
SCSI Architecture Model Device Interface Guide

Drivers, Kernel, & Hardware: User-Space Device Access



2007-02-08



Apple Inc.
© 2003, 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, Carbon, Cocoa, FireWire, Logic, Mac, Mac OS, Macintosh, Pages, and Xcode are trademarks of Apple 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 Corporation or its subsidiaries in the United States and other countries.

PowerPC 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 SCSI Architecture Model Device Interface Guide** 7

Organization of This Document 7

See Also 7

Chapter 1 **Accessing SCSI Parallel Devices** 9

SCSI Family Support for SCSI Parallel Devices 10

 Performance and Threading With CDB Commands 10

 The SCSI Device Interface 11

Working With SCSI Family Device Interfaces 12

 Getting the I/O Kit Master Port 13

 Finding the Device 14

 Getting the Device Interface 15

 Opening the Device and Sending Commands 17

 Closing the Device 20

Chapter 2 **Accessing SCSI Architecture Model Devices** 21

SCSI Architecture Model Family Device Support 21

 Sending SCSI or ATA Commands to a Storage Device 22

 SCSI Architecture Model Devices on Mac OS X 23

 The SCSI`TaskDeviceInterface` 25

 The `MMCTaskDeviceInterface` 26

 The `SCSITaskInterface` 27

SCSI Device Access in an Intel-Based Macintosh 27

Using SCSI Architecture Model Family Device Interfaces 28

 Device Matching for Authoring-Capable Devices 29

 Device Matching for Nonauthoring-Capable Devices 30

 Accessing the Device 31

Accessing a SCSI Architecture Model Device 31

 Setting Up Your Project 31

 Creating a Main Function 32

 Testing the Device 36

Document Revision History 43

Figures and Listings

Chapter 1 **Accessing SCSI Parallel Devices** 9

Figure 1-1	An application communicating with a SCSI device through a device interface	11
Listing 1-1	Setting up device look-up for both families	13
Listing 1-2	Creating a matching dictionary for a SCSI Parallel device	14
Listing 1-3	Finding a device	15
Listing 1-4	Creating a SCSI device interface	16
Listing 1-5	Getting cached data about a SCSI Parallel device	17
Listing 1-6	Getting a CDBCommandInterface object	18
Listing 1-7	Using a CDBCommandInterface object to send commands	19

Chapter 2 **Accessing SCSI Architecture Model Devices** 21

Figure 2-1	The mass storage driver stack	24
Figure 2-2	An application-based logical unit driver	25
Figure 2-3	An authoring application	26
Listing 2-1	Ensuring correct endian format of data read from a disc	28
Listing 2-2	Creating a subdictionary for the SCSTaskDeviceCategory	29
Listing 2-3	Creating a device characteristics subdictionary	30
Listing 2-4	Creating a matching dictionary for a nonauthoring device	30
Listing 2-5	Header files	32
Listing 2-6	Global variables	32
Listing 2-7	Setting up a main function to access a CD-R/W device	33
Listing 2-8	Handling command-line interrupts	34
Listing 2-9	Accessing the set of matching devices	35
Listing 2-10	Releasing the device objects	35
Listing 2-11	Getting an MMCDeviceInterface and obtaining exclusive access	36
Listing 2-12	Sending an INQUIRY command to the device	37
Listing 2-13	Sending a TEST_UNIT_READY command to the device	40
Listing 2-14	Printing the sense data	41

Introduction to SCSI Architecture Model Device Interface Guide

Note: This document was previously titled *Working With SCSI Architecture Model Device Interfaces*.

The I/O Kit provides a device interface mechanism that allows applications to communicate with and control hardware from outside the kernel. This document describes how to access SCSI devices from applications using the SCSI family device interfaces and the SCSI Architecture Model family device interfaces.

Organization of This Document

This document contains the following chapters:

- ["Accessing SCSI Parallel Devices"](#) (page 9) describes how to use both the SCSI Architecture Model family APIs and the deprecated SCSI family APIs to look up a SCSI Parallel device. It then shows how to access the device using the deprecated SCSI family API. To access a device using the SCSI Architecture Model family API, see ["Accessing SCSI Architecture Model Devices"](#) (page 21).
- ["Accessing SCSI Architecture Model Devices"](#) (page 21) describes how to use the SCSI Architecture Model family device interfaces to access and control devices that conform to the SCSI Architecture Model specifications and declare a peripheral device type other than \$00, \$05, \$07, or \$0E. This chapter also contains a section on creating a universal binary version of your device access application.
- ["Document Revision History"](#) (page 43) 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.

- *Accessing Hardware From Applications* describes various ways to access devices from outside the kernel, including the device interface mechanism provided by the I/O Kit. For an overview of the I/O Kit terms and concepts used in this document, read the chapter *Device Access* and the *I/O Kit*.
- *I/O Kit Framework Reference* contains API reference for I/O Kit methods and functions and for specific device families.
- *Sample Code > Hardware & Drivers > SCSI* includes both application-level and in-kernel code samples. Of particular relevancy to this document is the *SCSIOldAndNew* sample project.
- Additional application-level code samples are included as part of the Mac OS X Developer Tools installation package in `/Developer/Examples/IOKit/scsi`.
- *Mac OS X Man Pages* provides access to existing reference documentation for BSD and POSIX functions and tools in a convenient HTML format.

INTRODUCTION

Introduction to SCSI Architecture Model Device Interface Guide

- The [ata-scsi-dev](#) mailing list provides a forum for discussing Mac OS X development related to devices based on ATA and SCSI technology.

If you're ready to create a universal binary version of your SCSI device-access application 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.

A detailed description of the SCSI Architecture Model specifications is beyond the scope of this document—for more information, see <http://t10.org>.

Accessing SCSI Parallel Devices

In versions of Mac OS X prior to v10.2, if you wanted to access a SCSI Parallel device that was not accessible with the SCSI Architecture Model family's device interfaces, you used the SCSI family's device interfaces. Beginning with Mac OS X v10.2, however, Apple introduced the SCSI Parallel family to support SCSI controllers. The new family is designed to allow the SCSI Architecture Model family to support SCSI devices attached to those controllers, which means you can use the SCSI Architecture Model family's device interfaces to access all SCSI devices that do not declare a peripheral device type of \$00, \$05, \$07, or \$0E.

As third-party developers release new SCSI controller drivers that use the SCSI Parallel family and as users install these new drivers, the SCSI Parallel family will replace the SCSI family. During the transition, however, the APIs of both families are available. This means that you can use the API of both families to look up your device and then use the API that successfully found your device to communicate with it. You might need to use both APIs in a single application if the following is true:

- Your application must remain compatible with Mac OS X v10.2.x or earlier.
- You need to access a SCSI Parallel device you could not previously access using the SCSI Architecture Model family's device interfaces.

If these statements describe your situation, you can use both APIs to find and communicate with your device. This way, your application will be able to find all the devices it's interested in regardless of the SCSI controller drivers the user has installed. More importantly, your application will still work when the SCSI family is replaced and all SCSI controllers are supported by the SCSI Parallel family. (Note, however, that the APIs of the deprecated SCSI family will not work in an Intel-based Macintosh.)

If, on the other hand, compatibility with Mac OS X v10.2.x or earlier is not required or your application already uses the device interfaces of the SCSI Architecture Model family to access your device, you should not use the SCSI family API in your application. Instead, see "[Accessing SCSI Architecture Model Devices](#)" (page 21) for information on how to use the SCSI Architecture Model family device interfaces.

This chapter describes how to use the deprecated SCSI family API to find and access a SCSI Parallel device that you cannot access using the SCSI Architecture Model family API. To illustrate this, it uses the *SCSIOldAndNew* sample project, which employs the API of both families to find and access a SCSI Parallel device. Because the SCSI Architecture Model family's device interfaces are thoroughly covered in "[Accessing SCSI Architecture Model Devices](#)" (page 21), this chapter focuses on the SCSI family's application-level support for SCSI devices. Although the *SCSIOldAndNew* sample project demonstrates how to use the APIs of both families to find and access a SCSI Parallel device, this chapter does not describe the project's use of the SCSI Architecture Model API. If you are unfamiliar with the SCSI Architecture Model family and you plan to use the APIs of both families in your application, be sure to read "[Accessing SCSI Architecture Model Devices](#)" (page 21) in addition to reading this chapter.

Important: The sample code in this chapter is written to work with Mac OS X v10.2 or later and may not work with earlier versions.

Although the sample code outlined in this chapter has been compiled and tested to some degree, Apple does not recommend that you directly incorporate this code into a commercial application. Its function is to illustrate the techniques you need to access a SCSI Parallel device; it does not attempt to exercise all features of the APIs or to demonstrate exhaustive error handling or an ideal user interface.

Important: The APIs of the deprecated SCSI family (including the `IO SCSI Device Interface` and `CDB Command Interface`) will not work in an Intel-based Macintosh. If you are ready to develop a universal binary version of your device-access application, you must use the device interface APIs provided by the SCSI Architecture Model family. See "[SCSI Device Access in an Intel-Based Macintosh](#)" (page 27) for more information.

SCSI Family Support for SCSI Parallel Devices

The SCSI (Small Computer System Interface) Parallel Interface is an industry standard parallel data bus that provides a consistent method of connecting computers and peripheral devices. SCSI Parallel devices use SCSI Parallel technology to communicate with a computer.

On computers that do not have new SCSI Parallel family–supported SCSI controller drivers installed, the SCSI family provides application-level access to SCSI Parallel devices with two device interfaces:

- The `IO SCSI Device Interface`, which provides functions that access the device
- The `IO CDB Command Interface`, which provides functions that create and execute CDB (command descriptor block) commands

The `IO SCSI Device Interface` is defined in `IO SCSI Lib.h` and the `IO CDB Command Interface` is defined in `IO CDB Lib.h`, both in the I/O Kit framework. This section provides a brief description of CDB performance and an outline of how to gain access to a SCSI Parallel device.

Performance and Threading With CDB Commands

A command descriptor block is defined as a structure up to 16 bytes in length that is used to communicate a command from an application client to a device server. To improve performance, Mac OS X queues CDB commands to help keep a SCSI Parallel device busy. When you use asynchronous commands, you can create more than one CDB command instance and execute subsequent commands as soon as you have queued the first command (by executing it). In fact, for best performance, you should try to use at least two commands whenever possible.

