Kit eliminates drivers that descend from a SCSI class when searching for a USB driver.

2. In the passive matching phase, the I/O Kit examines the personality of the driver for family-specific properties. In the SCSI Architecture Model family, the more matching properties found, the higher the driver's probe score. For example, a driver that matches on both vendor name and product name has a higher probe score than a driver that matches only on vendor name. The Storage family, on the other hand, does not influence a driver's probe score during matching.

   Often this step is sufficient to determine if a driver is suitable for a device. If there is no family-specific matching, however, the next step is automatically invoked.

3. In the active matching phase, the candidate driver is allowed to communicate with the device and verify that it can drive it. The I/O Kit loads the drivers remaining after the passive matching phase and each driver's probe function is called with reference to the device it is being matched against. The probe method can examine the device in any way it chooses, as long as it leaves the device in the same state in which it was found.

   For example, a vendor may use certain bits of a property value to signify the presence of a particular device component. A logical unit driver for that device can implement a probe method that examines those bits to determine if the component is indeed present.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Depending on the results of the probe, the driver increases or decreases its probe score to indicate its suitability to drive the device.

The I/O Kit chooses the driver with the highest probe score and starts it. If the driver starts successfully, any remaining driver candidates are discarded. If it is not successful, the driver with the next highest probe score is started and the process continues until a successful driver is found.

Driver Starting

When probing, a driver can perform a detailed examination of the device, including, if necessary, memory allocations, but it must leave the device in the same state in which it found it. If a driver starts successfully, it can reuse the memory it allocated in its `probe` method but if it is unsuccessful, it must be sure to deallocate the memory in its `free` method.

When a driver starts, it should call its superclass's `start` method before doing anything else. If the superclass's `start` method succeeds, the driver can then perform its initializations or allocations. Because a driver may not be able to perform initializations or allocations safely after it starts, it should perform such tasks in its `start` method. If the driver is unable to complete its tasks, it can notify the I/O Kit and the driver with the next highest probe score starts.

Protocol Services Driver Matching

During the building of the mass storage driver stack, objects in the physical interconnect layer discover a mass storage device and publish a nub representing it in the I/O Registry. The I/O Kit finds a protocol services driver by performing driver matching on this nub.

The protocol services drivers rely on matching semantics that are specific to the family of the bus they communicate with. The following sections describe the matching properties and process for each Apple-provided protocol services driver.

The FireWire SBP-2 Protocol Services Driver

As described in "Construction of a Mass Storage Driver Stack" (page 24), the protocol services driver for a FireWire SBP-2 mass storage device must match on the `IOFireWireSBP2LUN` nub published by the `IOFireWireTarget` object in the physical interconnect layer. The `IOFireWireSBP2LUN` object contains the following seven keys:

- `Command_Set`
- `Command_Set_Spec_ID`
- `Vendor_ID`
- `Command_Set_Revision`
- `IOUnit`
- `Firmware_Revision`
- `Device_Type`

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

The IOFireWireTarget object scans the device's configuration ROM and fills in the values for these keys. If the device doesn't declare one or more of these properties in its configuration ROM, the IOFireWireSPB2LUN publishes the corresponding key with a zero value.

To match on the IOFireWireSBP2LUN, the IOFireWireSerialBusProtocolTransport driver personality includes the keys shown in Listing 3-1 (page 34).

Listing 3-1 The IOFireWireSerialBusProtocolTransport driver personality dictionary

```
<dict>
    <!-- CFBundleIdentifier denotes the name of the driver in
     -- reverse DNS notation. -->
    <key>CFBundleIdentifier</key>
    <string>com.apple.iokit.IOFireWireSerialBusProtocolTransport</string>

    <!-- Command_Set refers to the organization responsible
     -- for the definition of the command set. -->
    <key>Command_Set</key>
    <integer>66776</integer>

    <!-- Command_Set_Spec_ID specifies the commands
     -- understood by the device. -->
    <key>Command_Set_Spec_ID</key>
    <integer>24734</integer>

    <!-- The name of the class the I/O Kit instantiates
     -- when probing. -->
    <key>IOClass</key>
    <string>IOFireWireSerialBusProtocolTransport</string>

    <!-- The initial probe score for this personality. -->
    <key>IOProbeScore</key>
    <integer>4096</integer>

    <!-- The provider class is the name of the nub class this
     -- driver personality attaches to. -->
    <key>IOProviderClass</key>
    <string>IOFireWireSBP2LUN</string>

    <!-- The next two keys describe which
     -- bus the device is on and whether it is
     -- internal or external.-->
    <key>Physical Interconnect</key>
    <string>FireWire</string>

    <key>Physical Interconnect Location</key>
    <string>External</string>
</dict>
```

A subclass of the IOFireWireSerialBusProtocolTransport driver can use more of the seven keys in the IOFireWireSBP2LUN object to more narrowly define the device it is suited to drive. It can also examine the property values in its probe method to further determine its suitability. For example, a vendor can use some of the bits in a property value to declare the presence of a device component. A driver that needs to determine the presence of this component can examine those bits in its probe method. Listing 3-2 (page 35) shows how this can be done for a subclass of the IOFireWireSerialBusProtocolTransport driver.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Listing 3-2 Example FireWire protocol services driver probe method

```
// This example probe method tests the Firmware_Revision value.
IOService *com_MySoftwareCompany_driver_MyFWProtocolLayerDriver::probe(
                IOService *provider, SInt32 *score )
{
    IOFireWireSBP2LUN *fwSBP2LUN = NULL;
    OSObject *firmwareObject;
    IOService *returnValue = 0;

    // Override probe method inherited from IOFireWireSBP2LUN.
    // Incorporate additional matching based on bits within
    // firmware revision data.

    // Allow superclass first chance at probe
    if ( !IOFireWireSerialBusProtocolTransport::probe( provider, score ) )
        goto ErrorExit;

    fwSBP2LUN = OSDynamicCast( IOFireWireSBP2LUN, provider );
    if ( fwSBP2LUN == NULL )
        goto ErrorExit;

    // Get key from registry that IOFireWireSBP2LUN published
    firmwareObject = provider->getProperty( "Firmware_Revision" );
    if ( firmwareObject )
    {
        OSNumber *firmwareNumberObject;
        UInt32 firmwareValue = 0;

        // Translate the Firmware_Revision property
        // into an OSNumber value for inspection.
        firmwareNumberObject = OSDynamicCast( OSNumber, firmwareObject );
        if ( firmwareNumberObject )
        {
            firmwareValue = firmwareNumberObject->unsigned32BitValue( );
        }

        // Check bits 8 through 23 of the Firmware_Revision value by
        // comparing them with the constants kMyConstant1 and
        // kMyConstant2.
        // These constants represent identification codes and
        // would be defined earlier in the driver's code.
        if ( ( ( ( firmwareValue >> 8 ) & 0x000FFF ) == kMyConstant1 )
        ||
         ( ( ( firmwareValue >> 8 ) & 0x000FFF ) == kMyConstant2 ) )
        {
            IOLog( "%s: Device component detected\n", getName( ) );
            returnValue = this;
        }
    }
ErrorExit:
    return returnValue;
}
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

The USB Mass Storage Class Protocol Services Driver

When a USB mass storage class device is discovered, the USB family abstracts the contents of the device descriptor into an I/O Kit nub object called `IOUSBDevice`. The device descriptor includes information such as the device's class and subclass, vendor and product numbers, and the number of configurations.

Because USB mass storage class devices are defined as composite class devices, the `AppleUSBComposite` driver matches against the `IOUSBDevice` nub object and sets the first configuration in the device. This causes the USB family to abstract each interface descriptor in the configuration into an `IOUSBInterface` nub object. The `IOUSBMassStorageClass` driver then matches on the mass storage class-compliant interface nub objects.

The Apple-provided `IOUSBMassStorageClass` driver contains six personalities that correspond to the six mass storage subclasses. Each subclass represents the type of command block set the device's interfaces use. If the device is compliant with the USB mass storage class specification, its interface descriptor contains its subclass and protocol in the `bInterfaceSubClass` and `bInterfaceProtocol` fields, respectively.

The subclass code in the `bInterfaceSubClass` field refers to one of the subclasses listed in Table 3-1 (page 36). These codes denote industry-standard specifications that describe the command block definitions used by the interfaces of USB mass storage class devices. They do *not* refer to specific device types since most USB mass storage class devices can choose to comply with any command block specification.

Table 3-1 USB mass storage class subclasses

Subclass code	Command block specification	Typical usage
0x01	Reduced Block Commands (RBC)	Flash device, other mass storage class devices
0x02	SFF8020I	CD-ROM device, other mass storage devices
0x03	QIC-157	Tape device
0x04	UFI	Floppy disk device
0x05	SFF8070I	Floppy disk device, other mass storage devices
0x06	SCSI transparent command set	Any device that complies with a SCSI-defined command set

The `bInterfaceProtocol` field in the interface descriptor denotes the transport protocol the interface uses. The `IOUSBMassStorageClass` driver supports the interface protocols shown in Table 3-2 (page 36).

Table 3-2 USB mass storage class protocols

Protocol code	Protocol implementation
0x0	Control/Bulk/Interrupt protocol *with* command completion interrupt
0x01	Control/Bulk/Interrupt protocol *without* command completion interrupt
0x50	Bulk-only transport

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

If the device is mass storage class compliant, one of the `IOUSBMassStorageClass` driver's personalities matches on the device's interface subclass. Listing 3-3 (page 37) shows the first personality in the `IOUSBMassStorageClass` driver's personality dictionary.

Listing 3-3 One of the IOUSBMassStorageClass driver's personalities

```
<dict>
    <!-- CFBundleIdentifier denotes the name of the driver in
     -- reverse DNS notation. -->
    <key>CFBundleIdentifier</key>
    <string>com.apple.iokit.IOUSBMassStorageClass</string>

    <!-- IOUSBMassStorageClass is the name of the class the I/O Kit
     -- instantiates. -->
    <key>IOClass</key>
    <string>IOUSBMassStorageClass</string>

    <!-- IOUSBInterface is the name of the nub class this
     -- personality attaches to. -->
    <key>IOProviderClass</key>
    <string>IOUSBInterface</string>

    <!-- The next two keys describe which bus
     -- the device is on and whether it is internal
     -- or external.-->
    <key>Physical Interconnect</key>
    <string>USB</string>
    <key>Physical Interconnect Location</key>
    <string>External</string>

    <!-- The interface class this driver matches on.-->
    <key>bInterfaceClass</key>
    <integer>8</integer>

