Accessing Hardware From Applications

Hardware & Drivers

2007-02-08

Apple Inc.
© 2001, 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, eMac, FireWire, Mac, Mac OS, Objective-C, Pages, Quartz, QuickDraw, QuickTime, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

CDB is a trademark of Third Eye Software, Inc.

UNIX is a registered trademark of The Open Group

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 Accessing Hardware From Applications 7**

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

Chapter 1 **Hardware-Access Options 9**

Other APIs That Provide Access to Hardware 9
Where to Go From Here 11

Chapter 2 **Device Access and the I/O Kit 13**

I/O Kit Summary 13
- I/O Kit Definitions 13
- I/O Kit Driver-Stack Building 16

Device Interfaces and Device Files 19
- Inside the Device-Interface Mechanism 19
- Inside the Device-File Mechanism 21
- Communicating With the I/O Kit 23

Chapter 3 **Finding and Accessing Devices 25**

Finding Devices in the I/O Registry 25
- Device Matching 26
- Getting the I/O Kit Master Port 26
- Getting Keys and Values for a Device-Matching Dictionary 27
- Setting Up a Matching Dictionary to Find Devices 31
- Setting Up a Matching Dictionary to Find Device Files 33
- Looking Up Devices in the I/O Registry 34
- Getting Notifications of Device Arrival and Departure 35

Examining Matching Objects 36
Putting It All Together: Accessing a Device 36
- Getting a Device Interface 37
- Getting a Device-File Path 38

Chapter 4 **The IOKitLib API 41**

Object Reference-Counting and Introspection 41
- Reference Counting 42
- Introspection 43

Device Discovery and Notification 43

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Creating Matching Dictionaries 44
Looking Up Devices 45
Setting Up and Receiving Notifications 45
Iterating Over Matching Devices 46
I/O Registry Access 47
Traversing the I/O Registry 47
Getting Information About I/O Registry Objects 48
Viewing Properties of I/O Registry Objects 49
Setting Properties of I/O Registry Objects 50
Determining Busy States 51
Device-Interface Development 52
Creating a User Space–Kernel Connection 52
Managing the User Space–Kernel Connection 55

Chapter 5 **Handling Errors 57**

Interpreting I/O Kit Error Return Values 57
Handling Exclusive-Access Errors 59

Appendix A **I/O Kit Family Device-Access Support 61**

Document Revision History 65

Glossary 67

Index 69

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Figures, Tables, and Listings

Chapter 2 **Device Access and the I/O Kit 13**

Figure 2-1 The I/O Kit framework 15
Figure 2-2 I/O Kit objects supporting a FireWire device 16
Figure 2-3 I/O Kit objects supporting a FireWire device in I/O Registry Explorer 17
Figure 2-4 Adding a device interface to the FireWire driver stack 18
Figure 2-5 I/O Kit objects supporting a serial device 18
Figure 2-6 I/O Kit objects supporting a serial device in I/O Registry Explorer 19
Figure 2-7 The IOCFPlugInInterface functions 20
Figure 2-8 Some of the IOUSBDeviceInterface functions 21
Figure 2-9 An IOSerialBSDClient object in I/O Registry Explorer 22

Chapter 3 **Finding and Accessing Devices 25**

Figure 3-1 The I/O Registry Explorer application, showing various keys and values for the AppleUSBComposite driver 28
Figure 3-2 Some keys and values for a HID class device, shown in I/O Registry Explorer 29
Listing 3-1 Matching keys from IOKitKeys.h 28
Listing 3-2 HID class device matching keys from IOHIDKeys.h 28
Listing 3-3 A partial listing of the personality dictionary for the AppleFWAudio driver 30
Listing 3-4 Modifying a matching dictionary 32
Listing 3-5 Device interface definitions from IOUSBLib.h 37
Listing 3-6 Getting an intermediate IOCFPlugInInterface object 38
Listing 3-7 Getting a specific device interface object 38
Listing 3-8 Getting the device name of a storage device 39

Chapter 4 **The IOKitLib API 41**

Table 4-1 Dictionary-creation functions and the keys they use 44
Table 4-2 `IOConnectMethod` functions 53
Listing 4-1 Object definitions in IOTypes.h 41
Listing 4-2 Releasing the underlying kernel object of an `io_iterator_t` object 42
Listing 4-3 Getting the retain count of an `io_object_t` object 43
Listing 4-4 Using the `IORegistryEntryGet` functions 49
Listing 4-5 Creating a user client 52
Listing 4-6 Requesting I/O with the `IOConnectMethodScalarIStructureI` function 54

Chapter 5 **Handling Errors 57**

Figure 5-1 Bit layout for kernel and I/O Kit error return values 57
Listing 5-1 Error.h system error values 58
Listing 5-2 Error.h macros for working with error return values 58

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Listing 5-3 IOReturn.h error return values 58

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Introduction to Accessing Hardware From Applications

There are many reasons your application might need to access hardware: Receiving mouse and keyboard events, accessing devices, such as a FireWire DV camcorder, and driving a device from an application are just a few. Although only code that resides in the kernel can access hardware directly, Mac OS X provides many services that allow you to communicate with hardware from plug-ins, applications, shared libraries, and other code running outside the kernel.

Who Should Read This Document?

This document describes how software running in Mac OS X can access hardware by communicating with the kernel, focusing on services the I/O Kit provides to develop an application-based driver. You should read this document if you need to access a device from an application. Note that many applications will be able to handle all their hardware-access needs using high-level APIs, such as Open Transport and QuickTime, that are available through Carbon and Cocoa. To help you determine which approach is right for you, and for a summary of other services Mac OS X provides for hardware access from applications, see "Hardware-Access Options" (page 9).

This document includes many code fragments illustrating the various tasks involved in developing an application that accesses hardware, but it is not intended to be a step-by-step cookbook for accessing a particular type of device. To determine how to access a particular device, see "I/O Kit Family Device-Access Support" (page 61); For each device family it specifies how to access a device in that family and where to find more detailed documentation.

This document does *not* describe how to write kernel-resident code to access hardware. Kernel programmers should refer to *Kernel Programming Guide* and in-kernel device-driver developers should read *I/O Kit Device Driver Design Guidelines*. In particular, if you are developing your own device interface and user client to create a custom solution to access your device, you should read the "Making Hardware Accessible to Applications" chapter in that document. For other documents that cover how to access particular devices, visit Reference Library > Hardware & Drivers.

Organization of This Document

Accessing Hardware From Applications includes the following chapters:

- "Hardware-Access Options" (page 9)

  Describes many other methods you can use to access hardware from an application, such as Core Audio, QuickTime, and the Carbon Event Manager. Read this chapter to determine if such high-level APIs can meet your needs.

- "Device Access and the I/O Kit" (page 13)

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Summarizes I/O Kit architecture, providing a list of terms used throughout this document and describing how the I/O Kit models I/O connections. It then describes the two fundamental hardware-access methods the I/O Kit supports: device interfaces and device files.

- "Finding and Accessing Devices" (page 25)

  Describes the steps you take to access a device using an I/O Kit device interface and, for appropriate devices, using a device file.

- "The IOKitLib API" (page 41)

  Categorizes and describes the functions of the main API that supports user-space device access through the I/O Kit.

- "Handling Errors" (page 57)

  Describes how to interpret I/O Kit return values and provides information on the exclusive-access error.

- "I/O Kit Family Device-Access Support" (page 61)

  Lists the current I/O Kit families and describes what support they provide for hardware access from applications.

- "Document Revision History" (page 65)

  Lists changes to this document.

- "Glossary" (page 67)

  Defines key terms used in this document.

See Also

Although this document includes a summary of basic I/O Kit information (in "I/O Kit Summary" (page 13)), you should read *I/O Kit Fundamentals* for a thorough understanding of this subsystem.

Familiarity with the CFPlugIn architecture is useful in reading this document. This architecture is described in the developer documentation available in Reference Library > Core Foundation.

Knowledge of the Mac OS X kernel and device drivers may be useful but is not required. To get more information about these topics, visit Reference Library > Darwin.

When you install the Developer package, you get developer documentation as well as tools and example code. You can find all the I/O Kit and kernel documents mentioned in this document listed in `/Developer/ADC Reference Library/documentation/Darwin` and `/Developer/ADC Reference Library/documentation/HardwareDrivers`. Sample projects are available in `/Developer/Examples`. Most of the sample projects that are relevant to device access from applications reside in `/Developer/Examples/IOKit`.

There you can view the documentation for BSD and POSIX functions and tools by typing `man` *function_name* in a Terminal window (for example, `man gdb`) or in HTML at Mac OS X Man Pages.

You can access reference documentation on I/O Kit families from Xcode, Help Viewer, and Reference Library > Hardware & Drivers. Of course, you can also browse the header files for various I/O Kit families and other I/O Kit services accessible from user space in `/System/Library/Frameworks/IOKit.framework/Headers`.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Hardware-Access Options

Many applications can handle all their hardware-access needs using high-level APIs, such as QuickTime, CFNetwork, and Core Audio. Before you embark upon the development of an application-based device driver, you should read this chapter to determine if there is a more suitable (and probably easier) solution.

If you find that you must use I/O Kit or BSD APIs to access a device, note that Apple does not provide Objective-C interfaces for these APIs. However, because these are pure C APIs, you can call them from your Cocoa application.

Other APIs That Provide Access to Hardware

The high-level Mac OS X APIs listed here provide some access to hardware and do not require the use of I/O Kit services. This list is not exhaustive. A pointer to documentation for these APIs is provided, where available; otherwise, look for the for the latest documentation in the Reference Library.