When a command completes, you can reuse the CDB command instance to execute another command. In reusing commands, keep the following in mind:

- No data in a CDB command is changed by executing the command, so that when you reuse the command, you must reset any values that need to be different for the next command.

- When you use the `setAndExecuteCommand` function to set up and execute a CDB command, it sets every CDB option, so you don't need to worry about clearing previous data.
- Once you have executed a command, it is not safe to change its value until its completion routine is called.

You should create and communicate with a CDB command only from code in a single thread. However, you can have multiple threads that each create, execute, and reuse multiple commands. CDB commands are executed in the order sent, but if you use multiple threads to execute multiple commands, your code is responsible for ensuring any required ordering of commands.

When using multiple commands, don't call `close` on the device until all the commands have completed. If you close a device with one or more outstanding asynchronous commands, some queued commands may be cancelled with a completion status of failed.

The SCSI Device Interface

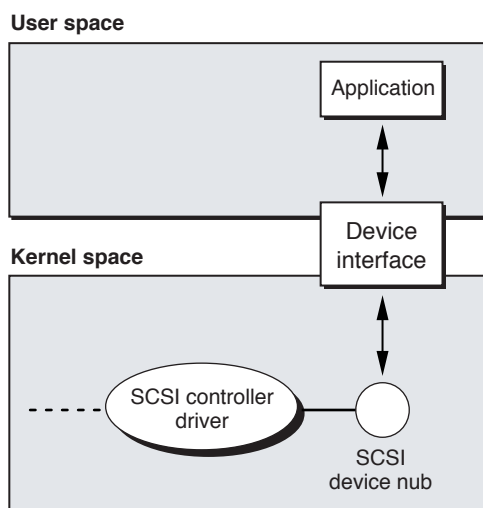
The I/O Kit's SCSI family provides a SCSI device interface that applications can use to access SCSI Parallel devices on Mac OS X. For the full definition of the SCSI device interface, see `IOSCSILib.h`.

You may need to access SCSI Parallel devices for a number of reasons, such as:

- Your utility application needs to list all the currently available SCSI Parallel devices, query them, and possibly perform operations on them (you can find devices and obtain cached device information from the I/O Registry without creating a device interface, but to query or control the device you do need a device interface).
- Your application needs to drive a SCSI scanner.

Figure 1-1 (page 11) shows an application that uses a device interface to act as a driver for a SCSI Parallel scanner. The application calls functions of the device interface, which communicates with a SCSI device nub (based on the `IO SCSI Device` class) in the kernel.

Figure 1-1 An application communicating with a SCSI device through a device interface



To the application, the device interface is just a plug-in interface that provides access to the device through functions such as `open`, `close`, and `reset`. To the device nub, the device interface looks like just another driver the nub can communicate with.

Working With SCSI Family Device Interfaces

Although each I/O Kit family that provides application-level access to devices may implement the device-interface mechanism in a slightly different way, the fundamental steps you take to use a device interface to access a device from an application remain the same:

1. Get the I/O Kit master port that allows applications to communicate with the I/O Kit.
2. Use family matching information and I/O Kit functions to find the device.
3. Get the appropriate device interface for the device.
4. Open a connection to the device and send it commands (in some cases, this step requires you to get an additional device interface).
5. Close the device and release any device interfaces you've acquired.

This section guides you through these steps, using code from the *SCSIOldAndNew* sample project to illustrate an example implementation. In the interest of brevity, this section does not reproduce the sample project in its entirety. Instead, this section provides partial listings from the project to illustrate the techniques you use to access a SCSI Parallel device using the API of the deprecated SCSI family. To run the sample project with your own devices, download the project from Sample Code > Hardware & Drivers > SCSI and build it on your computer.

Fundamental to the design of the *SCSIOldAndNew* sample project is the strict separation of the SCSI family functions from those of the SCSI Architecture Model family. Although you might choose to structure your code differently, the partitioning of the *SCSIOldAndNew* project is a useful feature to emulate. If your compatibility requirements change in the future, for example, it will be much easier to remove the code that uses the deprecated SCSI family API if it is kept separate from the rest of your application. The sample project divides its code into three modules:

- A main module, called `SCSIOldAndNew.c`, that establishes the user interface and calls functions from the remaining two modules to find and access the user-selected device.
- The `STUCMethod.c` module, which uses the SCSI Architecture Model family API to find and access the device. (The abbreviation STUC stands for SCSI Task User Client, the in-kernel counterpart of the SCSI Architecture Model family's SCSI Task Device Interface you use to access a SCSI device.)

For more information about the SCSI Architecture Model family's device interfaces, see "[Accessing SCSI Architecture Model Devices](#)" (page 21).

- The `OldMethod.c` module, which uses the deprecated SCSI family API to find and access the device.

The *SCSIOldAndNew* sample project is a Carbon application that defines an application-level event handler to process the user's device selection. Because this is a design issue that does not affect the core purpose of the application, this section glosses over the Carbon-related implementation details, focusing instead on the use of the I/O Kit and SCSI family APIs to find and access the device.

Getting the I/O Kit Master Port

The first step in accessing a device from an application is getting the I/O Kit master port. Because this step is required regardless of which family's API you use to find and access the device, the *SCSIOldAndNew* project performs it in its main module.

The main module of the *SCSIOldAndNew* project, called `SCSIOldAndNew.c`, gives the user a list of peripheral device types from which to choose. The application's event handler, a function called `DoAppCommandProcess`, passes the user's choice to the function `TestDevices.TestDevices` (shown in [Listing 1-1](#) (page 13)) acquires the I/O Kit master port and then attempts to find a device with the passed-in peripheral device type, first with the SCSI Architecture Model family API and then with the SCSI family API.

Listing 1-1 Setting up device look-up for both families

```
void TestDevices(int peripheralDeviceType)
{
    kern_return_t    kr;
    mach_port_t      masterPort = NULL;
    io_iterator_t    iterator = NULL;

    kr = IOMasterPort(MACH_PORT_NULL, &masterPort);
    if (kr != kIOReturnSuccess) {
        fprintf(stderr, "Couldn't retrieve the master I/O Kit port.
            (0x%08x)\n", kr);
        return;
    }

    // First try to find the device using the SCSI Architecture
    // Model family API:
    if (FindDevicesUsingSTUC(peripheralDeviceType, masterPort, &iterator)) {
        TestDevicesUsingSTUC(peripheralDeviceType, iterator);
    }
    else // Now try the SCSI family API:
        if (FindDevicesUsingOldMethod(peripheralDeviceType, masterPort,
            &iterator)) {
            TestDevicesUsingOldMethod(iterator);
        }
    else {
        fprintf(stderr, "No devices with peripheral device type %02Xh
            found.\n", peripheralDeviceType);
    }

    if (iterator) {
        IOObjectRelease(iterator);
    }

    if (masterPort) {
        mach_port_deallocate(mach_task_self(), masterPort);
    }
}
```

Although you might choose to develop your application differently, you should consider replicating the basic structure of this function somewhere in your code. If you require compatibility with Mac OS X v10.2.x or earlier or your device wasn't previously accessible using the SCSI Architecture Model family's API, you should

first attempt to find the device using the SCSI Architecture Model API and if that fails, then try to find the device using the SCSI family API. You should then use whichever API successfully found the device to access the device.

Finding the Device

To perform device matching (finding a device by locating its entry in the I/O Registry), you first create a matching dictionary. A matching dictionary is a dictionary of key-value pairs that describe the properties of a device or other service. Recall that when a device is discovered on a Mac OS X system, the I/O Kit instantiates a nub object that represents the device, attaches the nub to the I/O Registry, and registers it. The device family publishes properties in the nub that the I/O Kit uses to find a suitable driver for the device. Once you've created a matching dictionary, you can add keys and values that specify these properties to match on.

The *SCSIOldAndNew* project performs all its SCSI family-specific device matching and device access tasks in the module named `OldMethod.c`. To find a device of the user-selected peripheral device type, the main module calls the `FindDevicesUsingOldMethod` function (shown later in [Listing 1-3](#) (page 15)), passing it the peripheral device type, the I/O Kit master port, and a pointer to an iterator. The iterator, an object of type `io_iterator_t`, will hold a reference to the first device object in a list of matching device objects found in the I/O Registry, if any.

First, however, `FindDevicesUsingOldMethod` must create a matching dictionary that describes the device. To do this, it declares a variable of type `CFMutableDictionaryRef` and passes the address of this dictionary and the peripheral device type to the function `CreateMatchingDictionaryForOldMethod`, shown in [Listing 1-2](#) (page 14).

Listing 1-2 Creating a matching dictionary for a SCSI Parallel device

```
void CreateMatchingDictionaryForOldMethod(SInt32 peripheralDeviceType,
                                         CFMutableDictionaryRef *matchingDict)
{
    SInt32 deviceTypeNumber = peripheralDeviceType;
    CFNumberRef deviceTypeRef = NULL;

    // Set up a matching dictionary to search the I/O Registry by class name
    // for all subclasses of IO SCSI Device.
    *matchingDict = IOServiceMatching(kIO SCSI DeviceClassName);

    if (*matchingDict != NULL)
    {
        // Add key for device type to refine the matching dictionary.
        // First create a CFNumber to store in the dictionary.
        deviceTypeRef = CFNumberCreate(kCFAllocatorDefault, kCFNumberIntType,
                                     &deviceTypeNumber);
        CFDictionarySetValue(*matchingDict, CFSTR(kSCSIPropertyDeviceTypeID),
                             deviceTypeRef);
    }
}
```

The `IOProviderClass-myDeviceClassName` key-value pair is a common one in matching dictionaries. The function in [Listing 1-2](#) (page 14) passes the device-class name to the I/O Kit convenience function `IOServiceMatching`, which creates the matching dictionary and places in it this key-value pair. Because the resulting dictionary would match on a potentially large number of devices, however, the `CreateMatchingDictionaryForOldMethod` function also adds another key-value pair to narrow down the search to devices that identify themselves as the passed-in peripheral device type.