    <!-- Interface subclass 1 refers to the Reduced Block Commands
     -- subclass.-->
    <key>bInterfaceSubClass</key>
    <integer>1</integer>
```

Vendor-Specific Mass Storage Class Compliant Devices

Occasionally, a device is compliant with the USB mass storage specification but declares its device class to be vendor specific instead of mass storage. In this case, you need to induce the I/O Kit to load the `IOUSBMassStorageClass` driver for your device even though the driver matches on only mass storage class interfaces.

You do this by creating a KEXT that consists solely of an information property list that contains a personality for your device. Like any vendor-specific interface driver, this KEXT matches on the configuration value, interface number, and vendor and product numbers the `IOUSBInterface` nub supplies. Unlike most vendor-specific drivers, however, this KEXT sets its `IOClass` key to `IOUSBMassStorageClass` and includes a dictionary named "USB Mass Storage Characteristics" containing the subclass and protocol information that reflects how the device should be treated. The `IOUSBMassStorageClass` driver then uses those keys to determine the subclass and protocol of the device instead of relying on the information supplied by the device.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Listing 3-4 (page 38) shows the personality for a device that complies with the USB mass storage class specification but belongs to the vendor-specific class.

Listing 3-4 Partial listing of an Info.plist file for a vendor-specific device

```
<dict>
    <key>CFBundleIdentifier</key>
    <string>com.apple.iokit.IOUSBMassStorageClass</string>

    <!-- IOUSBMassStorageClass is the name of the class the
     -- I/O Kit instantiates when probing. -->
    <key>IOClass</key>
    <string>IOUSBMassStorageClass</string>

    <!-- IOUSBInterface is the name of the nub class
    < -- the driver attaches to. -->
    <key>IOProviderClass</key>
    <string>IOUSBInterface</string>

    <!-- The following two keys describe
     -- which bus the device is on and whether it
     -- is internal or external. -->
    <key>Physical Interconnect</key>
    <string>USB</string
    <key>Physical Interconnect Location</key>
    <string>External</string>

    <key>USB Mass Storage Characteristics</key>
    <dict>
        <!-- The bInterfaceProtocol value is Control/Bulk/Interrupt
         -- with command completion interrupt. -->
        <key>Preferred Protocol</key>
        <integer>1</integer>
        <!-- The bInterfaceSubclass value is SFF8070I. -->
        <key>Preferred Subclass</key>
        <integer>5</integer>
    </dict>

    <!-- The following four keys are used for interface
     -- matching. -->
    <key>bConfigurationValue</key>
    <integer>MyConfigurationValue</integer>
    <key>bInterfaceNumber</key>
    <integer>MyInterfaceNumber</integer>
    <key>idProduct</key>
    <integer>MyProductID</integer>
    <key>idVendor</key>
    <integer>MyVendorID</integer>
</dict>
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Matching for a Subclass of the USB Protocol Services Driver

If your device is not compliant with the USB mass storage class specification, you need to develop a subclass of the `IOUSBMassStorageClass` driver to support the differences. In order to ensure that your driver loads in favor of the generic `IOUSBMassStorageClass` driver you must use the keys defined for interface matching in the Universal Serial Bus Common Class Specification, Revision 1.0 (available for download from http://www.usb.org/developers/devclass_docs/usbccs10.pdf.)

The interface-matching keys defined in the USB Common Class Specification consist of specific combinations of property keys. For a successful match, you must include the property keys defined by exactly one interface-matching key in your `Info.plist` file. If you use a combination of property keys *not* defined by any interface-matching key, your driver will not match. For example, if you use the property keys for vendor, product, and interface protocol, your driver will not match. This is because there is no key that combines the property keys of vendor, product, and interface protocol.

Table 3-3 (page 39) lists the keys in order of specificity: The interface-matching key for the most specific match (and highest probe score) is listed first.

Table 3-3 Interface-matching keys from the USB Common Class Specification

Interface-matching key	Comment
`idVendor` + `idProduct` + `bInterfaceNumber` + `bConfigurationValue` + `bcdDevice`	
`idVendor` + `idProduct` + `bInterfaceNumber` + `bConfigurationValue`	
`idVendor` + `bInterfaceSubClass` + `bInterfaceProtocol`	Only if `bInterfaceClass` is0xFF
`idVendor` + `bInterfaceSubClass`	Only if `bInterfaceClass` is 0xFF
`bInterfaceClass` + `bInterfaceSubClass` + `bInterfaceProtocol`	Only if `bInterfaceClass` is not 0xFF
`bInterfaceClass` + `bInterfaceSubClass`	Only if `bInterfaceClass` is not 0xFF

The ATAPI Protocol Services Driver

When an ATAPI mass storage device is discovered, the ATAPI bus controller publishes an ATA device nub that is a concrete subclass of `IOATADevice`. The ATA family defines no family-specific matching so all matching is active. This means the driver probes the device to determine if it is suited to drive it.

In its `start` method during active matching, the `IOATAPIProtocolTransport` driver compares the properties in its personality to the device's properties. In particular, if the device's ATA device type is ATAPI, the driver loads for that device. Listing 3-5 (page 39) shows the personality for the `IOATAPIProtocolTransport` driver.

Listing 3-5 The IOATAPIProtocolTransport driver personality dictionary

```
<dict>
    <!-- CFBundleIdentifier denotes the name of the driver in
     -- reverse DNS notation. -->
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
    <key>CFBundleIdentifier</key>
    <string>com.apple.iokit.IOATAPIProtocolTransport</string>

    <!-- IOATAPIProtocolTransport is the class the I/O Kit
     -- instantiates when probing. -->
    <key>IOClass</key>
    <string>IOATAPIProtocolTransport</string>

    <!-- IOATADevice is the nub this driver attaches to. -->
    <key>IOProviderClass</key>
    <string>IOATADevice</string>

    <!-- The next two keys describe which bus the
     -- device is on and whether it is internal
     -- or external.-->
    <key>Physical Interconnect</key>
    <string>ATAPI</string>
    <key>Physical Interconnect Location</key>
    <string>Internal<string>

    <!-- The value of this key is compared to the ATA device type
     -- of the device. -->
    <key>ata device type</key>
    <string>atapi</string>
</dict>
```

A subclass of the IOATAPIProtocolTransport driver should include the same keys shown in Listing 3-5 (page 39) in its personality dictionary. If you need to address ATAPI-specific issues such as a device that needs to do a hard reset after a particular event, you need to develop a subclass of IOATAPIProtocolTransport that overrides the appropriate methods. To ensure that your subclass driver loads, you should implement the probe method and increase the probe score after determining that your driver can, in fact, drive the device.

In order to change a device's DMA or UDMA modes, however, you can take advantage of a feature in the IOATAPIProtocolTransport driver that allows a subclass to override the mode the device reports. You enable this feature by creating a KEXT that consists of an Info.plist file containing a dictionary named "ATAPI Mass Storage Characteristics" in addition to the keys shown in Listing 3-5 (page 39). This dictionary contains the device model name and the DMA and UDMA mode values you choose. The device model name is the string the device returns when it responds to the ATA Identify command. The DMA and UDMA mode values are bitmasks defined in the ATA/ATAPI-5 specification (available at http://www.t13.org). Listing 3-6 (page 40) shows an example personality dictionary that overrides the DMA and UDMA values returned by the device.

Listing 3-6 A personality dictionary that overrides DMA and UDMA values

```
<dict>
    <key>ATAPI Mass Storage Characteristics</key>
    <dict>
        <key>DMA Mode</key>
        <integer>0</integer>
        <key>UDMA Mode</key>
        <integer>0</integer>
        <key>device model</key>
        <string>MyDeviceModel</string>
    </dict>

    <key>CFBundleIdentifier</key>
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
    <string>com.apple.iokit.IOATAPIProtocolTransport</string>

    <key>IOClass</key>
    <string>IOATAPIProtocolTransport</string>

    <key>IOProbeScore</key>
    <integer>5000</integer>

    <key>IOProviderClass</key>
    <string>IOATADevice</string>

    <key>Physical Interconnect</key>
    <string>ATAPI</string>

    <key>Physical Interconnect Location</key>
    <string>Internal</string>

    <key>ata device type</key>
    <string>atapi</string>
</dict>
```

In this example, when this device is discovered, the I/O Kit allows all KEXTs with the key-value pair

```
<key>ata device type</key>
<string>atapi</string>
```

to probe the device. If a KEXT's personality contains the ATAPI Mass Storage Characteristics dictionary, the I/O Kit compares the value of the `device model` string with the device model name reported by the device. If they match, the I/O Kit loads the `IOATAPIProtocolTransport` driver and applies the DMA and UDMA overrides declared in the ATAPI Mass Storage Characteristics dictionary.

Logical Unit Driver Matching

After the protocol services driver loads, the peripheral device nub queries the device and publishes its device type in the I/O Registry. The I/O Kit then finds and loads the appropriate logical unit driver for the device. Unlike the bus-specific perspective of the protocol services drivers, the logical unit drivers view a mass storage device in terms of its device type as defined by the SCSI Architecture Model specifications. Thus, all four in-kernel logical unit drivers use the same matching language.

The following four properties in the personality of a logical unit driver determine which devices or device types the driver is suited for:

- The peripheral device type property
- The vendor property
- The product or model property
- The product or software revision property

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

These four properties vary from the general to the specific. Each specified property narrows the range of devices the driver is suitable for. The more properties the driver includes in its personality, the more specific the device it is suited to manage. The presence of the more specific properties does not make up for the absence of the peripheral device type property, however. If you do not include the peripheral device type property in your logical unit driver's personality, it will not be considered for loading.

During the passive matching phase, the properties are examined in the order listed and the driver's probe score is increased with each match. Every property in the driver's personality must match in order for the driver to be considered a candidate for the device. In other words, if a driver specifies a property, it must match for that driver to be considered. If one of the more specific properties is absent, however, it does not affect the probe score because that means the driver can manage a broader range of devices.

For example, of the four listed properties, the `IOSCSIPeripheralDeviceType00` driver lists only the peripheral device type property in its personality because it can drive any device that complies with the SCSI Architecture Model specification for block storage devices. The personality dictionary for the `IOSCSIPeripheralDeviceType00` driver is shown in Listing 3-7 (page 42).

Listing 3-7 The IOSCSIPeripheralDeviceType00 driver personality dictionary

```
<dict>
    <!-- The CFBundleIdentifier gives the name of this KEXT in
     -- reverse-DNS notation. -->
    <key>CFBundleIdentifier</key>
    <string>com.apple.iokit.IOSCSIBlockCommandsDevice</string>

    <!-- IOSCSIPeripheralDeviceType00 is the name of the class
     -- the I/O Kit instantiates. -->
    <key>IOClass</key>
    <string>IOSCSIPeripheralDeviceType00</string>

    <!-- IOSCSIPeripheralDeviceNub is the name of the nub
     -- class this personality attaches to. -->
    <key>IOProviderClass</key>
    <string>IOSCSIPeripheralDeviceNub</string>

    <!-- This personality is suited to drive devices of peripheral
     -- device type 0. ->
    <key>Peripheral Device Type</key>
    <integer>0</integer>
</dict>
```

If you subclass a logical unit driver to address a device's differently implemented command or added feature, you must ensure that your driver's probe score is higher than that of a more generic driver. To do this, you add as many of the four logical unit driver personality properties as necessary to uniquely identify your device.

For example, a driver can use both vendor and product information to ensure that it gets loaded in favor of one of the Apple-provided logical unit drivers. Listing 3-8 (page 42) shows the personality dictionary for a driver competing with the `IOSCSIPeripheralDeviceType05` driver.