- BSD networking

  You can access networking services from user space using the BSD socket API. The `socket` structure is used to keep track of network information on a per file-descriptor basis: Applications can reference the `socket` structure using file descriptors.

  For more information on the BSD socket API, refer to the `man` pages. In a Terminal window, type `man socket`. To view the man pages in HTML, see Mac OS X Man Pages.

- BSD file systems

  You can use POSIX file I/O functions to access disks represented as device files. These functions use a file descriptor, an integer index into a list of files a process has open, to access files. You can manipulate file descriptors using the standard file descriptor functions, such as `open`, `close`, `read`, and `write`. In addition, you can use the `fcntl` and `ioctl` functions to control files and devices.

  For more information on these functions, refer to the `man` pages. For example, in a Terminal window, type `man open` or `man ioctl`. To view the man pages in HTML, see Mac OS X Man Pages. See *Device File Access Guide for Serial Devices* and *Device File Access Guide for Storage Devices* for examples of how to use file-descriptor functions to access these devices.

- Carbon Event Manager

  The Carbon Event Manager provides routines that communicate user actions and give notice of changes in processing status. You can use the Carbon Event Manager, for example, to obtain information about mouse and keyboard events.

  For more information about the Carbon Event manager, see the Carbon Events and Other Input Reference Library.

- Carbon Printing Manager

  The Carbon Printing Manager defines an API that allows applications to print in both Mac OS 8 and 9 with existing printer drivers and in Mac OS X with new printer drivers.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

For more information on how any application (even non-Carbon ones) can use the functions of the Carbon Printing Manager, see the Carbon Printing Reference Library.

- CFNetwork

  The CFNetwork API provides abstractions that allow you to easily work with BSD sockets, manage information about remote hosts, and work with HTTP and FTP servers.

  For more information about the CFNetwork API, see *CFNetwork Programming Guide*.

- Core Audio

  The Audio HAL (Hardware Abstraction Layer) is at the heart of the Core Audio system and provides the interface between applications and hardware. The Audio HAL allows applications to manipulate devices through both an input/output procedure for streaming and a property mechanism for control.

  For more information about the Audio HAL in particular and Core Audio in general, see the Music & Audio Reference Library.

- Quartz 2D and Quartz Compositor

  Quartz Compositor (sometimes referred to as "Core Graphics Services" in older documentation) consists of the Mac OS X window server and the private system programming interfaces (SPIs) it implements. It creates the Mac OS X graphical user interface by compositing content from client graphics-rendering libraries, such as Quartz 2D and QuickDraw.

  For more information about Quartz Compositor and about using Quartz 2D, see the Graphics & Imaging Reference Library.

- File Manager

  The File Manager allows your application to access files and folders on physical storage devices such as disk drives. It supports several volume formats, including HFS and HFS Plus.

  For more information on how to use the File Manager, see the Carbon File Management Reference Library.

- NSEvent (Cocoa object class)

  An NSEvent object, or simply an event, contains information about an input action such as a mouse click or a key down. The Cocoa Application Kit associates each such user action with a window, reporting the event to the application that created the window.

  In Mac OS X version 10.4, applications can receive tablet events, such as tablet-pointing and tablet-proximity events, as NSEvent objects.

  For more information about Cocoa and the Application Kit, see the Cocoa Reference Library.

- Open Transport

  Open Transport is the Mac OS 8 and 9 API for accessing TCP/IP networks at the transport level. Apple provides Open Transport as a compatibility library to support migration of legacy applications to Mac OS X. New Mac OS X applications should instead use BSD sockets or higher-level Core Services and Core Foundation APIs, such as CFNetwork.

  For more information on Open Transport, see the Carbon Networking Reference Library.

- Power-source information

  You can get information about power sources and UPS (uninterruptible power supply) devices using the I/O Kit's power-source API located in `/System/Library/Frameworks/IOKit.framework/Headers/ps`. The header files in this folder,

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

`IOPowerSources.h` and `IOPSKeys.h`, contain methods and keys to extract information about both external and internal power sources. For example, an application can get a list of attached power sources, request notifications for changes in its power sources, and determine how much power is left in a battery.

- QuickTime

  QuickTime is a package of system-level code that higher-level software can use to control time-based data. QuickTime can handle video data, still images, animated images (sprites), vector graphics, multiple sound channels, MIDI music, 3D objects, virtual reality panoramas and objects, and even text. For example, you can use the QuickTime video digitizer and video output components to access FireWire DV devices such as DV camcorders.

  For more information on QuickTime, see the QuickTime Reference Library.

  (For further information on Apple's FireWire support, including access to development kits with device-interface support, see http://developer.apple.com/hardwaredrivers/firewire/.)

- SCSI Manager 4.3

  Beginning with Mac OS X version 10.2, the SCSI Manager 4.3 `SCSIAction` function is deprecated, although it will continue to function on previous versions of Mac OS X until users install SCSI HBA (host bus adapter) drivers developed with the new SCSI Parallel family API (shipped with Mac OS X version 10.2).

  If your application must access a SCSI Parallel device that was not previously accessible with the SCSI Architecture Model family's device interfaces and your application requires compatibility with versions of Mac OS X earlier than 10.2, your application should look up SCSI devices using both the new IOSCSITaskDeviceInterface and the old IOSCSIDeviceInterface APIs. If a user has installed the new HBA drivers, the new IOSCSITaskDeviceInterface API will find the device and your code can use that interface to communicate with the device. If a function of the old API succeeds in finding the device, it's because the user hasn't yet installed the new HBA drivers and your code should use the old API to communicate with the device.

  For more information on how to access SCSI devices, see *SCSI Architecture Model Device Interface Guide*.

Where to Go From Here

If you've determined that the hardware access options listed in "Other APIs That Provide Access to Hardware" (page 9) do not meet your application's needs, you can use the device interface mechanism that many I/O Kit device families provide to access your device.

If you're unfamiliar with the device interface mechanism, be sure to read the next chapter, "Device Access and the I/O Kit" (page 13), for more information. If you're wondering if an I/O Kit family provides a device interface for your device, see "I/O Kit Family Device-Access Support" (page 61).

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Device Access and the I/O Kit

In Mac OS X, kernel space is the protected memory partition in which the kernel resides, while user space is memory outside the kernel's partition. Most device drivers reside in kernel space, typically because they take primary interrupts (which requires them to live in the kernel) or because their primary client resides in the kernel (such as a device driver for an Ethernet card that resides in the kernel because the network stacks reside there).

Because only code running in the kernel can directly access hardware devices, Mac OS X provides two mechanisms that allow your application or other user-space code to make use of kernel-resident drivers and other kernel services. These mechanisms are I/O Kit device interfaces and POSIX support, using device files.

This chapter summarizes fundamental I/O Kit concepts and terms and describes some of the actions the I/O Kit takes to support devices attached to a Mac OS X computer. Then, it introduces device interfaces and device files, describing how they work and where they fit into the I/O Kit's layered, runtime architecture.

I/O Kit Summary

As the object-oriented framework for device-driver development for Mac OS X, the I/O Kit defines objects that represent the various hardware and software entities that form I/O connections in a running Mac OS X system. Using these objects, the I/O Kit models the layered, provider-client relationships between devices, drivers, and driver families.

The I/O Kit also provides services, accessible through a procedural interface, for obtaining device information and accessing devices from non-kernel code. By using this interface, you can obtain the hardware support your application needs without taking on the complexity of writing kernel-resident code.

This section first defines I/O Kit terms that describe the objects you find in a running Mac OS X system and some of the processes that act on them. Then, it summarizes the device-discovery process and how the use of device interfaces affects the layered architecture of the running system. For more in-depth coverage of the I/O Kit, see *I/O Kit Fundamentals*.

I/O Kit Definitions

The I/O Kit defines objects that represent both devices and the software that supports them. To work with device interfaces or device files, you should be familiar with the I/O Kit's object-oriented view of the devices and software that make up a running Mac OS X system. The following list describes the I/O Kit elements, processes, and data structures you'll encounter in the rest of this document.

- A **family** (or device family) is a collection of software abstractions that are common to all devices of a particular category. Families provide functionality and services to drivers (defined next). The I/O Kit defines families for bus protocols (such as USB and FireWire), storage devices, human interface devices, and many others. For a full list of families and what types of application-based access they support, see "I/O Kit Family Device-Access Support" (page 61).

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

- A **driver** is an I/O Kit object that manages a specific piece of hardware. When the I/O Kit loads a driver, it may need to load one or more families, too. Drivers are written as kernel extensions (or KEXTs) and are usually installed in the directory `/System/Library/Extensions`.

  Many of a driver's characteristics are found in its property list, a text file in XML (Extensible Markup Language) format that describes the contents, settings, and requirements of the driver. A driver usually stores its property list (or `Info.plist` file) in its `Contents` directory, where you can view it using the Property List Editor.

- A **nub** is an I/O Kit object that represents a detected, controllable entity, such as a device or logical service. A nub may represent a bus, a disk, a graphics adapter, or any number of similar entities. When it supports a specific piece of hardware, a nub can also be a driver.

  A nub supports dynamic configuration by providing a connection match point between two drivers (and, by extension, between two families). A nub can also provide services to code running in user space through a device interface (defined below).

- A **service** is an I/O Kit entity, based on a subclass of IOService, that provides certain capabilities to other I/O Kit objects. All driver and nub classes inherit from the IOService class. In the I/O Kit's layered architecture, each layer is a client of the layer below it and a provider of services to the layer above it. A family, nub, or driver can be a service provider to other I/O Kit objects.

- A **device interface** is a user-space library or plug-in that provides an interface that an application can use to communicate with or control a device. A device interface communicates with its in-kernel counterpart, called a user client (defined next), to transmit commands from an application or user-space process to the in-kernel object that represents a device. A family that provides a device interface also provides a user client to handle the kernel-side communication with the device. For information on which I/O Kit families provide the device interface–user client mechanism, see "I/O Kit Family Device-Access Support" (page 61).