With the matching dictionary from `CreateMatchingDictionaryForOldMethod`, the `FindDevicesUsingOldMethod` function performs device look-up, using the I/O Kit function `IOServiceGetMatchingServices`. This I/O Kit function looks in the I/O Registry for devices whose properties match the key-value pairs in the matching dictionary. It returns an `io_iterator_t` object you can think of as a pointer to a list of matching devices. To access each device, you pass the iterator object to the I/O Kit function `IOIteratorNext`, which returns an `io_object_t` object representing a matching device and resets the iterator to “point to” the next matching device. A partial listing of `FindDevicesUsingOldMethod` is shown in [Listing 1-3](#) (page 15).

Listing 1-3 Finding a device

```
boolean_t FindDevicesUsingOldMethod(SInt32 peripheralDeviceType,
                                   mach_port_t masterPort, io_iterator_t *iterator)
{
    CFMutableDictionaryRef matchingDict = NULL;
    boolean_t result = false;

    CreateMatchingDictionaryForOldMethod(peripheralDeviceType,
                                         &matchingDict);
    // ...

    // Now search I/O Registry for matching devices.
    kr = IOServiceGetMatchingServices(masterPort, matchingDict,
                                      iterator);

    if (*iterator && kr == kIOReturnSuccess) {
        result = true;
    }

    // IOServiceGetMatchingServices consumes a reference to the matching
    // dictionary, so we don't need to release the dictionary reference.

    return result;
}
```

The result `FindDevicesUsingOldMethod` returns tells the main module if it found any devices of the specified peripheral device type that the SCSI family supports.

Getting the Device Interface

A device interface provides functions your application or other code running on Mac OS X can use to access a device. Device interfaces are plug-in interfaces that conform to the Core Foundation plug-in model (for more information on Core Foundation plug-ins, see [Reference Library > Core Foundation](#)).

An I/O Kit family that provides a device interface defines a type that represents the collection of interfaces it supports and a type for each individual interface. The family gives these types UUIDs (universally unique IDs) that identify them. Most families also define meaningful names for these identifiers you can use in place of the 128-bit UUID values. The SCSI family, for example, defines the constant `kIO SCSIUserClientTypeID` as a synonym for the UUID `B4291228-0F0F-11D4-9126-0050E4C6426F` that identifies the SCSI device interface.

Before you can get a family-specific device interface, however, you must first create an interface of type `IOCFPlugInInterface`. This interface sets up functions required for all interfaces based on the Core Foundation plug-in model. Chief among these is the `QueryInterface` function, which creates instances of family-specific device interfaces.

To get the SCSI family device interface, your application performs the following steps:

1. Get an intermediate interface of type `IOCFPlugInInterface`.

To obtain this interface, the application calls the `IOCreatePlugInInterfaceForService` function, passing the `io_object_t` representing the matching device (received from `IOIteratorNext`), the value `kIO SCSIUserClientTypeID` for the plug-in type parameter, and the value `kIOCFPlugInInterfaceID` for the interface type parameter. (`kIO SCSIUserClientTypeID` is defined in `IO SCSILib.h` and `kIOCFPlugInInterfaceID` and `IOCreatePlugInInterfaceForService` are defined in `IOCFPlugIn.h`.)

2. Get a SCSI device interface.

To do this, the application calls the `QueryInterface` function of the `IOCFPlugInInterface` object, passing the UUID of the desired device interface. To retrieve the UUID from the family-defined device interface name, use the following term:

```
CFUUIDGetUUIDBytes(kIO SCSIDeviceInterfaceID)
```

3. Release the intermediate `IOCFPlugInInterface` object.

To do this, the application calls the `IODestroyPlugInInterface` function (defined in `IOCFPlugIn.h`).

After completing these steps, you have a device interface of type `IO SCSIDeviceInterface` you can use to examine the device's cached information, open the device, and create the more specific CDB command interface.

The *SCSI Old And New* project performs these steps in the `CreateDeviceInterfaceUsingOldMethod` function in the `OldMethod.c` module. Listing 1-4 (page 16) shows part of this function, which is passed an `io_object_t` representing a matching device (received from a call to `IOIteratorNext`) and a pointer to an interface of type `IO SCSIDeviceInterface`.

Listing 1-4 Creating a SCSI device interface

```
void CreateDeviceInterfaceUsingOldMethod(io_object_t scsiDevice,
                                       IO SCSIDeviceInterface ***interface)
{
    IOCFPlugInInterface **plugInInterface = NULL;
    HRESULT      plugInResult = S_OK;
    kern_return_t kr = kIOReturnSuccess;
    SInt32      score = 0;

    // Create the base interface of type IOCFPlugInInterface.
    // This object will be used to create the SCSI device interface object.
    kr = IOCreatePlugInInterfaceForService(scsiDevice,
                                           kIO SCSIUserClientTypeID, kIOCFPlugInInterfaceID,
                                           &plugInInterface, &score);

    if (kr != kIOReturnSuccess) {
        fprintf(stderr, "Couldn't create a plug-in interface for the
                       io_service_t. (0x%08x)\n", kr);
    }
    else {
        // Query the base plug-in interface for an instance of the specific
        // SCSI device interface object.
        plugInResult = (*plugInInterface)->QueryInterface(plugInInterface,
```

```

        CFUUIDGetUUIDBytes(kIO SCSI Device Interface ID),
        (LPVOID *) interface);

    if (plugInResult != S_OK) {
        fprintf(stderr, "Couldn't create SCSI device interface.
            (%ld)\n", plugInResult);
    }
    // We're now finished with the instance of IOCFPlugInInterface.
    IO Destroy Plug In Interface(plugInInterface);
}
}

```

Opening the Device and Sending Commands

With the `SCSIDeviceInterface` object you've created, you can examine cached information about a device before you open it. You might choose to do this if, for example, you want to display this information and allow the user to verify that this is, in fact, the desired device.

The `SCSIDeviceInterface` defines a function called `getInquiryData` that retrieves the information about the device. In the `OldMethod.c` module of the *SCSI Old And New* project, the function `GetInquiryDataUsingOldMethod` (shown in [Listing 1-5](#) (page 17)) demonstrates how to call this device-interface function.

Listing 1-5 Getting cached data about a SCSI Parallel device

```

void GetInquiryDataUsingOldMethod(IO SCSI Device Interface **interface)
{
    UInt8      inquiryData[255];
    UInt32     inquiryDataSize = sizeof(inquiryData);
    kern_return_t kr = kIOReturnSuccess;

    bzero(inquiryData, sizeof(inquiryData)); // Zero data block.

    // Call a function of the SCSI device interface that returns cached
    // information about the device.
    kr = (*interface)->getInquiryData(interface, (SCSIInquiry *) inquiryData,
        sizeof(inquiryData), &inquiryDataSize);

    // If error, print message and hang (for debugging purposes).
    if (kr != kIOReturnSuccess) {
        fprintf(stderr, "Couldn't get inquiry data for device. (0x%08x)\n",
            kr);
    }
    else {
        PrintSCSIInquiryDataUsingOldMethod((SCSIInquiry *) inquiryData,
            inquiryDataSize);
    }
}

```

The function `PrintSCSIInquiryDataUsingOldMethod` (not shown here) simply formats and displays device information, including:

- Peripheral device type
- Vendor ID

- Product ID and revision level
- Response data format
- Removability of media

Although you can get this information without opening the device, if you want to send commands to a SCSI Parallel device, you must open the device and create an additional interface, called a `CDBCommandInterface`.

The `SCSIDeviceInterface` defines the `open` function, which opens the device and, if it succeeds, causes all other calls to `open` to fail with the `kIOReturnExclusiveAccess` error. After you've opened the device, you use the `SCSIDeviceInterface`'s `QueryInterface` function (common to all Core Foundation plug-in interfaces) to create a `CDBCommandInterface`. You can use the `CDBCommandInterface` to send to the SCSI Parallel device commands such as:

- INQUIRY
- TEST UNIT READY
- READ BUFFER
- WRITE BUFFER
- REQUEST SENSE

These and other commands are defined in `SCSIPublic.h` in the I/O Kit framework.

The function `TestADeviceUsingOldMethod` (not shown here) demonstrates how to open a SCSI Parallel device using the `open` function of the passed-in `SCSIDeviceInterface`:

```
(*interface)->open(interface);
```

It then passes the `SCSIDeviceInterface` object to the function `CreateCommandInterfaceUsingOldMethod` to create the `CDBCommandInterface` object. `CreateCommandInterfaceUsingOldMethod` is shown in [Listing 1-6](#) (page 18).

Listing 1-6 Getting a `CDBCommandInterface` object

```
IOCDBCommandInterface **CreateCommandInterfaceUsingOldMethod
    (IOSCSIDeviceInterface **interface)
{
    HRESULT          plugInResult = S_OK;
    IOCDBCommandInterface **cdbCmdInterface = NULL;

    fprintf(stderr, "Opened device\n");

    // Use the constant kIOCDBCommandInterfaceID, defined in
    // IOCDLib.h, to identify the CDBCommandInterface.
    plugInResult = (*interface)->QueryInterface(interface,
        CFUUIDGetUUIDBytes(kIOCDBCommandInterfaceID),
        (LPVOID *) &cdbCmdInterface);

    // If error, print message and hang (for debugging purposes).
    if (plugInResult != S_OK) {
        fprintf(stderr, "Couldn't create a CDB command. (%ld)\n",
            plugInResult);
    }
}
```

```

    return cdbCmdInterface;
}

```

With a `CDBCommandInterface`, you can send SCSI commands to your device. The `ExecuteInquiryUsingOldMethod` function in the `OldMethod.c` module uses the INQUIRY command to illustrate how to set up a CDB command structure (defined in `CDBCommand.h`) and send it to the device. To do this, the `ExecuteInquiryUsingOldMethod` function performs the following steps:

1. It allocates and initializes stack variables (including `inquiryData`, `range`, and `cdb`) to specify an INQUIRY command and to store the results.
2. It sets up a variable of type `CDBInfo` to specify the command, then calls the `setAndExecuteCommand` function of the CDB command interface to set command values and execute the command.

On return from the `setAndExecuteCommand` function, for an asynchronous command, the `seqNumber` variable contains a unique sequence number. For a synchronous command, the sequence number is always 0.