Listing 3-8 Example logical unit driver personality dictionary

```
<dict>
    <!-- The CFBundleIdentifier value is the name of
     -- this KEXT. -->
    <key>CFBundleIdentifier</key>
    <string>com.MySoftwareCompany.driver.MyLogicalUnitDriver</string>
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
    <!-- The IOClass value is the name of the class
     -- the I/O Kit instantiates. -->
    <key>IOClass</key>
    <string>com_MySoftwareCompany_iokit_MyLogicalUnitDriver</string>

    <!-- The IOProviderClass value is the name of the
     -- nub this driver attaches to. -->
    <key>IOProviderClass</key>
    <string>IOSCSIPeripheralDeviceNub</string>

    <!-- The next three keys determine the device this
     -- driver is suited to drive. -->
    <key>Peripheral Device Type</key>
    <integer>5</integer>

    <key>Product Identification</key>
    <string>MyProductID</string>

    <key>Vendor Identification</key>
    <string>MyVendorID</string>
</dict>
```

Filter-Scheme Driver Matching

After the logical unit driver loads, it publishes the appropriate device services nub with the device's type in the I/O Registry. The I/O Kit initiates matching on this nub object and finds the appropriate generic block storage driver. The block storage driver then publishes an `IOMedia` object that represents the whole device. If the disk is Apple-formatted, the `IOApplePartitionScheme` matches on the `IOMedia` object and publishes new `IOMedia` objects for each partition.

Filter-scheme drivers must match on the properties the `IOMedia` objects publish. The standard set of properties for `IOMedia` objects include the following:

Table 3-4 IOMedia properties

Key	Type	Description
`kIOMediaEjectableKey`	Boolean	Is the media ejectable?
`kIOMediaPreferred-BlockSizeKey`	Number	The media's natural block size in bytes.
`kIOMediaSizeKey`	Number	The media's entire size in bytes.
`kIOMediaWholeKey`	Boolean	Is the media at the root of the media tree? This is true for the physical media representation, a RAID media representation, etc.
`kIOMediaWritableKey`	Boolean	Is the media writable?

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Key	Type	Description
kIOMediaContentHintKey	String	The media's content description, as hinted at the time of the object's creation. The string is of the *MyCompany_MyContent* format.
kIOBSDNameKey	String	The media's BSD device node name, which is dynamically assigned at the object's creation.

You can choose any subset of these properties to include in your driver's personality dictionary, but all properties in the personality must match for your driver to be considered for loading.

The kIOMediaContentHintKey is the most useful property because it matches on the unique content-hint string your disk utility program places in the media's partition (for more information on this process, see "Construction of a Mass Storage Driver Stack" (page 24)). You define the content-hint string your disk utility program uses, you place the same content-hint string in the kIOMediaContentHintKey property of your driver's personality, and your filter-scheme driver is the only candidate to match on that media.

Unlike the SCSI Architecture Model family, the Storage family does not increase a driver's probe score with each successful property match during the passive matching phase. If a filter-scheme driver's personality matches successfully on an IOMedia object's properties, the I/O Kit allows it to probe the media during the active matching phase. If the filter-scheme driver implements its own probe method, it can increase or decrease its probe score according to its ability to drive the media. However, because the filter-scheme driver that matches on the content-hint string is almost certainly the only driver candidate, it is seldom necessary to override the probe method. By default, the probe method returns true and the active matching phase ends as the I/O Kit chooses the one filter-scheme driver that matched on the content-hint string property.

If you develop your own filter-scheme driver, you must ensure that your driver's personality can match on your content as identified by your content-hint string. Listing 3-9 (page 44) shows the personality dictionary of a filter-scheme driver that matches on the content-hint string MySoftwareCompany_MyContent.

Listing 3-9 Example filter-scheme driver personality

```
<dict>
    <key>IOStorage</key>
    <dict>
        <!-- The CFBundleIdentifier gives the name of this KEXT in
         -- reverse-DNS notation. -->
        <key>CFBundleIdentifier</key>
        <string>com.MySoftwareCompany.driver.MyFilterScheme</string>

        <!-- The Content Hint value must be identical to the content hint
         -- string your disk utility program places in the partition. -->
        <key>Content Hint</key>
        <string>MySoftwareCompany_MyContent</string>

        <!-- The IOClass value is the name of the class
         -- the I/O Kit instantiates. -->
        <key>IOClass</key>
        <string>com_MySoftwareCompany_driver_MyFilterScheme</string>

        <!-- The IOMatchCategory key is a special key that allows
         -- multiple clients to match on an IOMedia object. -->
        <key>IOMatchCategory</key>
        <string>IOStorage</string>
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
        <!-- The IOProviderClass is the name of the
         -- nub this driver attaches to. -->
        <key>IOProviderClass</key>
        <string>IOMedia</string>
    </dict>
</dict>
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Developing a Universal Binary

If you plan to create a universal binary version of your logical unit driver, protocol services driver, or filter scheme, first read *Universal Binary Programming Guidelines, Second Edition*. That document covers architectural differences and byte-ordering formats and provides comprehensive guidelines for code modification and building universal binaries. Then, to find out how to decide which compiler version and SDK you need, see "Developing a Device Driver to Run on an Intel-Based Macintosh" in *I/O Kit Device Driver Design Guidelines*.

This chapter briefly outlines a few of the mass storage–specific issues you should keep in mind as you create a universal binary version of your driver or filter scheme.

Creating a Logical Unit or Protocol Services Driver Universal Binary

As you create a universal binary version of your logical unit or protocol services driver, be aware of places in your code where you might make assumptions about the byte ordering of multibyte numerical values. Be sure to replace any hard-coded byte swaps (such as code that always swaps a multibyte value from big endian to little endian) with the appropriate conditional byte-swapping macros defined in `libkern/OSByteOrder.h`.

For example, the Apple-provided `IOSCSIBlockCommandsDevice` class contains code that uses a byte-swapping macro defined in `OSByteOrder.h` to swap two four-byte fields in a `SCSI_Capacity_Data` structure, as shown in Listing 4-1. (The `SCSI_Capacity_Data` structure is defined as the capacity return structure for the `READ_CAPACITY 10` command in the `SCSICmds_READ_CAPACITYDefinitions.h` header file.)

Listing 4-1 Byte-swapping in IOSCSIBlockCommandsDevice code

```
bool IOSCSIBlockCommandsDevice::DetermineMediumCapacity (UInt64 * blockSize,
UInt64 * blockCount) {
SCSI_Capacity_Data    capacityData = {0};
...
*blockSize = 0;
*blockCount = 0;
...
// Create and send READ_CAPACITY command.
// If the command completed successfully:
*blockSize = OSSwapBigToHostInt32 (capacityData.BLOCK_LENGTH_IN_BYTES);
*blockCount = ((UInt64) OSSwapBigToHostInt32
(capacityData.RETURNED_LOGICAL_BLOCK_ADDRESS)) + 1;
...
}
```

In general, data returned from devices that comply with the SCSI Architecture Model specifications is in the big-endian format. Fortunately, however, the SCSI command model specification defines the CDB (command descriptor block) as a byte array. This means that the bytes are stored in the defined order regardless of the native endian format of the computer the driver is running in.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Creating a Filter Scheme Universal Binary

As you create a universal binary version of your filter-scheme driver, be aware that filter schemes frequently handle data structures that are read from or written to disk. It's essential that the data structure on the disk remain in the correct endian format so the disk can be used with both PowerPC-based and Intel-based Macintosh computers. Depending on the native endian format of the computer in which your filter-scheme driver is running, therefore, your driver may need to byte swap the data structures it handles.

If you've determined that byte-swapping is necessary, you can implement it in either of the following two ways:

- Perform the appropriate byte swap in memory when the data structure is read in from the disk and perform the opposite byte swap when the data structure is written out to the disk. This means your driver can access the data structure in memory without having to worry about the data structure's endian format.
- Do not swap the endian format of the data structure while it is in memory, but perform the appropriate byte swap on each access. This keeps the data structure in the correct endian format for the disk while it resides in memory, which means your driver does not have to byte swap the data structure when reading it in or writing it out.

To avoid confusion, it's best to choose only one of these two alternatives and be consistent in its implementation. Whichever option you choose, however, be sure to use the conditional byte-swapping macros defined in `libkern/OSByteOrder.h`. When you use these macros, the compiler optimizes your code so the routines are executed only if they are necessary for the architecture in which your driver is running.

For example, the built-in Apple partition-scheme driver, `IOApplePartitionScheme`, uses a byte-swapping macro defined in `OSByteOrder.h` to swap a two-byte field in a `dpme` structure, as shown in Listing 4-2. (The `dpme` structure is defined as a disk partition map entry in the `IOApplePartitionScheme.h` header file.)

Listing 4-2 Byte-swapping in IOApplePartitionScheme code

```
OSSet * IOApplePartitionScheme::scan (SInt32 * score) {
...
dpme *     dpmeMap = 0;
...
// Read in a partition entry and assign to dpmeMap.
...
// Determine whether the partition entry signature is present.
if (OSSwapBigToHostInt16(dpmeMap->dpme_signature) != DPME_SIGNATURE)
    ...
}
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Subclassing Logical Unit Drivers

The SCSI Architecture Model defines several shared command set specifications, each associated with a peripheral device type. These specifications document how SCSI commands are processed by a device's firmware. If your device does not conform with the shared command set specification for its peripheral device type in some way, either because it processes commands differently or because it services additional commands, you need to subclass the appropriate Apple logical unit driver to provide the support your device requires.

Important: Do not send `READ` or `WRITE` commands from your custom logical unit driver to a device. If you send these commands, you open a security hole that malicious code can take advantage of by using your driver to read or destroy data on a device that the user has protected by setting access permissions.

This chapter describes how to subclass an Apple-provided logical unit driver to address SCSI command set implementation issues. The sample code in this chapter is generic and emphasizes the form your driver should take, rather than the code required to implement a specific command. Because the sample drivers are generic, they will not attach to a particular device. To test them with your device, replace the generic values for parameters such as vendor or product identification with values that identify your device. For more information on how to develop kernel extensions in general and I/O Kit drivers in particular, see *Kernel Extension Programming Topics* and *I/O Kit Device Driver Design Guidelines*.

This chapter also contains code that shows how your driver can use a `SCSITask` object to send a command to a device and how to use the SCSI Architecture Model family's command-builder functions to build a custom CDB.

Important: The sample code in this chapter is designed to work with Mac OS X version 10.1 and later. It will not work with earlier versions.

Setting Up Your Project

This section describes how to create your driver project and edit your driver's information property list (`Info.plist` file). The sample driver in this chapter is a logical unit driver for a generic CD-ROM device so it is a subclass of the Apple-provided `IOSCSIPeripheralDeviceType05` driver.

The sample project uses `MyLogicalUnitDriver` for the name of the driver and generic values such as `MySoftwareCompany` for the developer name. You should replace these names and values with your own information in order to test this code with your device.