- A **user client** is an in-kernel object that inherits from IOService and provides a connection between an in-kernel device driver or device nub and an application or process in user space. Some documentation uses the term "user client" to refer to the combination of the user-space device interface and the in-kernel user client object, but for this discussion, "user client" refers to the in-kernel object alone.

- The **I/O Registry** is a dynamic database that describes a collection of "live" objects, each of which represents an I/O Kit entity (such as a driver, family, or nub). You can think of the I/O Registry as a tree with a nub representing the computer's main logic board at the root, and various device drivers and nubs as the leaves. In response to the addition or removal of hardware or other changes in the system, the I/O Registry automatically updates to reflect the current hardware configuration. (For an in-depth look at this structure, see "The I/O Registry" in *I/O Kit Fundamentals*.)

  The developer version of Mac OS X includes I/O Registry Explorer (located in `/Developer/Applications`), an application that allows you to examine the I/O Registry of your currently running system. There is also a command-line tool called `ioreg` that you can run in a Terminal window to display current I/O Registry information (the Terminal application is located in `/Applications/Utilities`). For more information about how to use `ioreg`, type `man ioreg` in a Terminal window.

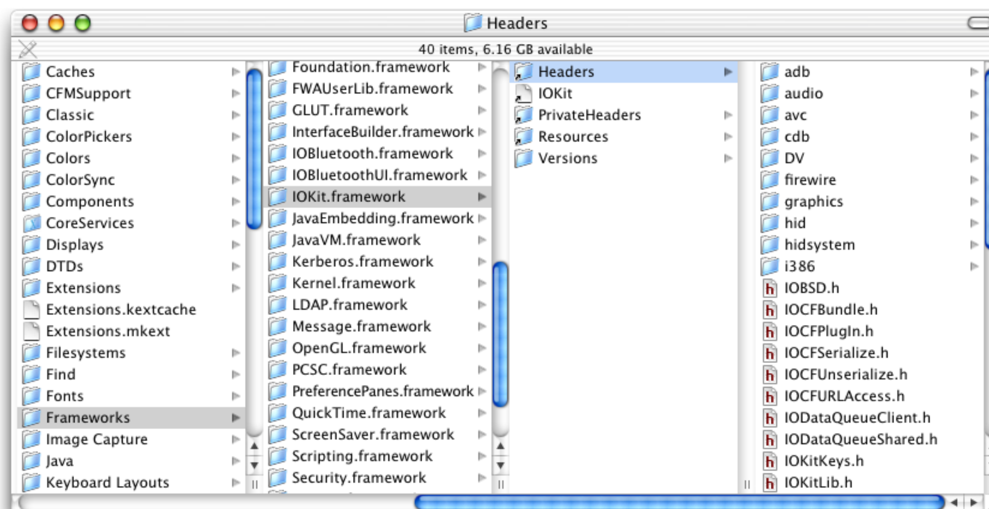
- A **driver personality** is a dictionary of key-value pairs that specify device property values, such as family type, vendor name, or product name. A driver is suitable for any device whose properties match one of the driver's personalities. In its `Info.plist` file, a driver stores its personalities as values of the `IOKitPersonalities` key.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

- **Driver matching** is the process the I/O Kit performs at boot time (and whenever the system's hardware configuration changes) to find in-kernel device drivers for all devices currently attached to the system. When it finds the driver personality that is most suitable for a particular device, the I/O Kit instantiates that personality, places a copy of its personality dictionary in the I/O Registry, and (in most cases) starts the driver.
- A **matching dictionary** is a dictionary of key-value pairs that describe the properties of a device or other service. You create a matching dictionary to specify the types of devices your application needs to access. The I/O Kit provides several general keys you can use in your matching dictionary and many device families define specific keys and matching protocols. During device matching (described next) the values in a matching dictionary are compared against nub properties in the I/O Registry.
- **Device matching** is the process of searching the I/O Registry for objects representing a specific device or device type in the current system. For example, an application or other code running in Mac OS X can initiate a search for all USB storage devices. In device matching, the I/O Kit compares the keys and values in the matching dictionary an application supplies with a device nub's properties stored in the I/O Registry.
- A **device file** is a special file the I/O Kit creates in the `/dev` folder for each serial and storage device it discovers. If your application needs to access such a device, you use I/O Kit functions to get the path to the device and use the POSIX API to communicate with it.

The I/O Kit framework (stored on disk as `IOKit.framework` in `/System/Library/Frameworks`) contains a wide range of APIs that allow your application to work with devices. In addition to many folders that contain device-interface libraries and other user-space APIs for I/O Kit families, such as FireWire and Storage, the I/O Kit framework contains several files that define general I/O Kit APIs. These files, such as `IOCFPlugIn.h` (introduced in "Inside the Device-Interface Mechanism" (page 19)) and `IOKitLib.h` (covered in "The IOKitLib API" (page 41)), provide the foundation for kernel–user space communication. Figure 2-1 (page 15) shows the location of the I/O Kit APIs as they appear on the desktop.

Figure 2-1 The I/O Kit framework



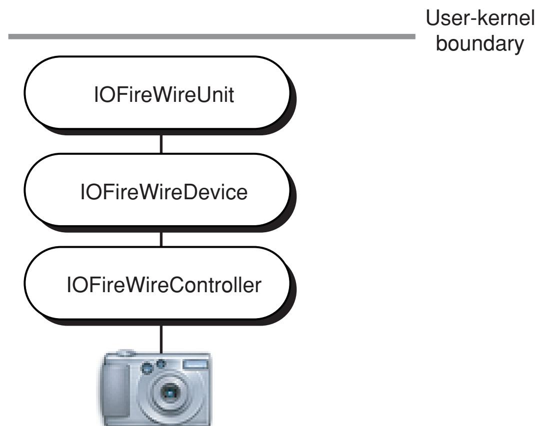
To use any of the I/O Kit APIs, your application must link with `IOKit.framework`.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

I/O Kit Driver-Stack Building

At boot time and whenever a system's hardware configuration changes, the I/O Kit discovers new devices and instantiates and loads nub and driver objects to support them. The nubs and drivers form a stack of objects that model the dynamic client-provider relationships among the various I/O Kit entities representing the hardware and software components in an I/O connection. As an example, consider the objects the I/O Kit instantiates when it discovers a FireWire device (shown in Figure 2-2 (page 16)).

Figure 2-2 I/O Kit objects supporting a FireWire device



In this example, the IOFireWire family and the I/O Kit take the following steps:

1. The I/O Kit instantiates an IOFireWireController object for each FireWire hardware interface, such as FireWire OHCI (Open Host Controller Interface), on a Mac OS X system.

2. The IOFireWire family queries each device on the bus and publishes an IOFireWireDevice object for each device that responds with its bus information.

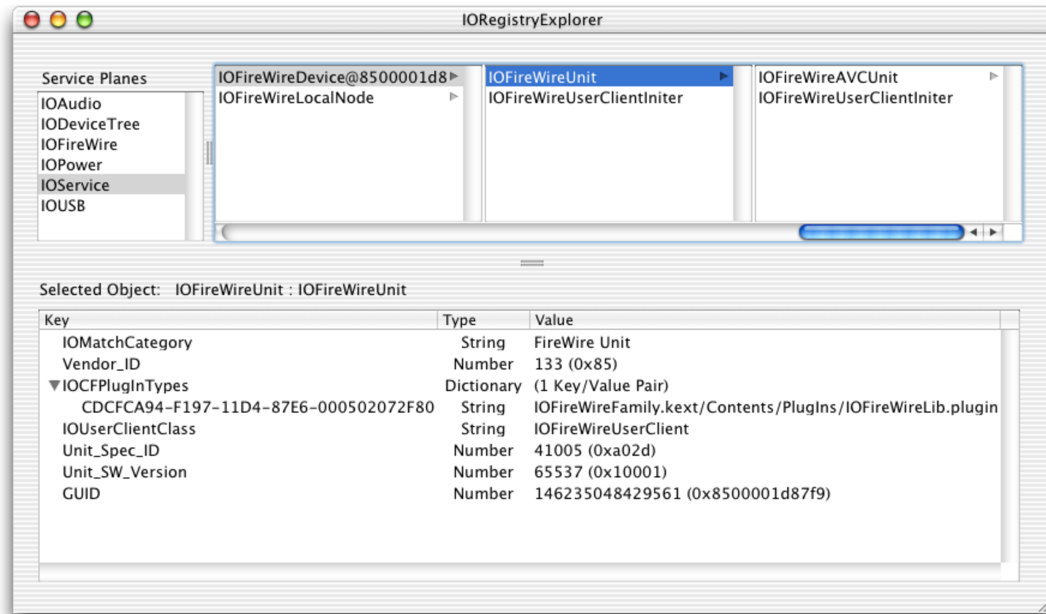
3. In its turn, the IOFireWireDevice object queries the device and publishes an IOFireWireUnit object for each unit directory it finds on the device.

Although this is not shown in Figure 2-2 (page 16), the I/O Kit matches drivers to particular unit types, such as SBP-2 or AV/C, which then publish nubs representing logical units.