The `setAndExecuteCommand` function (defined in `IOCDBLib.h`) is a utility function you can use instead of making multiple calls to set values and then calling the `execute` function.

3. It calls the `getResults` function of the CDB command interface to obtain the results of the INQUIRY command.
4. It calls the `MyPrintSCSIInquiryData` utility function (not shown here) to print the results.
5. Because it uses only stack variables, it has nothing to release.

[Listing 1-7](#) (page 19) shows the `ExecuteInquiryUsingOldMethod` function, minus its error-checking code.

Listing 1-7 Using a `CDBCommandInterface` object to send commands

```

void ExecuteInquiryUsingOldMethod(IOCDBCommandInterface
                                **cdbCommandInterface)
{
    UInt8          inquiryData[36 /* 255 */];
    IOVirtualRange range[1];
    CDBInfo        cdb;
    CDBResults     results;
    UInt32         seqNumber;
    kern_return_t  kr = kIOReturnSuccess;

    bzero(inquiryData, sizeof(inquiryData)); // Zero data block.

    range[0].address = (IOVirtualAddress) inquiryData;
    range[0].length  = sizeof(inquiryData);

    bzero(&cdb, sizeof(cdb));
    cdb.cdbLength = 6;
    cdb.cdb[0] = kSCSICmdInquiry;
    cdb.cdb[4] = sizeof(inquiryData);

    kr = (*cdbCommandInterface)->setAndExecuteCommand(
                                           cdbCommandInterface,
                                           &cdb,
                                           sizeof(inquiryData),

```

```

        range,
        sizeof(range) / sizeof(range[0]),
        0, /* isWrite */
        0, /* timeoutMS */
        0, /* target */
        0, /* callback */
        0, /* refcon */
        &seqNumber);

    // Check to be sure the INQUIRY command executed correctly here.

    kr = (*cdbCommandInterface)->getResults(cdbCommandInterface, &results);

    // Check to be sure the getResults command executed correctly here.

    PrintSCSIInquiryDataUsingOldMethod((SCSIInquiry *) inquiryData,
        results.bytesTransferred);
}

```

Closing the Device

When you've finished sending commands to the SCSI Parallel device, you must close it and release the device interfaces you acquired. Functions in the `OldMethod.c` module perform these tasks in reverse order, starting with the most recently acquired interface. The `TestDeviceUsingOldMethod` function releases the `CDBCommandInterface` object:

```

IOCDBCommandInterface **cdbCommandInterface;
(*cdbCommandInterface)->Release(cdbCommandInterface);

```

Next, it closes the device, using the `SCSIDeviceInterface` function `close`:

```

IOSCSIDeviceInterface **interface;
(*interface)->close(interface);

```

Finally, the `TestDevicesUsingOldMethod` function releases the `SCSIDeviceInterface` object:

```

IOSCSIDeviceInterface **interface;
(*interface)->Release(interface);

```

Accessing SCSI Architecture Model Devices

For devices supported by SCSI Architecture Model family drivers, the SCSI Architecture Model family provides two types of device interface, one for exclusive access to the device and one for nonexclusive access to media-mastering devices only. This chapter focuses on how to use these device interfaces to communicate with devices supported by SCSI Architecture Model family drivers and includes sample code that illustrates how to gain access to a CD-R/W drive.

If you need to access a SCSI Parallel device *and* your application must run in versions of Mac OS X prior to v10.2, see "[Accessing SCSI Parallel Devices](#)" (page 9) for information on how to use the API of both the SCSI Architecture Model family and the deprecated SCSI family to find your device. (Note that the APIs of the deprecated SCSI family will not work in an Intel-based Macintosh.)

Important: The sample code in this document is designed to work with Mac OS X v10.1 and later and will not work with earlier versions.

Although the sample code in this document has been compiled and tested, it is not intended to meet the needs of a commercial application. For example, error handling is minimal and simply facilitates debugging of this code—you should develop your own techniques for detecting and handling errors. In addition, you must employ your own method of disk arbitration. Therefore Apple does not recommend that you directly incorporate the entire sample program into a commercial application.

SCSI Architecture Model Family Device Support

The SCSI Architecture Model family supports peripheral devices that comply with both the SCSI Primary Commands specification, or SPC, and one of the following bus transport protocols:

- The ATA/ATAPI-5 specification (<http://t13.org>)
- The FireWire SBP-2 specification (<http://t10.org>)
- The USB Mass Storage Class specification (<http://www.usb.org>)
- The SCSI Parallel specification (SPI-4)

The SCSI Architecture Model family provides in-kernel logical unit drivers that translate generic I/O requests into device-specific commands for devices on these buses that declare one of the following four peripheral device types:

- \$00 for block storage devices that comply with the SCSI block commands specification
- \$05 for multimedia devices that comply with the SCSI multimedia commands specification
- \$07 for magneto-optical devices that comply with the SCSI block commands specification
- \$0E for reduced block command devices that comply with the SCSI reduced block commands specification

For devices that declare other peripheral device types, such as scanners, tape drives, and medium changers, the SCSI Architecture Model family provides a device interface, called the `SCSITaskDeviceInterface`, that allows an application to be the logical unit driver for that device.

The SCSI Architecture Model family also provides access to authoring devices that are supported by in-kernel logical unit drivers. The device interface for multimedia commands, called the `MMCDDeviceInterface`, allows an application to gain nonexclusive access to an authoring device. This allows the application to get information about the device prior to obtaining exclusive access to it with the `SCSITaskDeviceInterface`.

The SCSI Architecture Model family provides a third interface, called the `SCSITaskInterface`, that allows an application acting as a logical unit driver to manipulate the in-kernel `SCSITask` object. The `SCSITask` object contains the command being sent to the device, together with information such as task status and callback function pointers.

Sending SCSI or ATA Commands to a Storage Device

By design, Mac OS X does not allow applications to send SCSI or ATA commands to storage devices unless the application developer also provides an in-kernel device driver that supports the commands. The SCSI Architecture Model family allows only one logical unit driver to control a device at a time and provides in-kernel logical unit drivers for storage devices (as listed in "[SCSI Architecture Model Family Device Support](#)" (page 21)). Similarly, the ATA family does not allow applications to send ATA commands directly to ATA or SATA (Serial ATA) devices. This design preserves the integrity of the operating system and enhances the security of the user's data in two ways:

- An arbitrary application cannot directly change the state of a device without the explicit cooperation of a developer's custom in-kernel logical unit driver.
- An application cannot bypass the file-system permissions set by the user by sending I/O commands directly to a storage device.

Both the SCSI Architecture Model family and the ATA family provide device interfaces that allow applications various types of access to devices. The remaining sections in this chapter describe the device interfaces the SCSI Architecture Model family provides to allow an application to send commands to specific device types. The ATA family provides a device interface that allows applications to get SMART (self-monitoring, analysis, and reporting technology) data from ATA and SATA devices that implement the SMART feature set. For more information on this device interface, see the API reference documentation for `ATASMARTLib.h` in *I/O Kit Framework Reference*.

If your application needs information available from a `SCSI INQUIRY` or `ATA IDENTIFY DEVICE` command, however, you do not need to send commands to the device or use a device interface. Much of the information provided by the `INQUIRY` and `IDENTIFY DEVICE` commands is available in the I/O Registry, which you can view with the I/O Registry Explorer application (in `/Developer/Applications`) or the command-line tool `ioreg`.

If your application needs to perform block-level I/O operations on a storage device, you can access the BSD raw disk device node in `/dev`. For more information on how to do this, see *Device File Access Guide for Storage Devices*.

If you decide that your application must send SCSI or ATA commands to devices not supported by the device interfaces provided by the SCSI Architecture Model and ATA families, you will need to write a custom in-kernel logical unit driver. If you do create your own in-kernel logical unit driver, be sure you don't send `READ` or `WRITE` commands from the driver. If you send these commands, you create a security hole malicious code

can take advantage of by using your driver to read or destroy data on a device that a user has protected by setting access permissions. For more information on how to write a custom in-kernel logical unit driver, see *Mass Storage Device Driver Programming Guide* and the code sample *VendorSpecificType00*.

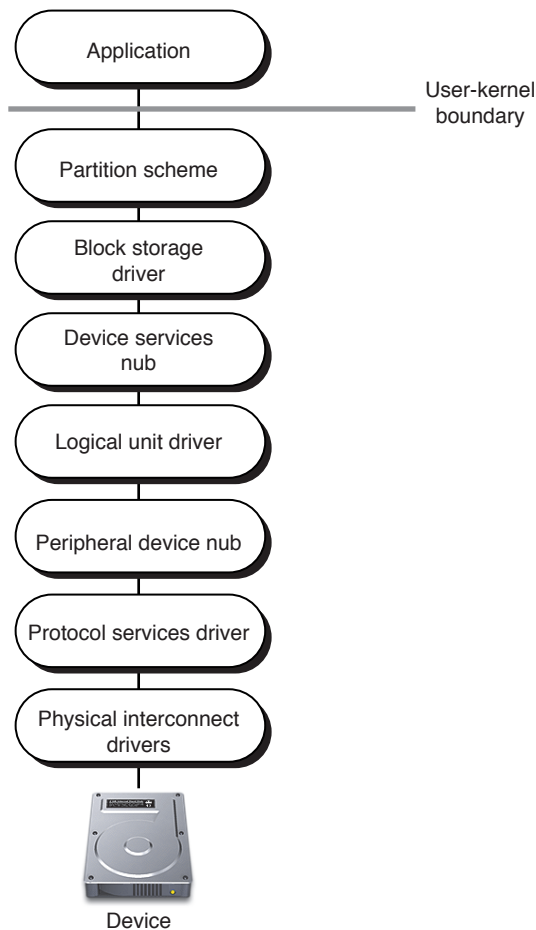
SCSI Architecture Model Devices on Mac OS X

When a device is discovered on Mac OS X, the I/O Kit finds and loads the drivers required to support it. In the case of a SCSI Architecture Model peripheral device that declares a peripheral device type of \$00, \$05, \$07, or \$0E, the I/O Kit loads several layers of drivers. Together, these layers are known as the mass storage driver stack (for more in-depth information about the mass storage driver stack, see *Introduction to Mass Storage Device Driver Programming Guide*).

The mass storage driver stack is divided into three main layers.