Create a New Project

Open the Xcode application and create a new I/O Kit driver project named `MyLogicalUnitDriver`. Specify a directory for the new project or accept the default.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

When you create a new I/O Kit driver project, Xcode supplies several files, including two empty source files—MyLogicalUnitDriver.h and MyLogicalUnitDriver.cpp. Before you add any code to these files, however, you should edit your driver's information property list.

Edit Your Driver's Property List

Every driver has an Info.plist file that contains information about the driver and what it needs, including its personalities. As described in "Driver Personalities and the Matching Process" (page 31), a driver's personality contains the matching information the I/O Kit uses to determine the appropriate driver for a device. To make sure your driver loads for your device, you add several properties to its personality dictionary that identify the device or type of device it supports.

In Xcode, a driver's Info.plist file is listed in the Groups & Files view in the project. You can edit the property list file as plain XML text in the Xcode editor window or you can choose a different application (such as Property List Editor) to use. For more information on how to select another editor, see Hello I/O Kit: Creating a Driver With Xcode.

The IOKitPersonalities dictionary in the driver's Info.plist file can contain multiple personality dictionaries, one for each device or type of device your driver supports. The sample driver in this chapter implements only one personality dictionary but you can create additional dictionaries if your driver can support more than one device or device type.

The sample code uses the following six property keys:

- CFBundleIdentifier
- IOClass
- IOProviderClass
- Peripheral Device Type
- Vendor Identification
- Product Identification

If you are developing a driver for a particular version of your device, you can add the product revision identification key to the personality for even more specific matching.

Using your chosen editing environment, create a new child of the IOKitPersonalities dictionary. Make the name of this new child MyLogicalUnitDriver and set its class to Dictionary.

Create six new children of the MyLogicalUnitDriver dictionary, one for each of the six properties you'll be adding. Table 5-1 (page 50) shows the properties, along with their classes and values. To test the sample code with your device, replace values such as MyProductIdentification with actual values for your device.

Table 5-1 Personality properties for MyLogicalUnitDriver

Property	Class	Value
CFBundleIdentifier	String	com.MySoftwareCompany.driver.MyLogicalUnitDriver
IOClass	String	com_MySoftwareCompany_driver_MyLogicalUnitDriver

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Property	Class	Value
IOProviderClass	String	IOSCSIPeripheralDeviceNub
Peripheral Device Type	Number	5
Vendor Identification	String	MyProductIdentification
Product Identification	String	MyVendorIdentification

A driver declares its dependencies on other loadable kernel extensions and in-kernel components in the `OSBundleLibraries` dictionary. Each dependency has a string value that declares the earliest version of the dependency the driver is compatible with.

The sample driver depends on two loadable extensions from the `IOSCSIArchitectureModel` family. To add these dependencies to the `OSBundleLibraries` dictionary, you create a new child for each dependency. Table 5-2 (page 51) shows the dependencies you add for the sample driver.

Table 5-2 Dependencies for MyLogicalUnitDriver

Property	Class	Value
com.apple.iokit.IOSCSIArchitectureModelFamily	String	1.0.0
com.apple.iokit.IOSCSIMultimediaCommandsDevice	String	1.0.0

Because the driver of a CD-ROM drive must be able to mount root on a local volume, you add the `OSBundleRequired` property to the top level of its `Info.plist` file. In other words, the new `OSBundleRequired` property is a sibling of the `IOKitPersonalities` and `OSBundleLibraries` dictionaries, not a child. Edit the new element to match the following:

```
OSBundleRequired        String   Local-Root
```

Creating Your Driver

This section describes some of the elements that must be included in your driver's source files. To demonstrate the process of subclassing, the sample driver simply overrides the `GetConfiguration` method and prints a message. You should replace this trivial function with your own code that supports your device's particular command implementations.

In Xcode, the driver's source files are listed in the Groups & Files pane, revealed by the discosure triangle next to the `MyLogicalUnitDriver` project and the disclosure triangle next to the Source folder.

Edit the Header File

The header file provides access to external declarations and supporting type definitions needed by the functions and objects in the C++ file. The header for the sample driver is simple because it includes only one method declaration. Edit the `MyLogicalUnitDriver.h` file to match the code in Listing 5-1 (page 52).

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Listing 5-1 The MyLogicalUnitDriver header file

```
#ifndef _MyLogicalUnitDriver_H_
#define _MyLogicalUnitDriver_H_

// Because the sample driver is a subclass of the Apple-provided
// peripheral device type 05 driver, it must include that driver's
// header file.
#include <IOKit/scsi-commands/IOSCSIPeripheralDeviceType05.h>

// Here, the sample driver declares its inheritance and the method
// it overrides.
class com_MySoftwareCompany_driver_MyLogicalUnitDriver : public
        IOSCSIPeripheralDeviceType05
{
    OSDeclareDefaultStructors (
                com_MySoftwareCompany_driver_MyLogicalUnitDriver )
protected:
    virtual IOReturn GetConfiguration ( void );
};

#endif /* _MyLogicalUnitDriver_H_ */
```

Edit the C++ File

The C++ file provides the code to override the chosen methods. The sample driver's C++ file contains all the elements required for a subclassed driver even though it accomplishes nothing more substantial than a message sent to the system log file, `/var/log/system.log`.

Edit the `MyLogicalUnitDriver.cpp` file to match the code in Listing 5-2 (page 52).

Listing 5-2 The MyLogicalUnitDriver C++ file

```
// Include the header file you created
#include "MyLogicalUnitDriver.h"

// This definition allows you to use the more convenient "super" in
// place of "IOSCSIPeripheralDeviceType05", where appropriate.
#define super IOSCSIPeripheralDeviceType05

// This macro must appear before you define any of your class's methods.
// Note that you must use the literal name of the superclass here, not
// "super" as defined above.
OSDefineMetaClassAndStructors (
        com_MySoftwareCompany_driver_MyLogicalUnitDriver,
        IOSCSIPeripheralDeviceType05 );

// Define the method to override.
IOReturn
com_MySoftwareCompany_driver_MyLogicalUnitDriver::GetConfiguration ( void )
{
    IOLog( "MyLogicalUnitDriver overriding GetConfiguration\n" );
// You can add code that accesses your device here.
// Call super's GetConfiguration method before returning.
    return super::GetConfiguration();
}
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Testing Your Driver

This section presents some advice on testing your driver. You cannot use `kextload` to load and test your driver "by hand" because there are generic drivers that will always load in its place at boot time. Therefore, you need to make sure you have multiple bootable disks or partitions so you can remove your driver if it behaves badly and reboot the disk or partition.

Because the `OSBundleRequired` property in the sample driver's `Info.plist` file is set to `Local-Root`, the BootX booter will automatically load it when the system is restarted (for more information on this process, see Loading Kernel Extensions at Boot Time).

For help with debugging, you can open a window in the Terminal application (located at `/Applications/Utilities/Terminal`) and type the following line to view the system log:

```
tail -f /var/log/system.log
```

Creating and Sending SCSI Commands

As described in "The Transport Driver Layer" (page 14), a logical unit driver subclass creates a `SCSITask` object to contain a command descriptor block (or CDB) and various status indicators related to the execution of a SCSI command. To create a command to put into a `SCSITask` object, you can use the built-in command-creation functions or you can build a custom CDB. The built-in command-creation functions are appropriate when you need to send standard SCSI commands, such as `INQUIRY`, `TEST_UNIT_READY`, and `REPORT_SENSE`. When you need to send a vendor-specific SCSI command, you use the `SetCommandDescriptorBlock` function to build an appropriately sized CDB. Note that `SetCommandDescriptorBlock` and the built-in command-creation functions are defined in `IOSCSIPrimaryCommandsDevice.h`. The sample code in this section shows both ways to build and send SCSI commands.

The code in this section shows a simple logical unit driver for a block commands device, specifically, a subclass of `IOSCSIPeripheralDeviceType00`. It shows how to use one of the built-in command-creation functions and it demonstrates how to set up your own command-creation function to build a custom command. It also shows how to check the command response and the status of the `SCSITask` object. Listing 5-3 shows the header file for the sample driver.

Listing 5-3 Header file for a driver that sends standard and custom SCSI commands

```
#ifndef _SampleINQUIRYDriver_H_
#define _SampleINQUIRYDriver_H_
#include <IOKit/scsi/IOSCSIPeripheralDeviceType00.h>

class com_MySoftwareCompany_driver_SampleINQUIRYDriver : public
IOSCSIPeripheralDeviceType00
{
    OSDeclareDefaultStructors ( com_MySoftwareCompany_driver_SampleINQUIRYDriver )

protected:
    bool    InitializeDeviceSupport ( void );
    void    SendBuiltInINQUIRY ( void );
    void    SendCreatedINQUIRY ( void );
    bool    BuildINQUIRY ( SCSITaskIdentifier    request,
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
                                IOBufferMemoryDescriptor *    buffer,
                                SCSICmdField1Bit      CMDDT,
                                SCSICmdField1Bit      EVPD,
                                SCSICmdField1Byte     PAGE_OR_OPERATION_CODE,
                                SCSICmdField1Byte     ALLOCATION_LENGTH,
                                SCSICmdField1Byte     CONTROL );
};
#endif /* _SampleINQUIRYDriver_H_ */
```

Listing 5-4 shows the implementation of the sample driver. Although there is some error handling shown, you should add more extensive error-handling code when you use this sample as the basis for an actual driver.

Note: The sample driver's custom command-building function builds an `INQUIRY` command instead of a hypothetical custom command. In a real driver, you use a built-in command-building function to build a standard command such as `INQUIRY`; you do not write custom functions to build standard commands.

Listing 5-4 Implementation of a driver that sends standard and custom SCSI commands

```
#include <IOKit/IOBufferMemoryDescriptor.h>
#include <IOKit/scsi/SCSICmds_INQUIRY_Definitions.h>
#include <IOKit/scsi/SCSICommandOperationCodes.h>
#include <IOKit/scsi/SCSITask.h>
#include "SampleINQUIRYDriver.h"
#define super IOSCSIPeripheralDeviceType00

OSDefineMetaClassAndStructors ( com_MySoftwareCompany_driver_SampleINQUIRYDriver,
IOSCSIPeripheralDeviceType00 );

bool
com_MySoftwareCompany_driver_SampleINQUIRYDriver::InitializeDeviceSupport ( void )
{
    bool    result = false;
    result = super::InitializeDeviceSupport ( );
    if ( result == true ) {
        SendBuiltInINQUIRY ( );
        SendCreatedINQUIRY ( );
    }
    return result;
}

void
com_MySoftwareCompany_driver_SampleINQUIRYDriver::SendBuiltInINQUIRY ( void )
{
    // The Service Response represents the execution status of a service request.
    SCSIServiceResponse                serviceResponse =
kSCSIServiceResponse_SERVICE_DELIVERY_OR_TARGET_FAILURE;
    IOBufferMemoryDescriptor *         buffer  = NULL;
    SCSITaskIdentifier                 request = NULL;
    UInt8 *                            ptr     = NULL;
    // Get a new IOBufferMemoryDescriptor object with a buffer large enough
    // to hold the SCSICmd_INQUIRY_StandardData structure (defined
    // in SCSICmds_INQUIRY_Definitions.h).
    buffer = IOBufferMemoryDescriptor::withCapacity ( sizeof (
SCSICmd_INQUIRY_StandardData ), kIODirectionIn, false );
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
    require ( ( buffer != NULL ), ErrorExit );