You can also think of the driver stack as a branch in the I/O Registry tree. You can view the entities in any branch (driver stack) with I/O Registry Explorer. For example, Figure 2-3 (page 17) shows the I/O Registry Explorer view of part of the driver stack shown in Figure 2-2 (page 16), beginning with the IOFireWireDevice object. Note that Figure 2-3 (page 17) also shows the IOFireWireAVCUnit object representing the logical AV/C unit.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Figure 2-3 I/O Kit objects supporting a FireWire device in I/O Registry Explorer



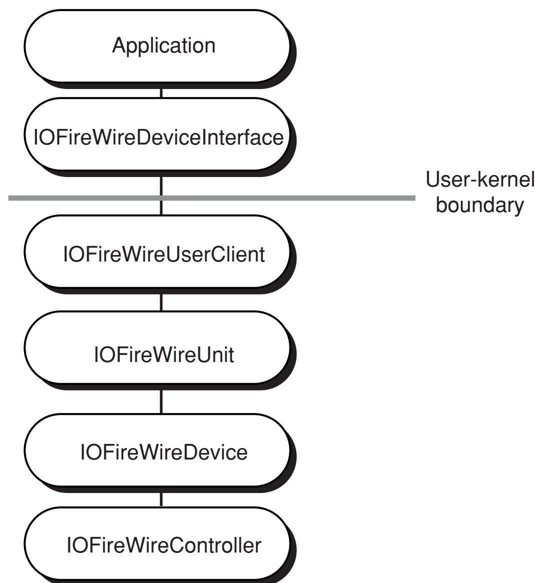
Notice the driver personality information (or, for a nub, the property table information) that the I/O Registry Explorer displays for the currently selected element. This information can help you choose key-value pairs to use for driver matching (for more information on the driver-matching process, see "Finding and Accessing Devices" (page 25)). For example, you might choose to look up devices with a specific vendor ID or GUID (globally unique ID), both of which are properties of a FireWire unit, as shown in Figure 2-3 (page 17).

When you use a device interface to communicate with a device, a user client object joins the driver stack. A family that provides a device interface also provides the user client object that transmits an application's commands from the device interface to the device. When your application requests a device interface for a particular device, the device's family instantiates the appropriate user client object, typically attaching it in the I/O Registry as a client of the device nub.

Revisiting the example of the FireWire device shown in Figure 2-2 (page 16), the acquisition of the IOFireWireDeviceInterface that the IOFireWire family provides results in the stack shown in Figure 2-4 (page 18).

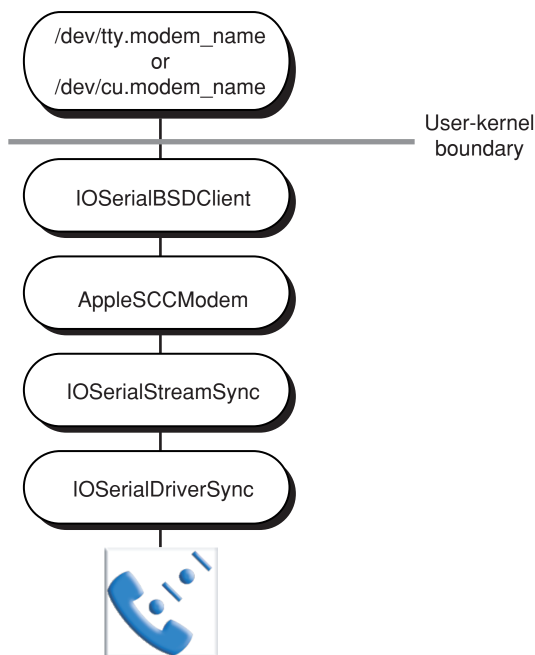
2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Figure 2-4 Adding a device interface to the FireWire driver stack



When the I/O Kit discovers a serial or storage device, it builds a stack of drivers and nubs similar to those shown in Figure 2-2 (page 16) for the FireWire device. In addition, however, the I/O Kit automatically creates a device file in the `/dev` directory, even if there is no current request for user-space access to the device. Figure 2-5 (page 18) shows the driver stack for a serial device.

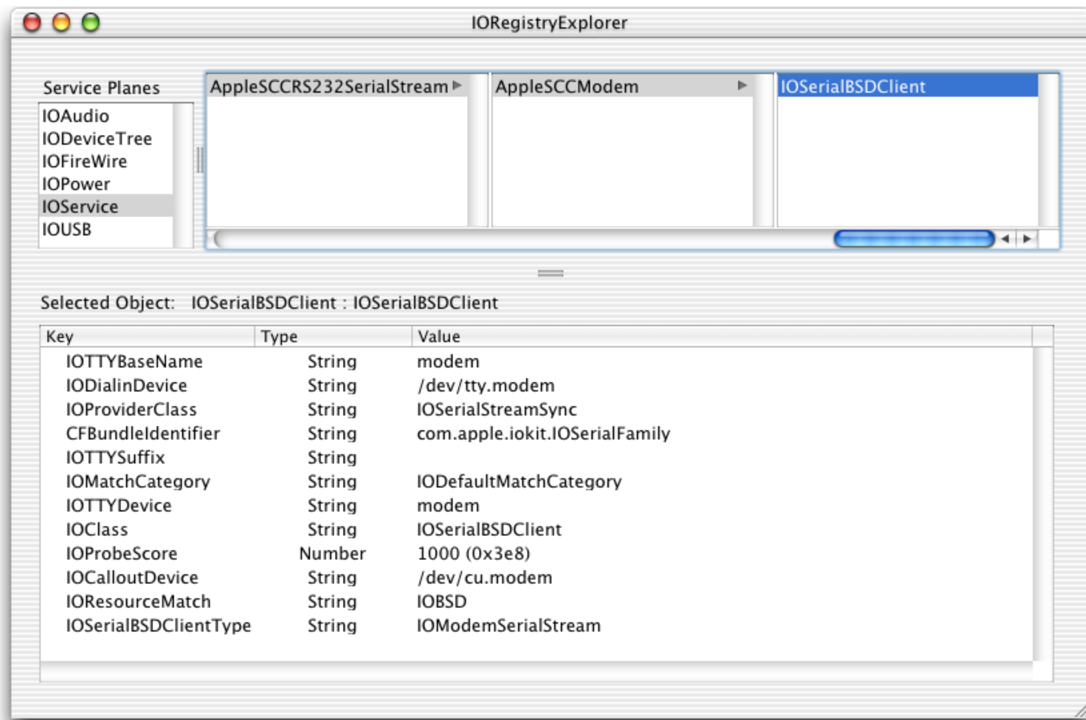
Figure 2-5 I/O Kit objects supporting a serial device



As with the FireWire device objects shown in Figure 2-3 (page 17), you can use the I/O Registry Explorer application to view the objects supporting a serial device, as Figure 2-6 (page 19) shows.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Figure 2-6 I/O Kit objects supporting a serial device in I/O Registry Explorer



Device Interfaces and Device Files

This section takes a closer look at the two main gateways to user-space device access, device interfaces and device files. The information in this section provides a conceptual overview of these mechanisms and does not constitute a step-by-step guide for using them. If you're more interested in finding out how to use device interfaces or device files in your application, you can skip ahead to "Finding and Accessing Devices" (page 25).

This section concludes with a brief description of how the I/O Kit uses Mach ports to allow user-space processes to communicate with the kernel.

Inside the Device-Interface Mechanism

Accessing hardware in Mac OS X with a device interface is similar in concept to using the Device Manager in previous versions of the Mac OS. Instead of calling Device Manager functions such as `OpenDriver`, `CloseDriver`, and `Control`, however, you call functions from the I/O Kit to obtain a device interface, then call functions the device interface defines, such as `open`, `close`, and `getInquiryData`.

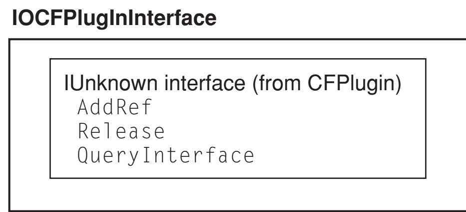
A **device interface** is a plug-in interface that conforms to the Core Foundation plug-in model (described in Core Foundation developer documentation, available in the Core Foundation Reference Library). As such, it is compatible with the basics of Microsoft's COM (Component Object Model) architecture. In practice, the only elements a device interface shares with COM are the layout of the interface itself (which conforms to the COM guidelines) and its inheritance from the COM-compatible `IUnknown` interface.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Following the Core Foundation plug-in model, an I/O Kit family that provides a device interface first defines a type that represents the collection of interfaces it supports. Then, it defines each interface, declaring the functions the interface implements. The type and the interfaces each receive a UUID (universal unique identifier, or ID), a 128-bit value that uniquely and permanently identifies it. The family makes these UUIDs available as a predefined name your application uses when it requests a device interface. For example, the IOFireWire family defines its version 5 FireWire device interface UUID as `kIOFireWireDeviceInterfaceID_v5`, which is much easier to use than the UUID string itself, `127A12F6-C69F-11D6-9D11-0003938BEB0A`.

Before an application can get a specific, family-defined device interface, it first gets an instance of an intermediate plug-in of type IOCFPlugInInterface. Defined in `IOCFPlugIn.h` (in the I/O Kit framework), the IOCFPlugInInterface structure begins with the `IUNKNOWN_C_GUTS` macro, defined in `CFPlugInCOM.h` (in the Core Foundation framework). This macro expands into the structure definition of the COM `IUnknown` interface. This means that an instance of the IOCFPlugInInterface begins with the `IUnknown` interface functions, required for all interfaces based on the Core Foundation plug-in model, shown in Figure 2-7 (page 20).