- The physical interconnect layer includes physical interconnect drivers that support the physical connection of the device to the bus.
- The transport driver layer includes the logical unit driver and protocol services driver that translate generic I/O requests into device-specific commands suitable for transport across a particular bus.
- The device services layer includes the block storage driver which supports generic system requests and optional filter drivers that can implement encryption or validation schemes.

[Figure 2-1](#) (page 24) shows the mass storage driver stack supporting a mass storage device that declares a peripheral device type of \$00, \$05, \$07, or \$0E.

Figure 2-1 The mass storage driver stack

The SCSI Architecture Model family supports the transport driver layer. It provides the logical unit driver that translates generic system requests into commands specific to the SCSI command set the device is compliant with and the protocol services driver that packages the commands from the logical unit driver into the format the bus expects.

The SCSI Architecture Model family enforces an exclusive-access policy for logical unit drivers. This means that there can be only one logical unit driver for each logical unit of a device. This applies to both in-kernel and application-based logical unit drivers.

When a device declaring a peripheral device type of \$00, \$05, \$07, or \$0E is discovered on one of the supported buses, the I/O Kit first finds and loads the appropriate physical interconnect drivers and protocol services driver. Then, an I/O Kit nub object, called the peripheral device nub, queries the device and publishes its peripheral device type in the I/O Registry. The I/O Kit then finds and loads the logical unit driver that matches the published peripheral device type.

Another I/O Kit nub object, called the device services linkage object, then publishes the logical unit driver's type in the I/O Registry. If the device is authoring-capable, the SCSI Architecture Model family publishes the `SCSITaskDeviceCategory` key in this nub object. This key, with the value `SCSITaskAuthoringDevice`, indicates that the `MMCDeviceInterface` is available for the device. The SCSI Architecture Model family also adds a globally unique identification value (or GUID) to the nub that an application can use to identify the device.

Next, the I/O Kit finds and loads the block storage driver, and, if the device is a CD-ROM or DVD-ROM with media present, it loads a CD or DVD partition scheme. The device is then ready to process I/O requests from the host system.

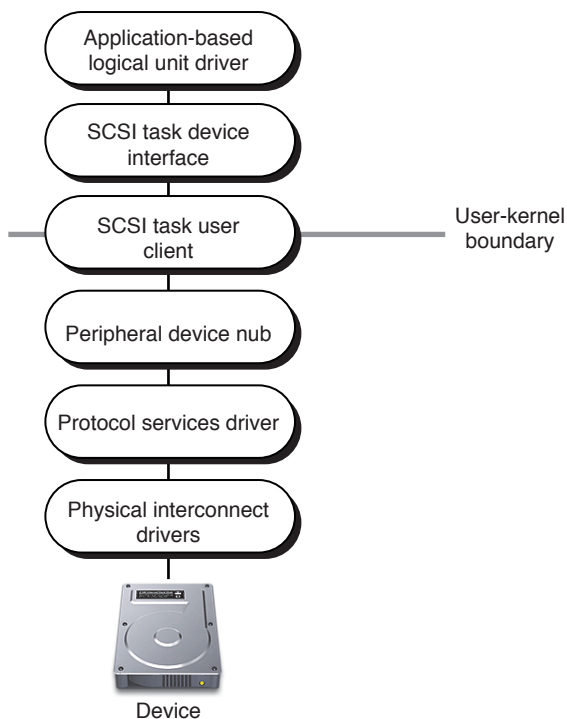
The SCSI Task Device Interface

When a device that declares a peripheral device type other than \$00, \$05, \$07, or \$0E is discovered on a supported bus, the I/O Kit begins the matching and loading process described in "[SCSI Architecture Model Devices on Mac OS X](#)" (page 23).

As before, the peripheral device nub publishes the peripheral device type reported by the device. Because the peripheral device type is *not* \$00, \$05, \$07, or \$0E, the SCSI Architecture Model family adds the GUID and the `SCSITaskDeviceCategory` key with the value `SCSITaskUserClientDevice` to the peripheral device nub. This key-value pair indicates that an application can use the SCSI Task Device Interface to drive the device.

Because there are no in-kernel logical unit drivers that match on a peripheral device type other than \$00, \$05, \$07, or \$0E, the building of the driver stack halts after the peripheral device nub is published. [Figure 2-2](#) (page 25) shows the mass storage driver stack when an application-based driver is the logical unit driver for a device that declares a peripheral device type of \$01.

Figure 2-2 An application-based logical unit driver



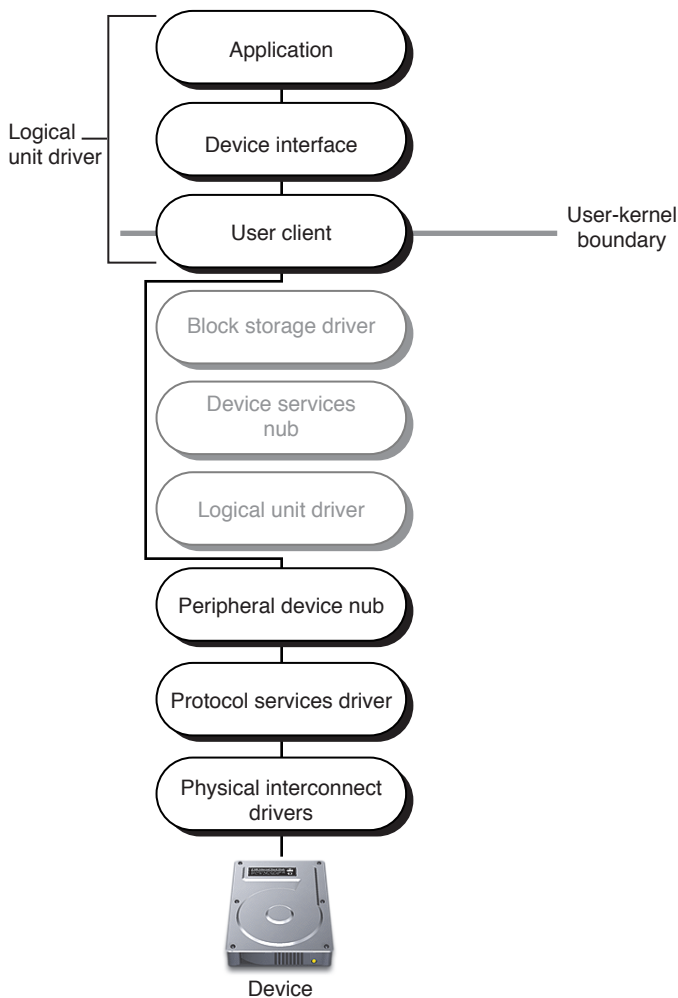
When an application-based driver for a device with no in-kernel logical unit driver starts, it searches the I/O Registry for the device or device type it can drive. When it finds the device, it acquires a SCSI Task Device Interface and the application becomes the logical unit driver for the device.

Because the application has exclusive access to the device, the device has no other clients and the device services layer of the mass storage driver stack is unnecessary.

The MMCDeviceInterface

When an authoring or media-mastering device, such as a CD-R/W or DVD-R/W, is discovered on a supported bus, the I/O Kit performs the matching and loading of drivers required to build the entire mass storage driver stack (as shown in [Figure 2-1](#) (page 24)). Because the SCSI Architecture Model family enforces an exclusive access policy, however, an authoring application must somehow replace the in-kernel logical unit driver to gain exclusive access to the device. [Figure 2-3](#) (page 26) shows the mass storage driver stack when an authoring application is the logical unit driver for a CD-R/W device.

Figure 2-3 An authoring application



The authoring application replaces the logical unit driver, with the help of the SCSI Architecture Model family and the I/O Kit, in two phases:

1. The authoring application obtains an `MMCDeviceInterface` that provides nonexclusive access to the device while the in-kernel logical unit driver continues to drive it. The `MMCDeviceInterface` allows the application to safely query the device and determine if it can, in fact, support it.

The application can also use the `MMCDeviceInterface` to determine if media is present in the device and, if so, to get information about the media such as its type and size.

2. Based on information obtained from its query, the application reserves the media, obtains a `SCSITaskDeviceInterface`, and requests exclusive access to the device.

When the request for exclusive access is granted, the in-kernel logical unit driver yields control to the authoring application. The block storage driver in the device services layer also relinquishes control and the Storage family tears down any partition schemes that were present. While the authoring application has exclusive access to the device, it can perform authoring functions.

When the authoring application terminates, the in-kernel logical unit driver and the block storage driver regain control of the device and the Storage family rebuilds the partition scheme, if needed.

The SCSITaskInterface

Logical unit drivers, whether in-kernel or application-based, use `SCSITask` objects to communicate with devices. The `SCSITask` object encapsulates all the information required during the life span of a single I/O transaction. This information includes the command descriptor block (or CDB) appropriate to the SCSI command set specification the device complies with, task retry status, and callback function pointers.

An application-based logical unit driver must have exclusive access to a device before it can create and use `SCSITask` objects. Because an application-based driver cannot manipulate the in-kernel `SCSITask` object directly, it must obtain a `SCSITaskInterface`. The `SCSITaskInterface` provides access to the in-kernel `SCSITask` object—each `SCSITaskInterface` object corresponds to exactly one `SCSITask` object.

SCSI Device Access in an Intel-Based Macintosh

This section provides an overview of some of the issues related to developing a universal binary version of an application that accesses a SCSI Architecture Model (or SCSI Parallel) device. Before you read this section, be sure to 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. The guidelines in that document apply to all types of applications, including those that access hardware.

Before you build your application as a universal binary, make sure that:

- You port your project to GCC 4 (Xcode uses GCC 4 to target Intel-based Macintosh computers)
- You install the Mac OS X v10.4 universal SDK
- You develop your project in Xcode 2.1 or later

Devices that comply with the SCSI architecture and command-set standards return data in the big-endian format, regardless of the native endian format of the computer your application is running in. Because a PowerPC-based Macintosh is also big-endian, you should examine your PowerPC application for places where you might assume that multibyte data never needs to be swapped.

You should also search for hard-coded byte swaps in your application (such as code that always swaps a multibyte value from one endian format to another). If your code contains such swaps (or it assumes byte swaps will never be necessary), it will not run correctly in an Intel-based Macintosh. Replace all hard-coded swaps with the appropriate conditional byte-swapping macros defined in the Kernel framework in `libkern/OSByteOrder.h`. These macros are available for use in any project, including high-level applications.