    // Get the address of the beginning of the buffer and zero-fill the buffer.
    ptr = ( UInt8 * ) buffer->getBytesNoCopy ( );
    bzero ( ptr, buffer->getLength ( ) );

    // Create a new SCSITask object; if unsuccessful, release

    request = GetSCSITask ( );
    require ( ( request != NULL ), ReleaseBuffer );

    // Prepare the buffer for an I/O transaction. This call must be
    // balanced by a call to the complete method (shown just before
    // ReleaseTask).
    require ( ( buffer->prepare ( ) == kIOReturnSuccess ), ReleaseTask );

    // Use the INQUIRY method to assemble the command. Then use the
    // SendCommand method to synchronously issue the request.

    if ( INQUIRY (  request,
                    buffer,
                    0,
                    0,
                    0x00,
                    buffer->getLength ( ),
                    0 ) == true )
    {
        serviceResponse = SendCommand ( request, kTenSecondTimeoutInMS );
    }

   // Check the SendCommand method's return value and the status of the SCSITask object.
    if ( ( serviceResponse == kSCSIServiceResponse_TASK_COMPLETE ) &&
           GetTaskStatus ( request ) == kSCSITaskStatus_GOOD )
    {
        IOLog ( "INQUIRY succeeded\n" );
    }
    else
    {
        IOLog ( "INQUIRY failed\n" );
    }

    // Complete the processing of this buffer after the I/O transaction
    // (this call balances the earlier call to prepare).
    buffer->complete ( );

// Clean up before exiting.
ReleaseTask:
    require_quiet ( ( request != NULL ), ReleaseBuffer );
    ReleaseSCSITask ( request );
    request = NULL;

ReleaseBuffer:
    require_quiet ( ( buffer != NULL ), ErrorExit );
    buffer->release ( );
    buffer = NULL;

ErrorExit:
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
    return;
}

void
com_MySoftwareCompany_driver_SampleINQUIRYDriver::SendCreatedINQUIRY ( void )
{
    SCSIServiceResponse                serviceResponse =
kSCSIServiceResponse_SERVICE_DELIVERY_OR_TARGET_FAILURE;
    IOBufferMemoryDescriptor *        buffer            = NULL;
    SCSITaskIdentifier                request            = NULL;
    UInt8 *                            ptr                = NULL;

    // Get a new IOBufferMemoryDescriptor object with a buffer large enough
    // to hold the SCSICmd_INQUIRY_StandardData structure (defined in
    // SCSICmds_INQUIRY_Definitions.h).
    buffer = IOBufferMemoryDescriptor::withCapacity ( sizeof (
SCSICmd_INQUIRY_StandardData ), kIODirectionIn, false );

    // Return immediately if the buffer wasn't created.
    require ( ( buffer != NULL ), ErrorExit );

    // Get the address of the beginning of the buffer and zero-fill the buffer.
    ptr = ( UInt8 * ) buffer->getBytesNoCopy ( );
    bzero ( ptr, buffer->getLength ( ) );

    // Create a new SCSITask object; if unsuccessful, release the buffer and return.
    request = GetSCSITask ( );
    require ( ( request != NULL ), ReleaseBuffer );

    // Prepare the buffer for an I/O transaction. This call must be
    // balanced by a call to the complete method (shown just before
    // ReleaseTask).
    require ( ( buffer->prepare ( ) == kIOReturnSuccess ), ReleaseTask );

    // The BuildINQUIRY function shows how you can design and use a
    // command-building function to create a custom command to send
    // to your device. Although the BuildINQUIRY function builds a standard INQUIRY
    // command from the passed-in values, you do not create a custom function to
    // build a standard command in a real driver. Instead, you use the SCSI
    // Architecture Model family's built-in command-building functions. The
    // BuildINQUIRY function uses INQUIRY as an example merely because
    // it is a well-understood command.
    if ( BuildINQUIRY ( request,
                        buffer,
                        0x00, // CMDDT (Command support data)
                        0x00, // EVPD  (Vital product data)
                        0x00, // PAGE_OR_OPERATION_CODE
                        buffer->getLength ( ), // ALLOCATION_LENGTH
                        0x00 )  // CONTROL
                == true)
    {
        serviceResponse = SendCommand ( request, kTenSecondTimeoutInMS );
    }
    if ( ( serviceResponse == kSCSIServiceResponse_TASK_COMPLETE ) &&
           GetTaskStatus ( request ) == kSCSITaskStatus_GOOD )
    {
        IOLog ( "Vendor-created INQUIRY command succeeded\n" );
    }
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
    else
    {
        IOLog ( "Vendor-created INQUIRY command failed\n" );
    }

    buffer->complete ( );

ReleaseTask:
    require_quiet ( ( request != NULL ), ReleaseBuffer );
    ReleaseSCSITask ( request );
    request = NULL;

ReleaseBuffer:
    require_quiet ( ( buffer != NULL ), ErrorExit );
    buffer->release ( );
    buffer = NULL;

ErrorExit:
    return;
}

bool
com_MySoftwareCompany_driver_SampleINQUIRYDriver::BuildINQUIRY (
                                            SCSITaskIdentifier    request,
                                            IOBufferMemoryDescriptor *    dataBuffer,
                                            SCSICmdField1Bit     CMDDT,
                                            SCSICmdField1Bit     EVPD,
                                          SCSICmdField1Byte     PAGE_OR_OPERATION_CODE,
                                            SCSICmdField1Byte     ALLOCATION_LENGTH,
                                            SCSICmdField1Byte     CONTROL )
{
    bool result = false;

    // Validate the parameters here.
    require ( ( request != NULL ), ErrorExit );
    require ( ResetForNewTask ( request ), ErrorExit );

    // The helper functions ensure that the parameters fit within the
    // CDB fields and that the buffer passed in is large enough for
    // the transfer length.
    require ( IsParameterValid ( CMDDT, kSCSICmdFieldMask1Bit ), ErrorExit );
    require ( IsParameterValid ( EVPD, kSCSICmdFieldMask1Bit ), ErrorExit );
    require ( IsParameterValid ( PAGE_OR_OPERATION_CODE, kSCSICmdFieldMask1Byte ),
ErrorExit );
   require ( IsParameterValid ( ALLOCATION_LENGTH, kSCSICmdFieldMask1Byte ), ErrorExit
 );
    require ( IsParameterValid ( CONTROL, kSCSICmdFieldMask1Byte ), ErrorExit );
    require ( IsMemoryDescriptorValid ( dataBuffer, ALLOCATION_LENGTH ), ErrorExit );

    // Check the validity of the PAGE_OR_OPERATION_CODE parameter, when using both the
 CMDDT and EVPD parameters.

    if ( PAGE_OR_OPERATION_CODE != 0 )
    {
        if ( ( ( CMDDT == 1 ) && ( EVPD == 1 ) ) || ( ( CMDDT == 0 ) && ( EVPD == 0 )
) )
        {
            goto ErrorExit;
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
        }
    }

    // This is a 6-byte command: fill out the CDB appropriately
    SetCommandDescriptorBlock ( request,
                                kSCSICmd_INQUIRY,
                                ( CMDDT << 1 ) | EVPD,
                                PAGE_OR_OPERATION_CODE,
                                0x00,
                                ALLOCATION_LENGTH,
                                CONTROL );

    SetDataTransferDirection ( request, kSCSIDataTransfer_FromTargetToInitiator );
    SetTimeoutDuration ( request, 0 );
    SetDataBuffer ( request, dataBuffer );
    SetRequestedDataTransferCount ( request, ALLOCATION_LENGTH );

    result = true;

ErrorExit:
    return result;
}
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Subclassing Protocol Services Drivers

The protocol services driver in the transport driver layer of the mass storage driver stack is responsible for preparing the commands it receives from the logical unit driver for transmission across a particular bus. Each supported bus defines a bus transport protocol that specifies how commands and data are sent across the bus. If your device does not comply with the bus transport protocol of the bus it's connected to, you need to subclass the appropriate Apple protocol services driver to provide the support your device requires.

Apple provides protocol services drivers for devices that comply with the following bus transport protocols:

- FireWire Serial Bus Protocol 2 specifications (see http://t10.org)
- USB mass storage class specifications (see http://www.usb.org)
- ATA/ATAPI-5 specifications (see http://t13.org)

This chapter describes how to subclass an Apple-provided protocol services driver to address bus-specific issues. The sample code in this chapter is generic and emphasizes the form your driver should take, rather than the code required to implement a specific method. Because the sample driver is generic, it will not attach to a particular device. To test it with your device, you can replace the generic values for parameters such as vendor or product identification with values that identify your device.

The sample code in this chapter is from a Xcode project that builds an I/O Kit driver. For more information on how to develop kernel extensions in general and I/O Kit drivers in particular, see *Kernel Extension Programming Topics* and *I/O Kit Device Driver Design Guidelines*.

Important: The sample code in this chapter is designed to work with Mac OS X version 10.1 and later. It will not work with earlier versions.

Setting Up Your Project

This section describes how to create your driver project and edit your driver's information property list. The sample driver in this chapter is a protocol services driver for a generic FireWire SBP-2 CD-ROM device so it is a subclass of the Apple-provided `IOFireWireSerialBusProtocolTransport` driver.

The sample project uses `MyFWProtocolServicesDriver` for the name of the driver and generic values such as `MySoftwareCompany` for the developer name. You should replace these names and values with your own information in order to test this code with your device.

Create a New Project

Open the Xcode application and create a new I/O Kit driver project named `MyFWProtocolServicesDriver`. Specify a directory for the new project or accept the default.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

When you create a new I/O Kit driver project, Xcode supplies several files, including two empty source files—`MyFWProtocolServicesDriver.h` and `MyFWProtocolServicesDriver.cpp`. Before you add any code to these files, however, you should edit your driver's information property list.

Edit Your Driver's Property List

Every driver has an information property list (`Info.plist` file) that contains information about the driver and what it needs, including its personalities. As described in "Driver Personalities and the Matching Process" (page 31), a driver's personality contains the matching information the I/O Kit uses to determine the appropriate driver for a device. To make sure your driver loads for your device, you add several properties to its personality dictionary that identify the device or type of device it supports.

In Xcode, a driver's `Info.plist` file is listed in the Groups & Files view in the project. You can edit the property list file as plain XML text in the Xcode editor window or you can choose a different application (such as Property List Editor) to use. For more information on how to select another editor, see Hello I/O Kit: Creating a Driver With Xcode.

The `IOKitPersonalities` dictionary in the driver's `Info.plist` file can contain multiple personality dictionaries, one for each device or type of device your driver supports. The sample driver in this chapter implements only one personality dictionary but you can create additional dictionaries if your driver can support more than one device or device type.