Figure 2-7 The IOCFPlugInInterface functions



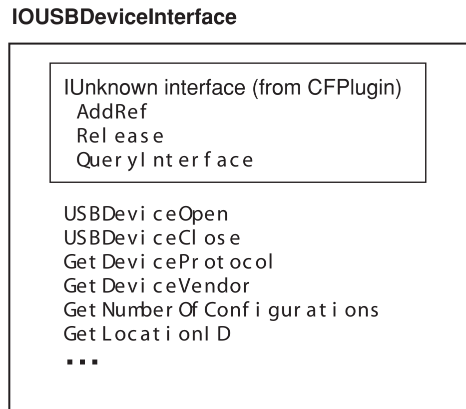
The `AddRef` and `Release` functions operate on the reference counts of the IOCFPlugInInterface object and the `QueryInterface` function creates new instances of the passed-in interface type, such as a device interface an I/O Kit family defines.

The `IOCFPlugIn.h` file also defines the function `IOCreatePlugInInterfaceForService`, which creates a new IOCFPlugInInterface object, and `IODestroyPlugInInterface`, which destroys the specified IOCFPlugInInterface object.

After your application gets the IOCFPlugInInterface object, it then calls its `QueryInterface` function, supplying it with (among other arguments) the family-defined UUID name of the particular device interface the application needs. The `QueryInterface` function returns an instance of the requested device interface and the application then has access to all the functions the device interface provides. For example, the USB family's device interface for USB devices provides the functions shown in Figure 2-8 (page 21).

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Figure 2-8 Some of the IOUSBDeviceInterface functions



As shown in Figure 2-8 (page 21), the family-defined device interface also begins with the `AddRef`, `Release`, and `QueryInterface` functions. This is because the family's device interface, like the IOCFPlugInInterface, fulfills the COM requirement that all interfaces must inherit from the `IUnknown` interface. Your application will probably never need to use these three functions from within the device interface, however, because most family-defined device interfaces provide their own accessor and reference-counting functions.

An application that has acquired a device interface for a device can act as a user-space driver for that device. For example, an application can drive a scanner that complies with the SCSI Architecture Model specifications because Apple does not supply an in-kernel driver for such a device. See the documentation for the family of the device you want to access for more information.

Inside the Device-File Mechanism

Darwin, the Mac OS X kernel, implements a version of 4.4BSD, a UNIX-based operating system that serves as the basis for the file systems and networking facilities of Mac OS X. In addition, Darwin's implementation of BSD includes much of the POSIX API. Darwin exports programmatic interfaces consistent with the POSIX API to application space that allow applications to communicate with serial, storage, and network devices through device files.

In a UNIX file system, an I/O **device file** is a special file that represents a block or character device such as a terminal, disk drive, printer, scanner, or tape drive. In essence, the device file acts as a buffer or stream of data for the device. Historically, device files reside in the `/dev` directory and have standard names, such as `mt0` for the first magnetic tape device, `tty0` for the first terminal, and so on. Because a UNIX system treats a device file like any other disk file, you can use UNIX commands with device files to perform input and output operations on devices. When you send data to a device file, the kernel intercepts it and redirects it to the device. Similarly, when a process reads from a device file, the kernel gets the data from the device and passes it to the application.

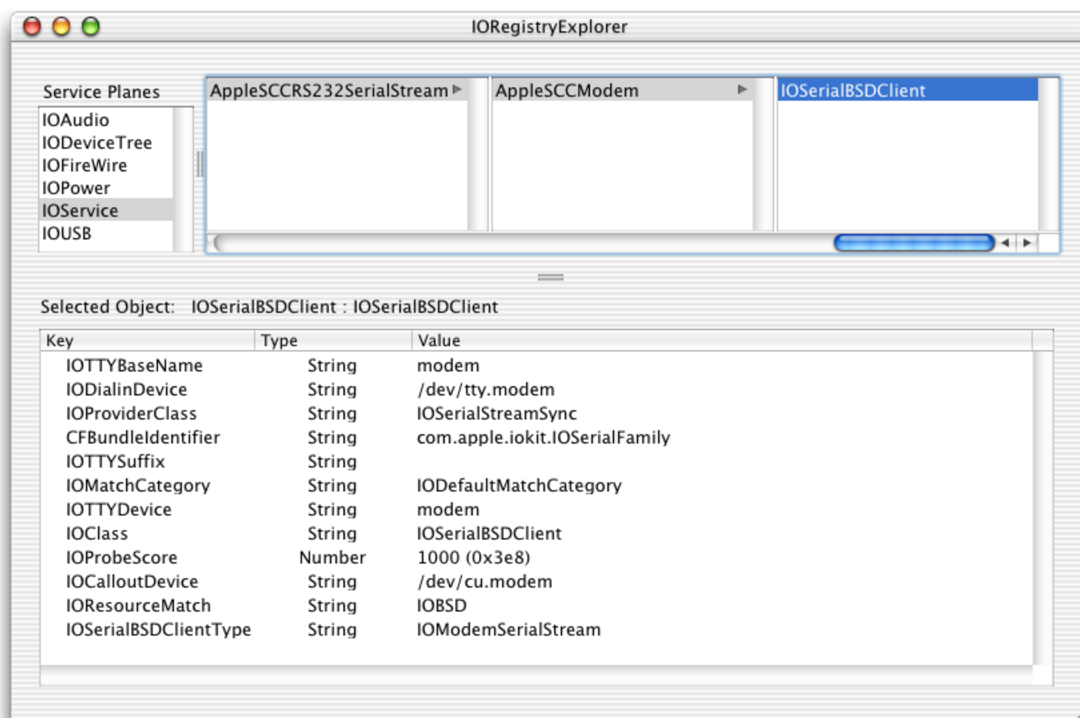
As it does with other devices, when the I/O Kit discovers a serial or storage device, it builds up a driver stack to support it. In addition, it instantiates a BSD user-client object that creates a device file node in the `/dev` directory. The BSD user-client object acts as a conduit between a client accessing a device through a device file and the in-kernel objects representing the device. For a serial device, the I/O Kit instantiates an IOSerialBSDClient object and for a storage device, the I/O Kit instantiates an IOMediaBSDClient object.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

When an application needs to access a serial or storage device for which it does not already have the device-file path, it can use I/O Kit functions to find a matching device in the I/O Registry. The application can then examine the object's properties to get its device-file path name. The properties you use to create the pathname string vary somewhat depending on whether you're accessing a serial or storage device; for more details, see "Finding and Accessing Devices" (page 25).

In Figure 2-9 (page 22), the I/O Registry Explorer application displays the properties of the IOSerialBSDClient for a modem.

Figure 2-9 An IOSerialBSDClient object in I/O Registry Explorer



Because of the dynamic and parallel nature of a Mac OS X system, a device may receive a different device-file name every time the I/O Kit discovers it. For this reason, you must search the I/O Registry for the device you're interested in to get the current device-file path before you attempt to access the device instead of hard-coding a device-file name, such as `/dev/cu.modem` or `/dev/rdisk0`, in your application. When you have the correct path to the device (including the device-file name), you then use either POSIX functions, such as `open` and `close`, to communicate with a storage device or the POSIX `termios` API for traditional UNIX serial port access to a serial device. For serial devices, data is also routed through PPP via the device file.

For information on the POSIX standard, see http://standards.ieee.org. For a POSIX programming reference, see *POSIX Programmer's Guide: Writing Portable Unix Programs with the POSIX* by Donald A. Lewine.

If your application needs to access a network connection, it does so using a particular IP address and standard socket functions. The I/O Kit builds up stacks of objects to support networking devices that are similar to the driver stacks for other devices. Because your application is concerned with accessing a connection instead of a device object, however, finding these objects in the I/O Registry is unnecessary.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Mac OS X provides networking APIs in Carbon and Cocoa that should handle most standard networking requirements for applications. You can also use the BSD sockets API to obtain network services. A recommended network programming book is *Unix Network Programming, Volume 1, Second Edition*, by W. Richard Stevens, Prentice-Hall PTR, 1998.

Communicating With the I/O Kit

Whether your application uses device interfaces or device files to access devices, it must communicate with the I/O Kit. The I/O Kit API (in the I/O Kit framework) includes functions to search the I/O Registry for specific devices or groups of devices, to access I/O Kit objects, and to access device and driver personality properties published in the I/O Registry. When an application uses these functions, it communicates with the I/O Kit through a Mach port, a unidirectional communication channel between a client that requests a service and a server (in this case, the I/O Kit) that provides the service.

Darwin's fundamental services and primitives are based on an Apple-enhanced version of Mach 3.0. Apple's improvements allow Mach to provide object-based APIs with communication channels (such as ports), a complete set of IPC (interprocess communication) primitives, and improved portability, among other features. When an application or other user-space process communicates with the I/O Kit, it does so by using RPC (remote procedure calls) on the I/O Kit's master port.

Specifically, your application requests the I/O Kit master port, using the I/O Kit function `IOMasterPort`, before it attempts to access any in-kernel objects, such as I/O Registry objects that represent devices. (Alternatively, it can use the convenience constant `kIOMasterPortDefault` to access the I/O Kit master port—for more information on how to do this, see "Getting the I/O Kit Master Port" (page 26).)

The `IOMasterPort` function calls the `host_get_io_master` function of `mach_host.c`, which returns a send right to the application. This means that the application has the right to use the port to access an object on the other side of the port, in this case, an I/O Kit object.

Don't worry if you're not knowledgeable about Mach. Except for the acquisition of the I/O Kit master port and some reference-counting issues (discussed in "Object Reference-Counting and Introspection" (page 41)), Darwin's Mach-based infrastructure is seldom exposed when using the I/O Kit to access hardware. If you're interested in learning more about how Darwin uses Mach, however, see *Kernel Programming Guide* for a good introduction.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Finding and Accessing Devices

This chapter assumes you have an application or other user-space code that needs to access one or more specific types of device and that you:

- Have determined that your application can't satisfy its hardware needs with the higher-level APIs described in "Hardware-Access Options" (page 9)
- Are familiar with the I/O Kit summary information in "Device Access and the I/O Kit" (page 13)
- Want to know how to access hardware with an I/O Kit device interface or using the POSIX API

This chapter provides a generic blueprint for accessing devices with I/O Kit device interfaces and device files. It describes several ways to find devices in the I/O Registry and how to examine each device you find. It then describes how to access a device through a device interface or a device file.