For example, the Authoring Unit Test sample project (in `/Developer/Examples/IOKit/scsi/SCSITaskLib/Authoring/Cocoa`) uses the `OSReadBigInt16` function to ensure the data returned from the disc is in host-endian byte order, as shown in Listing 2-1:

Listing 2-1 Ensuring correct endian format of data read from a disc

```
// Here, "interface" is an instance of an MMCDeviceInterface.
- (void) testReadDiscInfo {
    UInt8          discInfoBuffer[4096];
    IOReturn       err;
    SCSITaskStatus taskStatus;
    SCSI_Sense_Data senseData;
    UInt16         discInfoSize;
    ...
    // Issue a READ_DISC_INFORMATION command to the device. Ask for
    // 4 bytes to get the length field.
    err = (*interface)->ReadDiscInformation (interface, discInfoBuffer, 4,
&taskStatus, &senseData);
    // ...If the command completed successfully:
    // The first word contains the size. Add 2 bytes for the first word since
    it isn't counted
    discInfoSize = OSReadBigInt16 ( discInfoBuffer, 0 ) + sizeof ( UInt16 );
    ...
}
```

Fortunately, the SCSI command model specification defines the CDB (command descriptor block) as a byte array. This means that these bytes are stored in the defined order regardless of the native endian format of the computer your application is running in. As you create CDB commands using a `SCSITaskInterface` object, you do not have to worry about the endian format of the values you use.

You may need to perform byte swapping on values your application reads from a SCSI Parallel or SCSI Architecture Model device. Whether you have to perform byte swapping on the values you send and receive in parameters to device interface functions depends on the type of the value:

- A buffer containing multibyte fields that gets passed in a parameter must be in the big-endian format. For example, if you use the `MMCDeviceInterface` function `SetWriteParametersModePage` to issue a `MODE_SELECT` command, the buffer pointed to by the `buffer` parameter must be in the big-endian format.
- Other parameter values do not need to be byte-swapped. You should pass these parameter values in the computer's host format and the values you receive in the device interface functions will be in the computer's host format.

Using SCSI Architecture Model Family Device Interfaces

The device interfaces the SCSI Architecture Model family provides are plug-ins that specify functions your application can call to communicate with a device. These interfaces are defined in `/System/Library/Frameworks/IOKit.framework/Headers/scsi/SCSITaskLib.h`

Before you can use these interfaces, however, you must find the device you're interested in. Device matching is the process of using I/O Kit functions to search the I/O Registry for particular devices or device types. This process varies depending on whether you are developing an authoring application or an application-based

driver for a device that has no in-kernel logical unit driver. The next two sections describe how to perform device matching for both cases. Then, "[Accessing the Device](#)" (page 31) outlines the remaining steps in the process of getting and using SCSI Architecture Model family device interfaces.

Device Matching for Authoring-Capable Devices

If you're developing an authoring application, you first create a matching dictionary that matches on authoring-capable SCSI Architecture Model devices. To create the dictionary, you use Core Foundation functions as in this example:

```
CFMutableDictionaryRef    matchingDictionary;

matchingDictionary = CFDictionaryCreateMutable ( kCFAllocatorDefault, 0, NULL,
    NULL );
```

Now you need to place a key-value pair in the matching dictionary. The key is the `IOPropertyMatch` key and the value is a subdictionary you create. The subdictionary contains the `SCSITaskDeviceCategory` key with the value `SCSITaskAuthoringDevice`. Listing 2-2 shows how to create the subdictionary.

Listing 2-2 Creating a subdictionary for the `SCSITaskDeviceCategory`

```
CFMutableDictionaryRef    subDictionary;

subDictionary = CFDictionaryCreateMutable ( kCFAllocatorDefault, 0, NULL, NULL
    );
CFDictionarySetValue ( subDictionary, CFSTR ( kIOPropertySCSITaskDeviceCategory
    ), CFSTR ( kIOPropertySCSITaskAuthoringDevice ) );
```

If you do not need to perform a more specific search, you add the `IOPropertyMatch` key and this subdictionary to the matching dictionary you created earlier, as in this example:

```
CFDictionarySetValue ( matchingDictionary, CFSTR ( kIOPropertyMatchKey ),
    subDictionary );
```

You can refine this search by adding other properties to match on. For example, you can define a match on your device's GUID. You add a key-value pair for the GUID to the subdictionary you created above, as in this example:

```
CFDictionarySetValue ( subDictionary, CFSTR (
    kIOPropertySCSITaskUserClientInstanceGUID ), myGUID );
```

You can also create other subdictionaries that contain the physical interconnect protocol or device characteristics that describe your device. You then add these subdictionaries to the `SCSITaskDeviceCategory` subdictionary you created.

The Protocol Characteristics subdictionary can contain either or both of the following keys:

- `Physical Interconnect`
- `Physical Interconnect Location`

The Device Characteristics subdictionary can contain any of the following keys:

- `Vendor Name`

- Product Name
- Product Revision Level

If, for example, you need to match on a particular vendor and product name, you can create a subdictionary that contains those key-value pairs and add it to the `SCSITaskDeviceCategory` subdictionary you created earlier. [Listing 2-3](#) (page 30) shows how to do this.

Listing 2-3 Creating a device characteristics subdictionary

```
#define kMyVendorNameString    "MyVendorName"
#define kMyProductNameString   "MyProductName"

CFMutableDictionaryRef        deviceCharacteristicsDict;

deviceCharacteristicsDict = CFDictionaryCreateMutable ( kCFAllocatorDefault, 0,
    NULL, NULL );
CFDictionarySetValue ( deviceCharacteristicsDict, CFSTR ( kIOPropertyVendorNameKey
    ), CFSTR ( kMyVendorNameString ) );
CFDictionarySetValue ( deviceCharacteristicsDict, CFSTR (
    kIOPropertyProductNameKey ), CFSTR ( kMyProductNameString ) );

CFDictionarySetValue ( subDictionary, CFSTR ( kIOPropertyDeviceCharacteristicsKey
    ), deviceCharacteristicsDict );
```

Device Matching for Nonauthoring-Capable Devices

For an application that drives a device with no in-kernel logical unit driver, you create a matching dictionary that contains the `IOPropertyMatch` key. The value of this key is a subdictionary you create that contains the `SCSITaskDeviceCategory` key and the value `SCSITaskUserClientDevice`. [Listing 2-4](#) (page 30) shows how to do this.

Listing 2-4 Creating a matching dictionary for a nonauthoring device

```
CFMutableDictionaryRef        matchingDictionary;
CFMutableDictionaryRef        subDictionary;

matchingDictionary = CFDictionaryCreateMutable ( kCFAllocatorDefault, 0, NULL,
    NULL );
subDictionary = CFDictionaryCreateMutable ( kCFAllocatorDefault, 0, NULL, NULL
    );

CFDictionarySetValue ( subDictionary, CFSTR ( kIOPropertySCSITaskDeviceCategory
    ), CFSTR ( kIOPropertySCSITaskUserClientDevice ) );

CFDictionarySetValue ( matchingDictionary, CFSTR ( kIOPropertyMatchKey ),
    subDictionary );
```

You can refine this search by matching on your device's GUID. Before you add the subdictionary to the matching dictionary, you place a key-value pair that represents the GUID in the subdictionary, as in this example:

```
CFDictionarySetValue ( subDictionary, CFSTR (
    kIOPropertySCSITaskUserClientInstanceGUID ), myGUID );
```

Accessing the Device

The steps you take to access a device are divided into two sets. The first set applies only to authoring applications. The second set applies to both application-based drivers of devices with no in-kernel logical unit drivers *and* authoring applications that have already completed the first set of steps. The sample code in "[Accessing a SCSI Architecture Model Device](#)" (page 31) accesses a CD-R/W device and follows both sets of steps.

Use these steps *only* if you're developing an authoring application. Once you have completed these steps, continue on to the second set of steps. If you're developing an application that drives a device with no in-kernel driver, skip these steps and follow the second set of steps.

1. Get the device interface `MMCDeviceInterface`. This device interface provides functions that give you information about the media in the device, such as its size and whether or not it is blank.
2. Use the `MMCDeviceInterface` functions to query the device, if necessary.
3. If a disc is currently in the drive, reserve the media.

Use these steps if you're developing an application-based driver for a device with no in-kernel logical unit driver *or* you're developing an authoring application and you've already followed the first set of steps.

1. Get the device interface `SCSITaskDeviceInterface`. This device interface provides functions to help you prepare to send `SCSITask` objects to the device. It also enforces the exclusive access policy and provides functions to add asynchronous callback mechanisms to your run loop.
2. Request exclusive access to the device. To do this, you use the `SCSITaskDeviceInterface` function `ObtainExclusiveAccess`.
3. Create a `SCSITask` object to communicate with the device. To do this you use the `SCSITaskDeviceInterface` function `CreateSCSITask`.

Accessing a SCSI Architecture Model Device

This section provides step-by-step instructions, including sample code, for accessing a CD-R/W drive. The sample code finds the device in the I/O Registry by setting up a matching dictionary like the one in "[Device Matching for Authoring-Capable Devices](#)" (page 29) and then follows the first set of steps in "[Accessing the Device](#)" (page 31) to acquire an `MMCDeviceInterface`. Then, it follows the second set of steps to get a `SCSITaskDeviceInterface` and request exclusive access.

When the sample code has exclusive access to the CD-R/W drive, it gets a `SCSITaskInterface` and sends `SCSITask` objects to test the device. The sample code also shows how to set up notifications for the dynamic addition and removal of a device.

Setting Up Your Project

The sample code in this chapter is from an Xcode project that builds a Core Foundation tool. If you need detailed documentation on using Xcode you can find it at [Guides > Tools > Xcode](#).