The sample code uses the following ten property keys:

- `CFBundleIdentifier`
- `Command_Set`
- `Command_Spec_ID`
- `Device_Type`
- `IOClass`
- `IOProbeScore`
- `IOProviderClass`
- `Physical Interconnect`
- `Physical Interconnect Location`
- `Vendor_ID`

Using your chosen editing environment, create a new child of the `IOKitPersonalities` dictionary. Make the name of this new child `MyFWProtocolServicesDriver` and set its class to Dictionary.

Create ten new children of the `MyFWProtocolServicesDriver` dictionary, one for each of the ten properties you'll be adding. Table 6-1 (page 61) shows the properties, along with their classes and values. To test the sample code with your device, replace values such as `MyCommandSetNumber` with actual values for your device.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Table 6-1 Personality properties for MyFWProtocolServicesDriver

Property	Class	Value
CFBundleIdentifier	String	com.MySoftwareCompany.driver.My-FWProtocolServicesDriver
Command_Set	Number	MyCommandSetNumber
Command_Spec_ID	Number	MyCommandSpecIDNumber
Device_Type	Number	MyDeviceTypeNumber
IOClass	String	com_MySoftwareCompany_driver_-MyFWProtocolServicesDriver
IOProbeScore	Number	MyProbeScore
IOProviderClass	String	IOFireWireSBP2LUN
Physical Interconnect	String	FireWire
Physical Interconnect Location	String	External
Vendor_ID	Number	MyVendorID

A driver declares its dependencies on other loadable kernel extensions and in-kernel components in the OSBundleLibraries dictionary. Each dependency has a string value that declares the earliest version of the dependency the driver is compatible with.

The sample driver depends on loadable extensions from the IOSCSIArchitectureModel family and the IOFireWire family. To add these dependencies to the OSBundleLibraries dictionary, you create a new child for each dependency. Table 6-2 (page 61) shows the dependencies you add for the sample driver.

Table 6-2 Dependencies for MyFWProtocolServicesDriver

Property	Class	Value
com.apple.iokit.IOSCSIArchitectureModelFamily	String	1.0.0
com.apple.iokit.IOFireWireFamily	String	1.0.0
com.apple.iokit.IOFireWireSBP2	String	1.0.0
com.apple.iokit.IOFireWireSerialBusProtocolTransport	String	1.0.0

Because the driver of a CD-ROM drive must be able to mount root on a local volume, you add the OSBundleRequired property to the top level of its Info.plist file. In other words, the new OSBundleRequired property is a sibling of the IOKitPersonalities and OSBundleLibraries dictionaries, not a child. Edit the new element to match the following:

```
OSBundleRequired     String      Local-Root
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Creating Your Driver

This section describes some of the elements that must be included in your driver's source files. To demonstrate the process of subclassing, the sample driver simply overrides the `init`, `start`, and `stop` methods and prints messages. You should replace these trivial functions with your own code that supports your device's particular physical interconnect transport protocol requirements.

In Xcode, the driver's source files are listed in the Groups & Files pane, revealed by the discosure triangle next to the `MyFWProtocolServicesDriver` project and the disclosure triangle next to the Source folder.

Edit the Header File

The header file provides access to external declarations and supporting type definitions needed by the functions and objects in the C++ file. The header for the sample driver is simple because it includes only method declarations and no constant or variable declarations. Edit the `MyFWProtocolServicesDriver.h` file to match the code in Listing 6-1 (page 62).

Listing 6-1 The MyFWProtocolServicesDriver header file

```
#ifndef _MyFWProtocolServicesDriver_H_
#define _MyFWProtocolServicesDriver_H_

// Because the sample driver is a subclass of the Apple-provided
// FireWire Serial Bus Protocol driver, it must include that driver's
// header file.
#include <IOKit/sbp2/IOFireWireSerialBusProtocolTransport.h>

// Here, the sample driver declares its inheritance and the method
// it overrides.
class com_MySoftwareCompany_driver_MyFWProtocolServicesDriver : public
                            IOFireWireSerialBusProtocolTransport
{
    OSDeclareDefaultStructors (
            com_MySoftwareCompany_driver_MyFWProtocolServicesDriver )
public:
    virtual bool init ( OSDictionary * propTable );

    virtual bool start ( IOService * provider );

    virtual void stop ( IOService * provider );
};

#endif /* _MyFWProtocolServicesDriver_H_ */
```

Edit the C++ File

The C++ file provides the code to override the chosen methods. The sample driver's C++ file contains all the elements required for a subclassed driver even though it accomplishes nothing more substantial than messages sent to the system log file, `/var/log/system.log`.

Edit the `MyFWProtocolServicesDriver.cpp` file to match the code in Listing 6-2 (page 63).

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Listing 6-2 The MyFWProtocolServicesDriver C++ file

```
// Include the header file you created.
#include "MyFWProtocolServicesDriver.h"

// This definition allows you to use the more convenient "super" in
// place of "IOFireWireSerialBusProtocolTransport", where appropriate.
#define super IOFireWireSerialBusProtocolTransport

// This macro must appear before you define any of your class's methods.
// Note that you must use the literal name of the superclass here, not
// "super" as defined above.
OSDefineMetaClassAndStructors (
            com_MySoftwareCompany_driver_MyFWProtocolServicesDriver,
            IOFireWireSerialBusProtocolTransport );

// Define the methods to override.
bool
com_MySoftwareCompany_driver_MyFWProtocolServicesDriver::init (
                                            OSDictionary * propTable )
{
    bool returnValue;

    IOLog ( "MyFWProtocolServicesDriver overriding init\n" );
// You can add code that initializes your device here.

// Call super's init method to make sure all other initialization is done.
    returnValue = super::init ( propTable );

    return returnValue;
}

bool com_MySoftwareCompany_iokit_MyFWProtocolServicesDriver:: start (
                                            IOService * provider )
{
    bool returnValue = false;

    IOLog ( "MyFWProtocolServicesDriver overriding start\n" );
// You can add code for your driver's start functions here.

// Call super's start method to make sure all other start functions are
// fulfilled.
    returnValue = super::start ( provider );

    return returnValue;
}

void com_MySoftwareCompany_iokit_MyFWProtocolServicesDriver::stop (
                                            IOService * provider )
{
    IOLog ( "MyFWProcolLayerDriver overriding stop\n" );
// You can add code for your driver's stop functions here.

// Call super's stop method to ensure all other necessary clean-up is done.
    super::stop ( provider );
}
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Testing Your Driver

This section presents some advice on testing your driver. You cannot use `kextload` to load and test your driver "by hand" because there are generic drivers that will always load in its place at boot time. Therefore, you need to make sure you have multiple bootable disks or partitions so you can remove your driver if it behaves badly and reboot the disk or partition.

Because the `OSBundleRequired` property in the sample driver's `Info.plist` file is set to `Local-Root`, the BootX booter will automatically load it when the system is restarted (for more information on this process, see Loading Kernel Extensions at Boot Time).

For help with debugging, you can open a window in the Terminal application (located at `/Applications/Utilities/Terminal`) and type the following line to view the system log:

```
tail -f /var/log/system.log
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Developing a Filter Scheme

On Mac OS X, a filter-scheme driver provides a filtering mechanism between generic I/O requests and content on a media. A media-filter scheme matches on an `IOMedia` object representing the content present in a partition and publishes in the I/O Registry another `IOMedia` object that represents the unfiltered content. Because filter-scheme drivers are both consumers and producers of `IOMedia` objects, there can be an arbitrary number of filter schemes in the mass storage driver stack.

To create your own filter-scheme driver, you subclass `IOStorage` and implement your filtering functionality in the `read` and `write` methods. Other methods you implement, such as `init`, `start`, and `free`, create and initialize the new `IOMedia` object, attach it to the I/O Registry, and release it.

As described in "Filter Schemes" (page 23), a filter-scheme driver should not produce an `IOCDMedia` or `IODVDMedia` object, because these objects have provider requirements specific to CD and DVD media that can be met only by an `IOCDBlockStorageDriver` or `IODVDBlockStorageDriver`, respectively.

This chapter guides you through the process of creating a filter-scheme driver. It also describes how to test the driver by creating a disk image that contains a partition the driver can match on. The sample code in this chapter is generic and emphasizes the form your driver should take, rather than the implementation of specific filtering functionality. When you use this code as a basis for your own filter-scheme driver, you should replace the generic values, such as `MySoftwareCompany`, with your own values and add your filtering code to the appropriate methods.

The sample filter scheme described in this chapter includes code that allows you to install the filter scheme on the boot partition. If you do not need to do this, you can skip the portions of the code that implement this.

The sample code in this chapter is from an Xcode project that builds a filter-scheme driver. To download the complete project (which includes debugging and installation information), see *SampleFilterScheme* in the ADC Reference Library. Note that the *SampleFilterScheme* project defines two different targets, one of which allows you to install the filter scheme on the boot partition. Be sure to read the comments in the project's files before you decide which target to build.

For more information on how to develop kernel extensions in general and I/O Kit drivers in particular, see *Kernel Extension Programming Topics* and *I/O Kit Device Driver Design Guidelines*.

Important: The sample code in this chapter is designed to work with Mac OS X version 10.1 and later. It will not work with earlier versions.

Edit Your Driver's Property List

Every driver has an `Info.plist` file that contains information about the driver and what it needs, including its personalities. A filter-scheme driver matches on content in a partition rather than on a device, so its personality contains information that identifies specific content. As described in "Filter-Scheme Driver Matching" (page 43), a filter-scheme driver uses the `Content Hint` property to match on the content-hint

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

string a disk utility program places in a partition. To make sure your driver loads for your content, you add the `Content Hint` property and associated content-hint value to its personality dictionary. You can also add other properties that identify media characteristics, such as ejectability and writability.

For step-by-step instructions that describe how to create a personality dictionary for a driver and add children to it, see the Hello I/O Kit tutorial in *Kernel Extension Programming Topics*.

The sample code uses the following five property keys:

- `CFBundleIdentifier`
- `IOClass`
- `IOProviderClass`
- `Content Hint`
- `IOMatchCategory`

Create five new children of the `MyFilterScheme` personality dictionary, one for each of the five properties you'll be adding. Table 7-1 (page 66) shows the properties, along with their classes and values. To test the sample code with your device, replace values such as `MySoftwareCompany_MyContentHint` with actual values for your content.

Table 7-1 Personality properties for MyFilterScheme

Property	Class	Value
`CFBundleIdentifier`	String	`com.MySoftwareCompany.driver.MyFilterScheme`
`IOClass`	String	`com_MySoftwareCompany_driver_MyFilterScheme`
`IOProviderClass`	String	`IOMedia`
`Content Hint`	String	`MySoftwareCompany_MyContent`
`IOMatchCategory`	String	`IOStorage`

A driver declares its dependencies on other loadable kernel extensions and in-kernel components in the `OSBundleLibraries` dictionary. Each dependency has a string value that declares the earliest version of the dependency the driver is compatible with.

The sample driver depends on one loadable extension from the Storage family and three kernel components. To add these dependencies to the `OSBundleLibraries` dictionary, you create a new child for each dependency. Table 7-2 (page 66) shows the dependencies you add for the sample driver.

Table 7-2 Dependencies for MyFilterScheme

Property	Class	Value
`com.apple.iokit.IOStorageFamily`	String	`1.1`
`com.apple.kernel.iokit`	String	`1.1`
`com.apple.kernel.libkern`	String	`1.1`

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Property	Class	Value
com.apple.kernel.mach	String	1.1

Finally, to allow this filter scheme to filter the boot volume, you must ensure that it is loaded at boot time so that it can be installed on top of the boot volume. To do this, you add the OSBundleRequired property to the top level of your Info.plist file and give it the string value Local-Root. If you do not need to filter the boot partition, do not add this property-value pair to your Info.plist file.

Creating Your Filter Scheme

This section describes some of the elements that must be included in your driver's source files. To demonstrate the process of creating a filter-scheme driver, the sample driver implements most of the needed methods by acting as a pass-through, in other words, calling through to its provider media. You should replace these trivial implementations with your own code that supports your filtering functionality.

Edit the Header File

The header file for the sample filter-scheme driver includes ten method declarations and two external header files. In the interests of brevity, the sample code includes only a condensed version of the standard comments accompanying each method declaration. You can find fully commented versions of these method declarations in IOMedia.h and IOStorage.h (both of which are in /System/Library/Frameworks/IOKit.framework/Headers/storage).

Edit the MyFilterScheme.h file to match the code in Listing 7-1 (page 67).

Listing 7-1 The MyFilterScheme header file