Your code will implement many of the functions discussed in this chapter because some actions, such as getting a Mach port to communicate with the I/O Kit and looking up your devices in the I/O Registry, are common to all applications that use the I/O Kit API to access hardware.

Important: Bear in mind that individual I/O Kit families are free to define access to their devices as they choose. Although your application will follow the steps this chapter outlines, some of the particulars, such as determining matching keys and values, vary by family. Be sure to consult the documentation for the device family you'll be working with.

This chapter describes how to use several I/O Kit and Core Foundation functions. You can view the header file describing the I/O Kit functions in `/System/Library/Frameworks/IOKit.framework/Headers/IOKitLib.h` and the Core Foundation functions in various files in `/System/Library/Frameworks/CoreFoundation.framework/Headers`. You can also view header documentation for these files on disk in `/Developer/Documentation` or on the web in the Device Drivers Reference Library. Complete sample projects are available in Sample Code > Hardware & Drivers and, when you install the Developer package, in `/Developer/Examples/IOKit`.

Finding Devices in the I/O Registry

Before you can communicate with your device through a device interface or using POSIX functions, you must first find it. If your device is plugged in, it's represented in the I/O Registry, the dynamically updated database of all I/O Kit objects that make up a running Mac OS X system.

The I/O Kit provides functions you can use to find devices matching your criteria in the I/O Registry. This section describes the device-matching process in detail, showing how to use I/O Kit functions to create matching dictionaries and look up matching devices in the I/O Registry.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Important: Be aware that device matching is a family-specific task: Some I/O Kit families define special matching protocols or particular keys and values you must use with the I/O Kit functions this section describes. Family documentation (in header files in the I/O Kit framework) or device-access documents (available in the Device Drivers Reference Library) can help you determine how to implement device matching for your particular device.

Device Matching

Applications that use the I/O Kit to access hardware are typically looking for a particular device, such as an ATA, USB, FireWire, or other device. In the general case, if the desired device is plugged in to a Mac OS X system, there is an object that represents it attached in the I/O Registry. The process of finding this object is called **device matching**.

Recall that at boot time (and whenever devices are attached or removed), the I/O Kit:

- Instantiates a nub object that represents the device
- Attaches the nub to the I/O Registry
- Registers the nub

The device family publishes device properties in the nub object, which the I/O Kit uses to find a suitable driver for the device. You can view these properties with the I/O Registry Explorer application (available at `/Developer/Applications`) or on the command line with `ioreg`.

To find devices in the I/O Registry, you perform the following steps:

1. Get the I/O Kit master port to communicate with the I/O Kit.
2. Find the appropriate keys and values that sufficiently define the target device or set of devices.
3. Use the key-value pairs to create a matching dictionary.
4. Use the matching dictionary to look up matching devices in the I/O Registry.

To perform these steps, you use a combination of I/O Kit and Core Foundation functions. The following sections describe each of these steps in detail.

Getting the I/O Kit Master Port

When your application uses functions that communicate directly with objects in the kernel, such as objects that represent devices, it does so through a Mach port, namely, the I/O Kit master port. Several I/O Kit functions require you to pass in an argument identifying the port you're using. Starting with Mac OS X version 10.2, you can fulfill this requirement in either of two ways:

- You can get the I/O Kit master port from the function `IOMasterPort` and pass that port to the I/O Kit functions that require a port argument.
- You can pass the constant `kIOMasterPortDefault` to all I/O Kit functions that require a port argument.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

In versions of Mac OS X prior to Mac OS X version 10.2, an application was required to use the first option and explicitly request the I/O Kit master port (you will encounter this procedure in older documentation and sample code). If you choose this method, you add code like the following to your application:

```
mach_port_t myMasterPort;
kern_return_t result;

result = IOMasterPort(MACH_PORT_NULL, &myMasterPort);
```

Then, in calls to I/O Kit functions that require a port argument, such as `IOServiceGetMatchingServices`, you pass in `myMasterPort`, as in this example:

```
IOServiceGetMatchingServices(myMasterPort, myMatchingDictionary,
                            &myIterator);
```

When you're completely finished with the port you received from `IOMasterPort`, you should release it, using `mach_port_deallocate`. Although multiple calls to `IOMasterPort` will not result in leaking ports (each call to `IOMasterPort` adds another send right to the port), it's good programming practice to deallocate the port when you're finished with it.

Starting with Mac OS X version 10.2, you can bypass this procedure entirely and use instead the convenience constant `kIOMasterPortDefault` (defined in `IOKitLib.h` in the I/O Kit framework). This means that when you call a function that requires the I/O Kit master port, such as `IOServiceGetMatchingServices`, you can pass in `kIOMasterPortDefault` instead of the `mach_port_t` object you get from `IOMasterPort`, as in this example:

```
IOServiceGetMatchingServices(kIOMasterPortDefault, myMatchingDictionary,
                            &myIterator);
```

Getting Keys and Values for a Device-Matching Dictionary

An application begins device matching by creating a matching dictionary that defines which device (or set of devices) the application needs to access. A matching dictionary is a Core Foundation `CFMutableDictionaryRef` object, containing a set of key-value pairs that describe particular device properties.

You have a few options for obtaining keys and values for device matching:

- You can obtain matching keys from header files in `Kernel.framework` or `IOKit.framework` and you can define constants for property values.
- You can examine driver personality information that is stored in on-disk drivers.
- You can examine objects in the I/O Registry to obtain property information for the devices they represent.

Each of these options is described in the following sections.

Personality Property Keys and Values

The file `IOKitKeys.h` (located in the I/O Kit framework) defines many general matching keys, some of which are shown in Listing 3-1 (page 28).

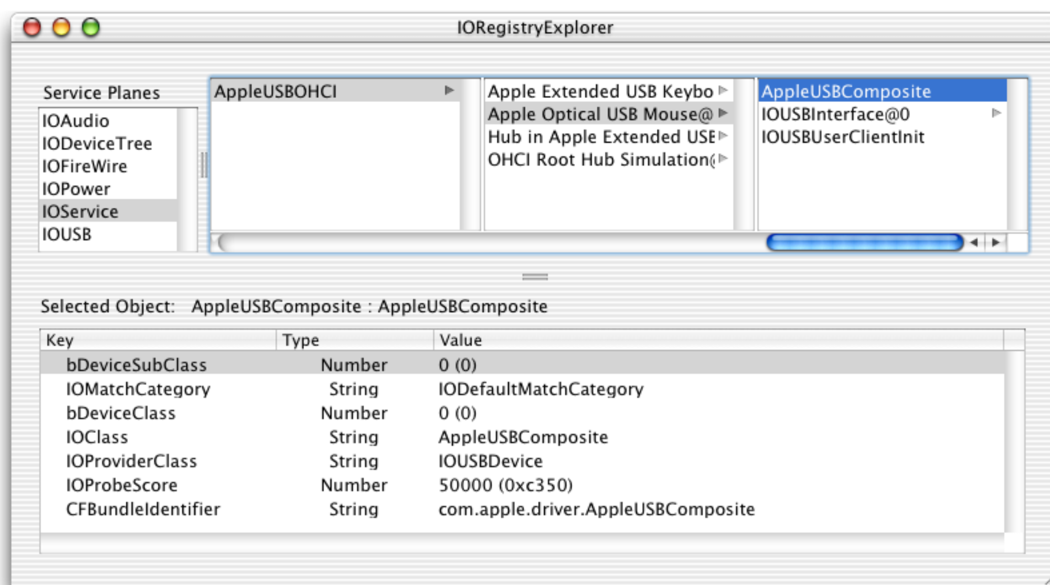
2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Listing 3-1 Matching keys from IOKitKeys.h

```
// Keys for matching IOService properties by name
#define kIOProviderClassKey      "IOProviderClass"
#define kIONameMatchKey          "IONameMatch"
#define kIOPropertyMatchKey      "IOPropertyMatch"
#define kIOPathMatchKey          "IOPathMatch"
#define kIOLocationMatchKey      "IOLocationMatch"
#define kIOResourceMatchKey      "IOResourceMatch"
```

Figure 3-1 (page 28) shows the AppleUSBComposite driver personality as it appears in I/O Registry Explorer (the AppleUSBComposite driver matches on composite-class USB devices for which there are no vendor-specific drivers). Notice that several of the keys in the AppleUSBComposite personality are defined in `IOKitKeys.h`.

Figure 3-1 The I/O Registry Explorer application, showing various keys and values for the AppleUSBComposite driver



One key of particular importance is the `IOProviderClass` key, which is included in all driver personalities. This key specifies the name of the nub class or nub superclass the driver attaches to. Although all device-matching dictionaries contain this key, few contain only this key because the resulting set of matching devices would be very large.

In most cases, you need to add more key-value pairs to make your matching dictionary more specific. By convention, keys for specific device properties are defined in device header files, such as `IOSerialKeys.h`, which defines property keys for serial devices, and `IOHIDKeys.h`, which defines property keys for HID (Human Interface Device) class devices. You can find these header files among the headers in the I/O Kit or Kernel frameworks. A partial listing of `IOHIDKeys.h` is shown in Listing 3-2 (page 28).