To set up Xcode to build the sample code, do the following:

1. Choose “Core Foundation Tool” when creating your project. This creates a skeletal `main.c` file and includes `CoreFoundation.h` in the project.
2. Use Add Frameworks... from the Projects menu to add `IOKit.framework` and `System.framework` to your project.
3. Replace the `main.c` file with [Listing 2-5](#) (page 32) through [Listing 2-14](#) (page 41), inclusive.
4. Build your code in the Mach-O executable format (Xcode’s default format).
5. It’s recommended that you start by building with debugging turned on. To do this in Xcode, open the target list by clicking the disclosure triangle. Double-click the only target listed. In the inspector that appears, click the build tab, then click the checkbox next to 'Generate Debug Symbols'.

[Listing 2-5](#) (page 32) shows the header files you’ll need to include for the sample code in this chapter.

Listing 2-5 Header files

```
#include <IOKit/IOKitLib.h>
#include <IOKit/scsi-commands/SCSITaskLib.h>
```

The sample code in this chapter uses the global variables in [Listing 2-6](#) (page 32) to set up asynchronous notifications for device addition and removal.

Listing 2-6 Global variables

```
static IONotificationPortRef gNotifyPort;
static io_iterator_t gAppearedIter;
static io_iterator_t fDisappearedIter;
```

Creating a Main Function

The sample code for working with a CD-R/W drive supported by a SCSI Architecture Model family driver begins with the `main` function, shown in [Listing 2-7](#) (page 33). This function accomplishes the following tasks.

- It sets up a signal handler (shown in [Listing 2-8](#) (page 34)) to take care of the command line interrupt used to exit the run loop.
- It establishes communication with the I/O Kit and sets up a matching dictionary to find authoring devices.
- It sets up an asynchronous notification that is called when a device is first attached to the I/O Registry and another that is called when a device is removed.
- It starts the run loop so the notifications will be received.

The `main` function uses I/O Kit functions to set up and modify a matching dictionary and set up notifications and Core Foundation functions to set up the run loop for receiving the notifications. It calls the following functions to get access to the device.

- `DeviceAppeared` (shown in [Listing 2-9](#) (page 35)) iterates over the set of matching devices and calls `TestDevice` (shown in [Listing 2-11](#) (page 36)) for each one.

- DeviceDisappeared (shown in Listing 2-10 (page 35)) iterates over the set of matching devices and releases each one in turn.

Listing 2-7 Setting up a main function to access a CD-R/W device

```
int main (int argc, const char *argv[])
{
    mach_port_t      masterPort;
    CFMutableDictionaryRef  matchingDict;
    CFMutableDictionaryRef  subDict;
    CFRunLoopSourceRef      runLoopSource;
    kern_return_t      kr;
    sig_t              oldHandler;

    // Set up a signal handler so we can clean up when we're interrupted from
the command line
    // Otherwise we stay in our run loop forever.
    oldHandler = signal(SIGINT, SignalHandler);
    if (oldHandler == SIG_ERR)
        printf("Could not establish new signal handler");

    // first create a master_port for my task
    kr = IOMasterPort(MACH_PORT_NULL, &masterPort);
    if (kr || !masterPort)
    {
        printf("ERR: Couldn't create a master IOKit Port(%08x)\n", kr);
        return -1;
    }

    // Create the dictionaries
    matchingDict = CFDictionaryCreateMutable ( kCFAllocatorDefault, 0, NULL,
NULL );
    subDict      = CFDictionaryCreateMutable ( kCFAllocatorDefault, 0, NULL,
NULL );

    // Create a dictionary with the "SCSITaskDeviceCategory" key =
"SCSITaskAuthoringDevice"
    CFDictionarySetValue ( subDict,
                           CFSTR ( kIOPropertySCSITaskDeviceCategory ),
                           CFSTR ( kIOPropertySCSITaskAuthoringDevice ) );

    // Add the dictionary to the main dictionary with the key "IOPropertyMatch"
to
    // narrow the search to the above dictionary.
    CFDictionarySetValue ( matchingDict,
                           CFSTR ( kIOPropertyMatchKey ),
                           subDict );

    // Create a notification port and add its run loop event source to our run
loop
    // This is how async notifications get set up.
    gNotifyPort = IONotificationPortCreate(masterPort);
    runLoopSource = IONotificationPortGetRunLoopSource(gNotifyPort);

    CFRunLoopAddSource(CFRunLoopGetCurrent(), runLoopSource,
kCFRunLoopDefaultMode);
}
```

```

    // Retain a reference since we arm both the appearance and disappearance
    notifications
    // and the call to IOServiceAddMatchingNotification() consumes a reference
    each time.
    matchingDict = ( CFMutableDictionaryRef ) CFRetain ( matchingDict );

    // Now set up two notifications, one to be called when a raw device is first
    matched by I/O Kit, and the other to be
    // called when the device is terminated.
    kr = IOServiceAddMatchingNotification( gNotifyPort,
                                           kIOFirstMatchNotification,
                                           matchingDict,
                                           DeviceAppeared,
                                           NULL,
                                           &gAppearedIter );

    DeviceAppeared(NULL, gAppearedIter); // Iterate once to get already-present
    devices and

                                           // arm the notification

    kr = IOServiceAddMatchingNotification( gNotifyPort,
                                           kIOTerminatedNotification,
                                           matchingDict,
                                           DeviceDisappeared,
                                           NULL,
                                           &gDisappearedIter );

    DeviceDisappeared(NULL, gDisappearedIter); // Iterate once to arm the
    notification

    // Now done with the master_port
    mach_port_deallocate(mach_task_self(), masterPort);
    masterPort = 0;

    // Start the run loop. Now we'll receive notifications.
    CFRunLoopRun();

    // We should never get here
    return 0;
}

```

When you press Control-C to stop the run loop and exit the program, some objects may still be in use. The `SignalHandler` function, shown in [Listing 2-8](#) (page 34), releases these objects and exits.

Listing 2-8 Handling command-line interrupts

```

void SignalHandler(int sigraised)
{
    printf("\nInterrupted\n");

    // Clean up here
    IONotificationPortDestroy(gNotifyPort);

    if (gAppearedIter)
    {
        IOObjectRelease(gAppearedIter);
        gAppearedIter = 0;
    }
}

```

```

    if (gDisappearedIter)
    {
        IOObjectRelease(gDisappearedIter);
        gDisappearedIter = 0;
    }

    exit(0);
}

```

Now that you've obtained an iterator for a set of matching devices, you can use it to gain access to each device. The function `DeviceAppeared` (shown in [Listing 2-9](#) (page 35)) uses the I/O Kit function `IOIteratorNext` to access each device object and then calls the function `TestDevice` (shown in [Listing 2-11](#) (page 36)) to create device interfaces for the device and test it.

Listing 2-9 Accessing the set of matching devices

```

void DeviceAppeared(void *refCon, io_iterator_t iterator)
{
    kern_return_t    kr;
    io_service_t     obj;

    while (obj = IOIteratorNext(iterator))
    {
        printf("Device appeared.\n");

        TestDevice(obj);

        kr = IOObjectRelease(obj);
    }
}

```

The function `DeviceDisappeared` (shown in [Listing 2-10](#) (page 35)) simply uses the iterator obtained from the main function (shown in [Listing 2-7](#) (page 33)) to release each device object. This also has the effect of arming the device termination notification so it will notify the program of future device removals.

Listing 2-10 Releasing the device objects

```

void DeviceDisappeared(void *refCon, io_iterator_t iterator)
{
    kern_return_t    kr;
    io_service_t     obj;

    while (obj = IOIteratorNext(iterator))
    {
        printf("Device disappeared.\n");
        kr = IOObjectRelease(obj);
    }
}

```

Testing the Device

The function `TestDevice` (shown in [Listing 2-11](#) (page 36)) creates an intermediate plug-in interface for the device and then queries this interface to obtain an `MMCDeviceInterface`. The sample code does not use the `MMCDeviceInterface` to query the device although functions such as `GetPerformance` and `GetTrayState` are available.

The function `TestDevice` uses I/O Kit functions to create the `MMCDeviceInterface` and then calls the `MMCDeviceInterface` function `GetSCSITaskDeviceInterface` to create the `SCSITaskDeviceInterface`. Once it gets the `SCSITaskDeviceInterface`, it calls its function `ObtainExclusiveAccess` to request exclusive access to the device. After receiving exclusive access, `TestDevice` calls the following functions to create `SCSITask` objects and send them to the device.

- `Inquiry`, (shown in [Listing 2-12](#) (page 37)), creates a `SCSITask` object to send an INQUIRY command to the device and prints out the inquiry data it receives.
- `TestUnitReady`, (shown in [Listing 2-13](#) (page 40)), creates a `SCSITask` object to send a TEST_UNIT_READY command to the device and calls `PrintSenseString` (shown in [Listing 2-14](#) (page 41)) to print the sense string the device returns.

`TestDevice` then relinquishes exclusive access to the device and releases the `SCSITaskDeviceInterface` and `MMCDeviceInterface`.

Listing 2-11 Getting an `MMCDeviceInterface` and obtaining exclusive access

```
void TestDevice(io_service_t service)
{
    SInt32                score;
    HRESULT               herr;
    kern_return_t         err;
    IOCFPlugInInterface  **plugInInterface = NULL;
    MMCDeviceInterface    **mmcInterface = NULL;
    SCSITaskDeviceInterface **interface = NULL;

    // Create the IOCFPlugIn interface so we can query it.
    err = IOCreatePlugInInterfaceForService ( service,
                                              kIOMMCDeviceUserClientTypeID,
                                              kIOCFPlugInInterfaceID,
                                              &plugInInterface,
                                              &score );

    if ( err != noErr )
    {
        printf("IOCreatePlugInInterfaceForService returned %d\n", err);
        return;
    }

    // Query the interface for the MMCDeviceInterface.
    herr = ( *plugInInterface )->QueryInterface ( plugInInterface,
                                                  CFUUIDGetUUIDBytes (
kIOMMCDeviceInterfaceID ),
                                                  ( LPVOID *) &mmcInterface );

    if ( herr != S_OK )
    {
        printf("QueryInterface returned %ld\n", herr);
        return;
    }
}
```

```

    }

    interface = ( *mmcInterface )->GetSCSITaskDeviceInterface ( mmcInterface );
    if ( interface == NULL )
    {
        printf("GetSCSITaskDeviceInterface returned NULL\n");
        return;
    }

    err = ( *interface )->ObtainExclusiveAccess ( interface );
    if ( err != noErr )
    {
        printf("ObtainExclusiveAccess returned %d\n", err);
        return;
    }

    Inquiry(interface);
    TestUnitReady(interface);

    ( *interface )->ReleaseExclusiveAccess ( interface );

    // Release the SCSITaskDeviceInterface.
    ( *interface )->Release ( interface );
    ( *mmcInterface )->Release ( mmcInterface );

    IODestroyPlugInInterface ( plugInInterface );
}

```

Now that you've obtained exclusive access, you can create `SCSITask` objects and send them to the device. The function `Inquiry` (shown in [Listing 2-12](#) (page 37)) uses the `SCSITaskDeviceInterface` function `CreateSCSITask` to create a `SCSITaskInterface` object. It then uses `SCSITaskInterface` functions to prepare the object to send an `INQUIRY` command. If the command executes successfully, `Inquiry` prints the results. Finally, `Inquiry` releases the `SCSITaskInterface` object and returns.