```
#include <IOKit/storage/IOMedia.h>
#include <IOKit/storage/IOStorage.h>

class com_MySoftwareCompany_driver_MyFilterScheme : public IOStorage {

    OSDeclareDefaultStructors(com_MySoftwareCompany_driver_MyFilterScheme)

protected:

    IOMedia*    _childMedia;

    // Free all of this object's outstanding resources.

    virtual void free(void);

    // The handleOpen method grants or denies permission to access this
    // object to an interested client.

    virtual bool handleOpen(IOService*   client,
                            IOOptionBits options,
                            void*        access);
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
    // The handleIsOpen method determines whether the specified client,
    // or any client if none is specified, presently has an open
    // on this object.

    virtual bool handleIsOpen(const IOService* client) const;

     // The handleClose method closes the client's access to this object.

    virtual void handleClose(IOService* client, IOOptionBits options);

    // Attach the passed-in media object to the device tree plane.
    // This is necessary if you want to stack this filter scheme on top
    // of the boot volume. You do not need to include this method if you
    // do not need to filter the boot volume.

    virtual bool attachMediaObjectToDeviceTree(IOMedia* media);

    // Detach the passed-in media object from the device tree plane.
    // This is necessary if you want to stack this filter scheme on top
    // of the boot volume. You do not need to include this method if you
    // do not need to filter the boot volume.

    virtual void detachMediaObjectFromDeviceTree(IOMedia* media);

public:

     // Initialize this object's minimal state.

    virtual bool init(OSDictionary* properties = 0);

     // Publish the new IOMedia object that represents the filtered content.

    virtual bool start(IOService* provider);

    // Clean up after the published media object before terminating.

    virtual void stop(IOService* provider);

    // Read data from the storage object at the specified byte offset into
    // the specified buffer, asynchronously.  When the read completes,
    // the caller will be notified via the specified completion action.
    // The buffer will be retained for the duration of the read.

    virtual void read(IOService*           client,
                      UInt64               byteStart,
                      IOMemoryDescriptor*  buffer,
                      IOStorageCompletion  completion);

    // Write data into the storage object at the specified byte offset from
    // the specified buffer, asynchronously.  When the write completes, the
    // caller will be notified via the specified completion action.
    // The buffer will be retained for the duration of the write.

    virtual void write(IOService*           client,
                       UInt64               byteStart,
                       IOMemoryDescriptor*  buffer,
                       IOStorageCompletion  completion);
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
    // Flush the cached data in the storage object, if any, synchronously.
    // The I/O Kit provides for data caches at the driver level, but
    // Apple discourages this because it is rarely needed. In the majority
    // of cases, a pass-through implementation is sufficient.

    virtual IOReturn synchronizeCache(IOService* client);

    // Obtain this object's provider.  This method returns IOMedia,
    // rather than the less-specific OSObject, as a convenience.

    virtual IOMedia* getProvider() const;
};
```

Edit the C++ File

The C++ file provides the code to implement the chosen methods. The sample driver's C++ file contains all the elements required for a subclassed filter-scheme driver even though it performs no filtering. To implement your filtering scheme, add code to the `read` and `write` methods.

The sample code in includes two methods you must implement if you want your filter scheme to filter the boot volume:

- `attachMediaObjectToDeviceTree`
- `detachMediaObjectFromDeviceTree`

Note: If you do not need to filter the boot volume, you can skip ahead to the code in Listing 7-2 (page 70), ignoring the code for these two methods.

The `attachMediaObjectToDeviceTree` method, is called in your `start` routine after the call to the standard `attach` method that attaches the new media to your filter scheme. This method detaches your filter scheme's parent object from the Open Firmware device tree and attaches the filter scheme's child object to the Open Firmware device tree in its place. This must be done before you publish the new media object in the I/O Registry using the `registerService` method. The second method, the `detachMediaObjectFromDeviceTree` method, performs the operation in reverse in your `stop` routine.

To understand why this rearranging of device tree nodes is necessary, it helps to know more about the Mac OS X boot process. When you turn on your computer, Open Firmware determines which volume to boot from. It then loads the secondary booter (named BootX) from that volume and jumps to it. BootX loads and runs the kernel, passing to it parameters it inherits from Open Firmware, including the device tree.

After the kernel comes up, it must mount the root volume. By this time, Open Firmware is no longer running, so the kernel determines the root volume by interpreting a parameter Open Firmware passed to it earlier. The parameter contains the root path property of the `/chosen` node in the Open Firmware device tree. The kernel searches the I/O Registry for a node whose Open Firmware path matches the root path. The kernel uses this node as the root device.

If there is a filter scheme installed on top of this node, the kernel is not aware of it and it continues to boot from the unfiltered node. Later on in the process, the system notices that the filter scheme is publishing a new child node that hasn't been mounted on, so it mounts the file system on that node. This results in the appearance of two copies of the boot volume on the Desktop, each with a separate data path, which is an undesirable outcome.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Edit the `MyFilterScheme.cpp` file to match the code in Listing 7-2 (page 70).

Listing 7-2 The MyFilterScheme C++ file

```
#include <IOKit/assert.h>   // For debugging purposes.
#include <IOKit/IOLib.h>
#include "MyFilterScheme.h"

// This definition allows you to use the more convenient "super" in
// place of "IOStorage", where appropriate.
#define super IOStorage

// This macro must appear before you define any of your class's methods.
// Note that you must use the literal name of the superclass here,
// not "super" as defined above.
OSDefineMetaClassAndStructors(com_MySoftwareCompany_driver_MyFilterScheme,
                                IOStorage)

// Define the methods to implement.
bool com_MySoftwareCompany_driver_MyFilterScheme::init(OSDictionary*
                                            properties = 0)
{
    //
    // Initialize this object's minimal state.
    //

    // Call superclass's init.

    if (super::init(properties) == false)  return false;

    // Initialize state.

    _childMedia = 0;

    return true;
}


void com_MySoftwareCompany_driver_MyFilterScheme::free(void)
{
    //
    // Free all of this object's outstanding resources.
    //

    if ( _childMedia )  _childMedia->release();

    // Call superclass's free.
    super::free();
}


IOMedia* com_MySoftwareCompany_driver_MyFilterScheme::getProvider(void)
    const
{

    return (IOMedia*) IOService::getProvider();
}
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
bool com_MySoftwareCompany_driver_MyFilterScheme::start(IOService* provider)
{
    //
    // Publish the new media object that represents the filtered content.
    //

    IOMedia* media = OSDynamicCast (IOMedia, provider);

    // State assumptions.

    assert(media);

    // Call superclass's start.

    if ( super::start(provider) == false )
        return false;

    // Attach and register the new media object.

    IOMedia* childMedia = new IOMedia;

    if ( childMedia )
    {
        if ( childMedia->init(
                /* base               */ 0,
                /* size               */ media->getSize(),
                /* preferredBlockSize */ media->getPreferredBlockSize(),
                /* isEjectable        */ media->isEjectable(),
                /* isWhole            */ false,
                /* isWritable         */ media->isWritable(),
                /* contentHint        */ "Apple_HFS" ) )
        {
            // Set a name for this partition.

            UInt32 partitionID = 1;

            char name[24];
            sprintf(name, "MySoftwareCompany_Filtered %ld", partitionID);
            childMedia->setName(name);

            // Set a location value (partition number) for this partition.

            char location[12];
            sprintf(location, "%ld", partitionID);
            childMedia->setLocation(location);

            // Attach the new media to this driver

            _childMedia = childMedia;

            childMedia->attach(this);

            // Move parent node to child node.
            (void) attachMediaObjectToDeviceTree(childMedia);

            // Publish the new media object.
            childMedia->registerService();
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
            return true;
        }
        else
        {
            childMedia->release();
            childMedia = 0;
        }
    }

    return false;
}

void com_MySoftwareCompany_driver_MyFilterScheme::stop(IOService* provider)
{
    // Clean up after the media object before terminating.

    // State assumptions.
    assert(_childMedia);

    // Detach the media object previously attached in start().
    if (_childMedia)
        detachMediaObjectFromDeviceTree(_childMedia);

    super::stop(provider);
}

bool com_MySoftwareCompany_driver_MyFilterScheme::handleOpen(IOService*
                                            client,
                                            IOOptionBits options,
                                            void* argument)
{
    return getProvider()->open(this, options, (IOStorageAccess) argument);
}


bool com_MySoftwareCompany_driver_MyFilterScheme::handleIsOpen(const
                                            IOService* client) const
{
    return getProvider()->isOpen(this);
}


void com_MySoftwareCompany_driver_MyFilterScheme::handleClose(IOService*
                                        client, IOOptionBits options)
{
    getProvider()->close(this, options);
}

bool com_MySoftwareCompany_driver_MyFilterScheme::attachMediaObjectToDeviceTree(
                                            IOMedia* media)
{
    //
    // Attach the given media object to the device tree plane.
    //

    IORegistryEntry* child;
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
    if ((child = getParentEntry(gIOServicePlane))) {

        IORegistryEntry* parent;

        if ((parent = child->getParentEntry(gIODTPlane))) {

            const char* location = child->getLocation(gIODTPlane);
            const char* name     = child->getName(gIODTPlane);

            if (media->attachToParent(parent, gIODTPlane)) {
                media->setLocation(location, gIODTPlane);
                media->setName(name, gIODTPlane);

                child->detachFromParent(parent, gIODTPlane);

                return true;
            }
        }
    }

    return false;
}

void com_MySoftwareCompany_driver_MyFilterScheme::detachMediaObjectFromDeviceTree
                                            (IOMedia* media)
{
    //
    // Detach the given media object from the device tree plane.
    //

    IORegistryEntry* child;

    if ((child = getParentEntry(gIOServicePlane))) {

        IORegistryEntry * parent;

        if ((parent = media->getParentEntry(gIODTPlane))) {

            const char* location = media->getLocation(gIODTPlane);
            const char* name     = media->getName(gIODTPlane);

            if (child->attachToParent(parent, gIODTPlane)) {
                child->setLocation(location, gIODTPlane);
                child->setName(name, gIODTPlane);
            }

            media->detachFromParent(parent, gIODTPlane);
        }
    }
}

void com_MySoftwareCompany_driver_MyFilterScheme::read(IOService* __attribute__
 ((unused)) client,
                                        UInt64 byteStart,
                                        IOMemoryDescriptor* buffer,
                                        IOStorageCompletion completion)
{
    // Add filtering code here.
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

```
        getProvider()->read(this, byteStart, buffer, completion);
}


void com_MySoftwareCompany_driver_MyFilterScheme::write(IOService* __attribute__
 ((unused)) client,
                                        UInt64 byteStart,
                                        IOMemoryDescriptor* buffer,
                                        IOStorageCompletion completion)
{
   // Add filtering code here.
    getProvider()->write(this, byteStart, buffer, completion);
}


IOReturn com_MySoftwareCompany_driver_MyFilterScheme::synchronizeCache(
                                                    IOService* client)
{
    return getProvider()->synchronizeCache(this);
}
```

Testing Your Filter Scheme

To test the sample filter-scheme driver, you must first create a disk image for it to match on. You do this using the command-line tools `hdiutil`, which creates and manipulates disk images and `newfs_hfs`, which builds a file system on the disk image. For full documentation on these commands, see the `man` pages.