Listing 3-2 HID class device matching keys from IOHIDKeys.h

```
#define kIOHIDDeviceKey                  "IOHIDDevice"
#define kIOHIDTransportKey               "Transport"
#define kIOHIDVendorIDKey                "VendorID"
#define kIOHIDProductIDKey               "ProductID"
```

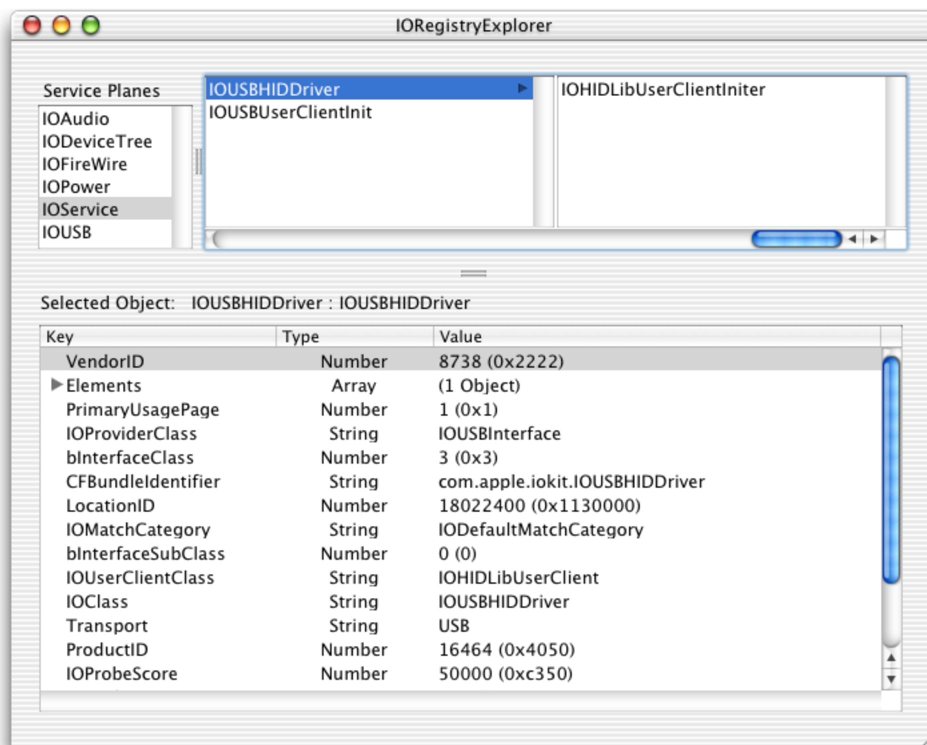

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

```
#define kIOHIDVersionNumberKey           "VersionNumber"
#define kIOHIDManufacturerKey            "Manufacturer"
#define kIOHIDProductKey                 "Product"
#define kIOHIDSerialNumberKey            "SerialNumber"
#define kIOHIDLocationIDKey              "LocationID"
#define kIOHIDPrimaryUsageKey            "PrimaryUsage"
#define kIOHIDPrimaryUsagePageKey        "PrimaryUsagePage"
```

Your application must supply a value for each device-property key it uses in a matching dictionary. You may have access to header files that define values for these keys—for example, you may be working on an application to access a device driver supplied by your own company. If you don't have predefined header files, you can define your own constants. For example, you can define property value constants for a device whose driver properties you have found with the I/O Registry Explorer application, as described in "Examining the I/O Registry" (page 30).

As a further example, suppose you have examined the keys and values for the driver of type IOUSBHIDDriver, in this case a driver for a joystick, shown in Figure 3-2 (page 29).

Figure 3-2 Some keys and values for a HID class device, shown in I/O Registry Explorer



To find the device this driver supports, you set up a matching dictionary to look up all HID class devices, using the `kIOHIDDeviceKey` (shown in Listing 3-2 (page 28)). Then, you can narrow the search to HID class devices on the USB bus by setting the transport key (identified by the constant `kIOHIDTransportKey` from `IOHIDKeys.h`) to the value "USB" as defined by the following constant in your code:

```
#define kMyDeviceTransportValue "USB"
```


2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

For more detail on how to modify a matching dictionary, see "Setting Up a Matching Dictionary to Find Devices" (page 31).

Examining Driver Personality Information

Another way to get the keys and values you need to describe a device is to examine the driver personality information in a device's on-disk driver.

Before a device driver is loaded, it is stored on disk or in ROM as a kernel extension, or KEXT. KEXTs are usually located in `/System/Library/Extensions`. A KEXT file on disk is a bundle that contains other files and folders. Each driver KEXT includes an information property list, in XML format, typically in a file named `Info.plist`. The property list includes one or more driver personalities—dictionaries whose properties specify which devices the driver is suitable to manage.

A KEXT usually stores its `Info.plist` file in its `Contents` directory. You can examine its property list with the Property List Editor application (which is located in `/Developer/Applications`), by displaying it in the Terminal application (using `more` or another text-display command), or by opening the property list with an application such as Xcode. Listing 3-3 (page 30) shows a partial dictionary listing for the AppleFWAudio driver. It contains keys such as the class key and the provider class key, as well as values for these keys (the strings "AppleFWAudioDevice" and "IOFireWireAVCUnit," respectively).

Listing 3-3 A partial listing of the personality dictionary for the AppleFWAudio driver

```
<key>IOKitPersonalities</key>
<dict>
    <key>AppleFWAudioDevice (AVC)</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.driver.AppleFWAudio</string>
        <key>IOClass</key>
        <string>AppleFWAudioDevice</string>
        <key>IOProviderClass</key>
        <string>IOFireWireAVCUnit</string>
        ...
    </dict> ...
```

By examining a driver's personality dictionary, your application can determine the keys and values to put in a matching dictionary to specify that driver's device.

Examining the I/O Registry

You can examine the I/O Registry to obtain driver personality keys and values to use in a matching dictionary. When the I/O Kit loads a driver for a device, it stores a copy of the matching personality information for that device in the I/O Registry. The developer version of Mac OS X includes the I/O Registry Explorer application, which displays the I/O Registry for your currently running system. You can see examples of I/O Registry Explorer's display in Figure 3-1 (page 28) and Figure 3-2 (page 29).

The developer version of Mac OS X also includes `ioreg`, a tool you can use to examine the I/O Registry. You run the tool in a Terminal window to display I/O Registry information for your current system. Using the application or the tool, you can examine the personality information of loaded drivers and the property information in device nubs.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

If you find driver personality keys and values you want to use in your application, you may be able to obtain constants for them in Apple or third-party headers, or you may need to define your own constants. For more information, see "Personality Property Keys and Values" (page 27).

Setting Up a Matching Dictionary to Find Devices

When you've determined the keys and values that define your device, you can use that information to set up a matching dictionary to find the device in the I/O Registry. The I/O Kit provides functions that create a CFMutableDictionary object containing a specific key and the value you pass to it. Bear in mind, however, that you are not limited by the dictionaries these functions create. If you require a different or more focused search, you can modify these dictionaries or even create your own, as described later in this section.

You can use any of the following I/O Kit functions to create a matching dictionary:

- `IOServiceMatching` creates a dictionary you can use to match on an object's class or superclass. The dictionary that `IOServiceMatching` creates consists of the `IOProviderClass` key and the value you pass to `IOServiceMatching`.

  You can initiate a very broad search of the I/O Registry by using `IOServiceMatching` to create a dictionary that matches on a specific object class (and its subclasses). For example, you can create a dictionary to match on all objects of class IOUSBDevice by using code like the following:

  ```
  CFMutableDictionaryRef myUSBMatchDictionary = NULL;
  myUSBMatchDictionary = IOServiceMatching(kIOUSBDeviceName);
  ```

- `IOServiceNameMatching` creates a dictionary that consists of the key `IONameMatch` and the value you pass to the function.

  You might use `IOServiceNameMatching` to create a dictionary that matches on a device's name, rather that its class, perhaps if your company's device has a unique name. For example:

  ```
  CFMutableDictionaryRef myCompanyDeviceMatchDictionary = NULL;
  myCompanyDeviceMatchDictionary = IOServiceNameMatching("MyCompany");
  ```

- `IOBSDNameMatching` creates a dictionary you can use to find devices that have device-file representations in the BSD layer, such as storage devices. The dictionary the `IOBSDNameMatching` function creates consists of the `kIOBSDNameMatching` key (defined in `IOKitServer.h`) and the name of the device file you pass to the function.

  If you've determined the device-file name of your device (by, for example, looking in the `/dev` directory using the Terminal application), you can create a dictionary to match on it using the `IOBSDNameMatching` function, using code like the following:

  ```
  CFMutableDictionaryRef myBSDMatchDictionary = NULL;
  myBSDMatchDictionary = IOBSDNameMatching(kIOMasterPortDefault, 0,
                                  "disk1s8");
  ```

  Note that `IOBSDNameMatching` expects only the device-file name in the last parameter, not the complete path.

  It's possible for a storage device to receive a different device-file name each time the I/O Kit discovers it, however, so your code should not rely on a device-file name remaining unchanged.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

You use the dictionaries these I/O Kit functions create to pass to one of the I/O Kit look-up functions (described in "Looking Up Devices in the I/O Registry" (page 34)). The I/O Kit look-up functions each consume one reference to the dictionary. If you use the dictionary in some other way, however, you must use `CFRelease` to release it when you're finished.

The I/O Kit dictionary-creation functions each create a dictionary with a single key-value pair. Unless you're looking for a device that is sufficiently described by such a key-value pair, you probably want to either modify one of these dictionaries or create your own.

Important: Because `CFMutableDictionary` is a standard Core Foundation data type, you can use Core Foundation functions to modify a matching dictionary or to create your own from scratch. For more information on Core Foundation collection objects, such as dictionaries, see the Core Foundation Reference Library.