Listing 2-12 Sending an `INQUIRY` command to the device

```

void Inquiry(SCSITaskDeviceInterface **interface)
{
    SCSICommand_INQUIRY_StandardData    inqBuffer;
    SCSITaskStatus                       taskStatus;
    SCSI_Sense_Data                      senseData;
    SCSICommandDescriptorBlock           cdb;
    SCSITaskInterface **                 task = NULL;
    UInt8 *                               bufPtr = ( UInt8 * ) &inqBuffer;
    char                                  vendorID[9];
    char                                  productID[17];
    char                                  firmwareRevLevel[5];
    IOReturn                              err = 0;
    UInt32                                 index = 0;
    IOVirtualRange *                     range = NULL;
    UInt64                                 transferCount = 0;
    UInt32                                 transferCountHi = 0;
    UInt32                                 transferCountLo = 0;

    // Create a task now that we have exclusive access
    task = ( *interface )->CreateSCSITask ( interface );
}

```

```

if ( task != NULL )
{
    // Zero the buffer.
    memset ( bufPtr, 0, sizeof ( SCsICmd_INQUIRY_StandardData ) );

    // Allocate a virtual range for the buffer. If we had more than 1
scatter-gather entry,
    // we would allocate more than 1 IOVirtualRange.
    range = ( IOVirtualRange * ) malloc ( sizeof ( IOVirtualRange ) );
    if ( range == NULL )
    {
        printf("***** ERROR Malloc'ing IOVirtualRange *****\n\n");
    }

    // zero the senseData and CDB
    memset ( &senseData, 0, sizeof ( senseData ) );
    memset ( cdb, 0, sizeof ( cdb ) );

    // Set up the range. The address is just the buffer's address. The length
is our request size.
    range->address = ( IOVirtualAddress ) bufPtr;
    range->length = sizeof ( SCsICmd_INQUIRY_StandardData );

    // We're going to execute an INQUIRY to the device as a
    // test of exclusive commands.
    cdb[0] = 0x12 /* inquiry */;
    cdb[4] = sizeof ( SCsICmd_INQUIRY_StandardData );

    // Set the actual CDB in the task
    err = ( *task )->SetCommandDescriptorBlock ( task, cdb, kSCsICDBSize_6Byte
);
    if ( err != kIOReturnSuccess )
    {
        printf("***** ERROR Setting CDB *****\n\n");
    }

    // Set the scatter-gather entry in the task
    err = ( *task )->SetScatterGatherEntries ( task, range, 1, sizeof (
SCsICmd_INQUIRY_StandardData ),
kSCsIDataTransfer_FromTargetToInitiator );
    if ( err != kIOReturnSuccess )
    {
        printf("***** ERROR Setting SG Entries *****\n\n");
    }

    // Set the timeout in the task
    err = ( *task )->SetTimeoutDuration ( task, 10000 );
    if ( err != kIOReturnSuccess )
    {
        printf("***** ERROR Setting Timeout *****\n\n");
    }

    printf("***** Requesting Inquiry Data *****\n\n");

    // Send it!

```

```

    err = ( *task )->ExecuteTaskSync ( task, &senseData, &taskStatus,
&transferCount );
    if ( err != kIOReturnSuccess )
    {
        printf("***** ERROR Executing Task *****\n\n");
    }

    // Get the transfer counts
    transferCountHi = ( ( transferCount >> 32 ) & 0xFFFFFFFF );
    transferCountLo = ( transferCount & 0xFFFFFFFF );

    printf("taskStatus = %d, transferCountHi = 0x%08lx, transferCountLo =
0x%08lx\n", taskStatus, transferCountHi, transferCountLo);

    // Task status is not GOOD, print any sense string if they apply.
    if ( taskStatus == kSCSITaskStatus_GOOD )
    {

        printf("***** INQUIRY DATA *****\n\n");

        printf("Peripheral Device Type = %d\n", bufPtr[0] & 0x1F);
        printf("Removable Media Bit = %d\n", ( bufPtr[1] & 0x80 ) == 0x80);

        for ( index = 8; index < 16; index++ )
        {
            if ( bufPtr[index] == 0 )
                break;
            vendorID[index-8] = bufPtr[index];
        }
        vendorID[index-8] = 0;

        for ( index = 16; index < 32; index++ )
        {
            if ( bufPtr[index] == 0 )
                break;
            productID[index-16] = bufPtr[index];
        }
        productID[index-16] = 0;

        for ( index = 32; index < 36; index++ )
        {
            if ( bufPtr[index] == 0 )
                break;
            firmwareRevLevel[index-32] = bufPtr[index];
        }
        firmwareRevLevel[index-32] = 0;

        printf("Vendor Identification = %s\n", ( char * ) vendorID);
        printf("Product Identification = %s\n", ( char * ) productID);
        printf("Product Revision Level = %s\n", ( char * ) firmwareRevLevel);

        printf("\n");

    }

    // Be a good citizen and cleanup
    free ( range );

```

```

        // Release the task interface
        ( *task )->Release ( task );

    }

}

```

The function `TestUnitReady` (shown in [Listing 2-13](#) (page 40)) sends a `TEST_UNIT_READY` command to the device. First, it creates a `SCSITaskInterface` object and then it creates a CDB that contains the command. After sending the command, `TestUnitReady` checks the task status and, depending on the status value, calls the function `PrintSenseString` (shown in [Listing 2-14](#) (page 41)) to print the sense data. Before returning, `TestUnitReady` releases the `SCSITaskInterface` object.

Listing 2-13 Sending a `TEST_UNIT_READY` command to the device

```

void TestUnitReady(SCSITaskDeviceInterface **interface)
{
    SCSITaskStatus          taskStatus;
    SCSI_Sense_Data         senseData;
    SCSICommandDescriptorBlock cdb;
    SCSITaskInterface **    task = NULL;
    IOReturn                err = 0;
    UInt64                  transferCount = 0;

    // Create a task now that we have exclusive access
    task = ( *interface )->CreateSCSITask ( interface );
    if ( task != NULL )
    {
        // zero the senseData and CDB
        memset ( &senseData, 0, sizeof ( senseData ) );
        memset ( cdb, 0, sizeof ( cdb ) );

        // The TEST_UNIT_READY code consists of all zeroes so it is
        // not necessary to set any additional values in the CDB
        // Set the actual CDB in the task
        err = ( *task )->SetCommandDescriptorBlock ( task, cdb, kSCSICDBSize_6Byte
    );
        if ( err != kIOReturnSuccess )
        {
            printf("***** ERROR Setting CDB *****\n\n");
        }

        // Set the timeout in the task
        err = ( *task )->SetTimeoutDuration ( task, 5000 );
        if ( err != kIOReturnSuccess )
        {
            printf("***** ERROR Setting Timeout *****\n\n");
        }

        // Send it!
        err = ( *task )->ExecuteTaskSync ( task, &senseData, &taskStatus,
&transferCount );
        if ( err != kIOReturnSuccess )
        {
            printf("***** ERROR Executing Task *****\n\n");
        }
    }
}

```

```

printf("taskStatus = %d\n", taskStatus);

// Task status is not GOOD, print any sense string if they apply.
if ( taskStatus == kSCSITaskStatus_GOOD )
{
    printf("Good Status\n");
}
else if ( taskStatus == kSCSITaskStatus_CHECK_CONDITION )
{
    // Something happened. Print the sense string
    PrintSenseString(&senseData);
}
else
{
    printf("taskStatus = 0x%08x\n", taskStatus);
}

printf("\n");

// Release the task interface
( *task )->Release ( task );
}
}

```

The function `PrintSenseString` (shown in [Listing 2-14](#) (page 41)) prints the sense data the device returns after executing the `TEST_UNIT_READY` command sent in `TestUnitReady` (shown in [Listing 2-13](#) (page 40)).

Listing 2-14 Printing the sense data

```

void
PrintSenseString ( SCSI_Sense_Data * sense )
{
    char    str[256];
    UInt8   key, ASC, ASCQ;

    key     = sense->SENSE_KEY & 0x0F;
    ASC     = sense->ADDITIONAL_SENSE_CODE;
    ASCQ    = sense->ADDITIONAL_SENSE_CODE_QUALIFIER;

    // Print the sense string information.
    sprintf ( str, "Key: %02lx, ASC: %02lx, ASCQ: %02lx  ", key, ASC, ASCQ
);
}

```


Document Revision History

This table describes the changes to *SCSI Architecture Model Device Interface Guide*.

Date	Notes
2007-02-08	Made minor corrections.
2005-11-09	Added information to emphasize what types of commands applications can and cannot send to storage devices.
2005-09-08	Added information about endian issues. Changed title from "Working With SCSI Architecture Model Devices."
2003-06-12	The chapter " Accessing SCSI Parallel Devices " (page 9) was revised to use a new code sample (the <code>SCSI01dAndNew</code> project, available at http://developer.apple.com/samplecode/Sample_Code/Devices_and_Hardware.htm) that demonstrates how to use the APIs of both the deprecated SCSI family and the SCSI Architecture Model family to find and access devices.
2003-05-01	Preliminary version of <i>Working With SCSI Parallel and SCSI Architecture Model Devices</i> . This document comprises two chapters from an earlier version of <i>Accessing Hardware From Applications</i> and includes additional information on how to choose the correct API to work with SCSI Parallel devices.

REVISION HISTORY

Document Revision History