To create the disk image, open a window in the Terminal application (located at `/Applications/Utilities/Terminal`) and type the following commands.

```
$ hdiutil create -megabytes 5 -partitionType MySoftwareCompany_MyContent
                            ~/MySoftwareCompany_MyContent_Example.dmg
```

Then, you attach the disk image without mounting it, because it doesn't yet contain a valid file system. In Mac OS X version 10.2 and later, use this command:

```
$ hdiutil attach -nomount ~/MySoftwareCompany_MyContent_Example.dmg
```

In version of Mac OS X prior to 10.2, use this command (the `-nomount` option was added in Mac OS X version 10.2):

```
$ hdiutil attach ~/MySoftwareCompany_MyContent_Example.dmg
```

The `hdiutil attach` command displays the special device name that is associated with each partition on the disk image:

```
/dev/disk1          Apple_partition_scheme
/dev/disk1s1        Apple_partition_map
/dev/disk1s2        MySoftwareCompany_MyContent
```

Then, use the `newfs_hfs` command to create the file system on your disk image. Because you should always use the raw, uncached disk, you add an `r` to the disk node representing the partition with your content.

```
$ newfs_hfs -v "My_Volume_Name" /dev/rdisk1s2
```


2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

The -v option allows you to specify a volume name.

After you've created the disk image and its file system, you can assume super user (or root) privileges and use the kextload command to load the sample filter-scheme driver. Alternately, if the root account owns the filter-scheme driver, you can copy it to /System/Library/Extensions, reboot, and the driver will load automatically.

To use the kextload command, type the following line in a Terminal window:

```
$ kextload -v MyFilterScheme.kext
```

The -v option makes kextload provide more verbose information.

After you've successfully loaded the driver, you can use Disk Copy to open the disk image (double-click on the disk image in the Finder).

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Document Revision History

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

Date	Notes
2007-04-03	Added guidance for creating CDB commands in a custom logical unit driver.
2006-05-23	Added a caveat that a filter scheme should not produce an IOCDMedia or IODVDMedia object.
2005-12-06	Made minor corrections.
2005-11-09	Added caution against sending READ and WRITE commands from a custom logical unit driver.
2005-09-08	Added chapter on endian issues for mass storage drivers and filter schemes. Changed title from "Writing Drivers for Mass Storage Devices."
2005-04-08	Fixed links; updated to refer to Xcode.
2005-02-03	Fixed quotes in newfs_hfs command.
2005-01-11	Fixed filter scheme sample code to allow filtering the boot volume. Added -nomount option to hdiutil command (for Mac OS X v. 10.2 and later).
2004-05-27	Fixed URL for USB Common Class Specification.
2002-01-15	First version.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

Index

A

Apple mass storage drivers
  logical unit drivers 29
  protocol services drivers 30
ATA mass storage drivers 11
ATAPI mass storage characteristics dictionary 40–41
ATAPI mass storage device. *See* IOATAPIProtocolTransport driver

B

block storage driver. *See* generic block storage driver
block storage layer 16

C

CDB (command descriptor block)
  and SCSITask object 20
  creating 53–58
  defined 15
command descriptor block. *See* CDB
command set builders 15, 19
compression. *See* filter schemes
content-hint string 28, 44, 65–66

D

developer resources
device compliance
  for logical unit drivers 29–30
  for protocol services drivers 30
device interfaces
  in SCSI Architecture Model family 21
  in Storage family 24
device services layer 15–17
  filter-scheme drivers in 16
  illustrated 16
  media filter layer in 16
device services nub 14, 17
disk image, for testing filter schemes 74–75
disk utility program 27
disk utility program 18
driver matching
  and driver personalities 31–32
  and information property list 31
  and matching dictionaries 31
  for filter schemes 43–44
  for IOATAPIProtocolTransport driver 39–41
  for IOFireWireSerialBusProtocolTransport driver 33–36
  for IOUSBMassStorageClass driver 36–39
  for logical unit drivers 41–42
  introduced 31–33
  three phases of 32
driver personalities
  filter scheme 44
  in filter scheme sample 65–66
  in IOATAPIProtocolTransport driver 39–41
  in IOFireWireSerialBusProtocolTransport driver 34
  in IOUSBMassStorageClass driver 36–39
  in logical unit driver sample 50–51
  in protocol services driver sample 60–61
  introduced 31–32
  logical unit driver 41–42
driver starting 33

E

encryption. *See* filter schemes

F

filter scheme sample 65–75
  driver personality 65–66
  information property list 65–66

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

testing 74–75
filter schemes
creating 27–28, 65–75
driver matching for 43–44
filter schemes
types of 16
FireWire SBP-2 mass storage device. *See* IOFireWireSerialBusProtocolTransport driver

G

generic block storage driver
in mass storage driver stack 16
in mass storage driver stack construction 26
in Storage family 22–23

I

`Info.plist`. *See* information property list
information property list
See also driver personalities
for filter scheme sample 65–66
for logical unit driver sample 50–51
for protocol services driver sample 60–61
introduced 31–32
IOApplePartitionScheme driver 23
IOATAPIProtocolTransport driver
and supported devices 30
driver matching for 39–41
subclassing 40–41
IOBlockStorageDevice class 22
IOBlockStorageDriver class 23
IOBlockStorageServices class 20, 22
IOCDBlockStorageDevice class 22
IOCDBlockStorageDriver class 23
IOCDMedia class 23
IOCDMediaBSDClient class 23
IOCDPartitionScheme driver 23
`IOClass` key 31
IOCommand class 20
IOCompactDiscServices class 20, 22
`ioctl` system call 23, 24
IODVDBlockStorageDevice class 22
IODVDBlockStorageDriver class 23
IODVDMedia class 23
IODVDMediaBSDClient class 23
IODVDServices class 20, 22
IOFDiskPartitionScheme driver 23
IOFireWireSerialBusProtocolTransport driver
and supported devices 30
driver matching for 33–35
IOMedia class
accessing from applications 24
defined 22
matching properties for 43–44
IOMediaBSDClient class 23
IONeXTPartitionScheme driver 23
IOPartitionScheme class 23
`IOProbeScore` key 32
`IOProviderClass` key 32
IOReducedBlockServices class 20, 22
IOSCSIBlockCommandsDevice class 20
IOSCSIMultimediaCommandsDevice class 20
IOSCSIPeripheralDeviceNub 19
IOSCSIPeripheralDeviceType00 driver
defined 30
driver matching for 42
IOSCSIPeripheralDeviceType05 driver 30
IOSCSIPeripheralDeviceType07 driver 30
IOSCSIPeripheralDeviceType0E driver 30
IOSCSIPrimaryCommandsDevice class 20
IOSCSIProtocolInterface class 19
IOSCSIProtocolServices class 19
IOSCSIReducedBlockCommandsDevice class 20
IOStorage class 22
IOUSBMassStorageClass driver
and supported devices 30
and vendor-specific class devices 37–38
driver matching for 36–39
subclassing 39
IOUserClient class 20

L

logical unit driver sample 49–53
driver personality 50–51
information property list 50–51
testing 53
logical unit drivers
and device compliance 29
Apple-provided 29
driver matching for 41–42
subclassing 26–27, 49–53
LUD. *See* logical unit drivers

M

mass storage driver stack
construction of 24
illustrated 13

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.

mass storage driver stack
layers of 13
matching dictionary 31–32
See also driver personalities
media filter layer 16
media filter schemes. *See* filter schemes

P

parallel SCSI 11
parallel SCSI 12
partition-scheme drivers 23
peripheral device nub 15
See also IOSCSIPeripheralDeviceNub
personality dictionary. *See* driver personalities
physical interconnect layer 13
physical interconnect transport protocol compliance 29
probe score 32–33
probing
in filter schemes 44
in IOATAPIProtocolTransport driver and subclasses 39–41
in IOFireWireSerialBusProtocolTransport driver 34–36
introduced 32–33
protocol services driver sample 59–64
driver personality 60–61
information property list 60–61
testing 64
protocol services drivers
and device compliance 29–30
Apple-provided 30
driver matching for 33
in SCSI protocol layer 15
IOATAPIProtocolTransport 30, 39–41
IOFireWireSerialBusProtocolTransport 30, 33–35
IOUSBMassStorageClass 30, 36–39
subclassing 59–64

R

RAID scheme 16
resources for developers 8

S

SCSI application layer 14
SCSI Architecture Model family 18–21
and SCSITask objects 20
device interfaces in 21
SCSI Architecture Model specifications 8, 11, 12, 29
SCSI command set 29–30
SCSI command set compliance 29
SCSI commands, creating and sending 53–58
SCSI protocol layer 15
SCSIBlockCommands class 19
SCSIMultimediaCommands class 19
SCSIPrimaryCommands class 19
SCSIReducedBlockCommands class 19
SCSITask object 15, 20
SCSITaskUserClient class 20
Storage family 21–24
and IOMedia objects 24, 27–28
class hierarchy 22
device interfaces in 24
filter-scheme drivers in 23
partition-scheme drivers in 23
subclassing
in filter schemes 18, 23, 27–28, 65, 67–74
logical unit drivers 17–18, 26, 42, 49–53
protocol services drivers 34–36, 39, 40–41

T

transport driver layer 14–15
illustrated 14
logical unit drivers in 14
protocol services drivers in 15
SCSI application layer in 14
SCSI protocol layer in 15

U

USB mass storage characteristics dictionary 37–38
USB mass storage class device. *See* IOUSBMassStorageClass driver

V

validation. *See* filter schemes

2007-04-03 | © 2002, 2007 Apple Inc. All Rights Reserved.