Many I/O Kit families define matching rules that help you narrow down your search. The USB family, for example, bases its matching rules on the USB Common Class specification. Before you modify an existing dictionary or create one of your own, you should become thoroughly familiar with your device family's matching rules. This is especially important when you use your matching dictionary to search the I/O Registry. The I/O Kit look-up functions apply family-defined matching rules so it is essential for your matching dictionary to contain the correct combination of properties.

To modify an existing dictionary, you get a generic dictionary, such as one created by `IOServiceMatching`, and use the Core Foundation function `CFDictionaryAddValue` to add other key-value pairs. For example, to find a FireWire unit, you can call `IOServiceMatching` to create a dictionary containing the key `IOProviderClass` and the value `IOFireWireUnit`. Then, a call to `CFDictionaryAddValue` modifies the dictionary by adding the IOFireWire family–defined key `Unit_SW_Version` and the application-defined value `myFireWireUnitSWVersionID`. Listing 3-4 (page 32) shows how to do this.

Listing 3-4 Modifying a matching dictionary

```
CFMutableDictionaryRef  matchingDictionary =
                            IOServiceMatching("IOFireWireUnit" );
UInt32 value;
CFNumberRef cfValue;
value = myFireWireUnitSwVersionID;
cfValue = CFNumberCreate( kCFAllocatorDefault, kCFNumberSInt32Type, &value );
CFDictionaryAddValue( matchingDictionary, CFSTR( "Unit_SW_Version" ),
                        cfValue);
CFRelease( cfValue );
```

Occasionally, you may find that you must perform a look-up on a handful of properties that don't follow any device family's matching rules. Perhaps there are no family-defined matching rules for the device you're interested in or you're creating a tool to test device matching. Although you should always follow your device family's matching rules (when they exist), you can use the special `IOPropertyMatch` key (defined in `IOKitKeys.h` in the I/O Kit framework) to create a dictionary containing a set of arbitrary matching properties.

To do this, you create two dictionaries: one that contains the set of key-value pairs that describe your device and one that contains the `IOPropertyMatch` key and your first dictionary as its value. You use the Core Foundation function `CFDictionaryCreateMutable` to create both dictionaries. The following code fragment shows how to create the subdictionary (the dictionary containing your key-value pairs that you'll place in your matching dictionary):

```
CFMutableDictionaryRef mySubDictionary;

mySubDictionary = CFDictionaryCreateMutable(kCFAllocatorDefault, 0,
```


2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

```
                    &kCFTypeDictionaryKeyCallBacks,
                    &kCFTypeDictionaryValueCallBacks);
CFDictionarySetValue(mySubDictionary, CFSTR(kMyDevicePropertyKey),
                    CFSTR(kMyDevicePropertyValue));
```

Now you use `CFDictionaryCreateMutable` again, this time to create your matching dictionary. The matching dictionary should contain only the key `IOPropertyMatch` and your subdictionary as its value. The `IOPropertyMatch` key signals the I/O Kit to match on the set of properties in the subdictionary without regard to any family-defined matching rules. The following code fragment shows how to create your matching dictionary:

```
CFMutableDictionaryRef myMatchingDictionary;

myMatchingDictionary = CFDictionaryCreateMutable(kCFAllocatorDefault, 0,
                    &kCFTypeDictionaryKeyCallBacks,
                    &kCFTypeDictionaryValueCallBacks);
CFDictionarySetValue(myMatchingDictionary, CFSTR(kIOPropertyMatchKey),
                    mySubDictionary);
```

The callback constants `kCFTypeDictionaryKeyCallBacks` and `kCFTypeDictionaryValueCallBacks` define Core Foundation callback structures you can use when the keys and values in your dictionary are all CFType-derived objects. You can find documentation about these structures in the Core Foundation Reference Library.

Setting Up a Matching Dictionary to Find Device Files

You set up a matching dictionary to find serial or storage devices using the same I/O Kit dictionary-creation functions you use for other types of matching dictionaries. As with other devices, the provider-class name you use to create a matching dictionary depends on the type of device you're looking for:

- All serial devices can be identified by the provider class IOSerialBSDClient. The `IOProviderClass` key you send to `IOServiceMatching` to create a matching dictionary is `kIOSerialBSDServiceValue`, defined in `IOSerialKeys.h` in the I/O Kit framework.

  Because each serial device object in the I/O Registry has a property with the key `kIOSerialBSDTypeKey`, you can refine your dictionary to match on specific types of serial devices. Currently, the possible values for this key (also defined in `IOSerialKeys.h`) are `kIOSerialBSDAllTypes`, `kIOSerialBSDModemType`, or `kIOSerialBSDRS232Type`.

- Storage device objects are members of the Storage family and their provider class is usually IOMedia, but can also depend on the device type. CD devices, for example, are subclasses of IOCDMedia. Check the Storage family header files (available in the I/O Kit framework) for the appropriate provider-class key to pass to `IOServiceMatching`.

  Also check the header files for additional keys and values you can use to refine your matching dictionary, such as the `kIOMediaEjectableKey`, which identifies whether the media is ejectable.

After you've created a matching dictionary, you pass it to one of the I/O Kit look-up functions. If devices matching your criteria exist in the I/O Registry, the look-up function returns an iterator you can use to access each matching object.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Looking Up Devices in the I/O Registry

Device look-up is at the heart of the device-matching process: It consists of searching the I/O Registry for device objects that match the criteria specified in a matching dictionary. The I/O Kit contains a few functions that perform device look-up and provide access to objects that match your matching dictionary:

- `IOServiceGetMatchingServices` follows the family-defined matching rules (if any) to find registered objects that match the passed-in matching dictionary. It supplies an IOIterator object that you can use to access the set of matching objects.
- `IOServiceGetMatchingService` is similar to `IOServiceGetMatchingServices` except that it returns only the first IOService object that matches the passed-in matching dictionary. Because it returns the first matching IOService object itself, `IOServiceGetMatchingService` does not give you an iterator that allows you to access other objects that may meet the matching criteria. You might choose to use this function if you're reasonably certain you can create a matching dictionary that will match only on your device and you want to bypass the code that iterates over a list of matching devices.

  `IOServiceGetMatchingService` also strictly follows the matching rules defined by the value of the passed-in `IOProviderClass` key.
- `IOServiceAddMatchingNotification` follows the family-defined matching rules (if any) to find objects matching the passed-in matching dictionary whose state changes in the way you identify. When a matching object's state change matches the state change you're interested in (such as being registered or terminated), `IOServiceAddMatchingNotification` notifies the caller and provides an iterator with which to access the set of matching objects.

  Your application can use `IOServiceAddMatchingNotification` to receive notification of when matching devices come or go. This is particularly useful for applications that need to access hot-pluggable devices, such as FireWire or USB devices. For an outline of how to receive notifications about matching devices, see "Getting Notifications of Device Arrival and Departure" (page 35).

The I/O Kit look-up functions each consume one reference to the matching dictionary you pass to them. If you need to use the dictionary again, be sure to increase its reference count by calling`CFRetain` on it before you send it to a look-up function.

If you use a look-up function that returns an iterator (`IOServiceGetMatchingServices` or `IOServiceAddMatchingNotification`), your application must release this iterator when it is finished with it. In the case of `IOServiceAddMatchingNotification`, make sure you release the iterator *only* if you're also ready to stop receiving notifications: When you release the iterator you receive from `IOServiceAddMatchingNotification`, you also disable the notification. If you use `IOServiceGetMatchingService`, your application is responsible for releasing the object reference it receives.

Applications typically use these look-up functions to find device nubs whose property information matches the criteria defined in the application's matching dictionary. Occasionally, however, an application looks for a driver in the I/O Registry, instead of a nub. Your application would look for a driver, for example, if you're writing both the application and the in-kernel driver and intend for them to work together. A key point is the distinction between drivers and nubs: Nubs are always registered in the I/O Registry (it's the act of registration that triggers driver matching), but drivers can be attached in the I/O Registry without being registered. This is important, because the I/O Kit look-up functions find only registered objects. To find an unregistered object, you start by finding its registered provider or client and use an I/O Registry traversal function to step to the object you're looking for.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

If, for example, your application needs to access an unregistered object, you must first identify the object's immediate parent or child, whichever is easiest to find. Let's say you can find the registered child of the object you want: You create a matching dictionary that describes the child object and pass it to `IOServiceGetMatchingServices`. You use `IOIteratorNext` (discussed in "Examining Matching Objects" (page 36)) to get access to the child object and then use the I/O Kit function `IORegistryEntryGetParentEntry` to get access to its unregistered parent object. For more information on the I/O Registry traversal functions the I/O Kit provides, see "The IOKitLib API" (page 41).

Getting Notifications of Device Arrival and Departure

If you're working with hot-pluggable devices, such as FireWire or USB devices, you can use I/O Kit and Core Foundation functions to set up a mechanism that notifies you when the devices you're interested in come or go. To set up such a mechanism, follow these steps:

1. Use the `IONotificationPortCreate` function to create a notification object that can listen for I/O Kit notifications, either on a run loop or a Mach port (these steps show only the recommended run-loop approach—for a brief description of the functions you use to set up a notification mechanism with a Mach port, see "The IOKitLib API" (page 41)):

```
IONotificationPortRef notificationObject;
mach_port_t masterPort; //This is the port you received from IOMasterPort.
        //Alternatively, you can pass kIOMasterPortDefault to the
        //IONotificationPortCreate function.

notificationObject = IONotificationPortCreate(masterPort);
```

2. Create a run-loop source for the notification object, using the `IONotificationPortGetRunLoopSource` function:

```
CFRunLoopSourceRef notificationRunLoopSource;

//Use the notification object received from IONotificationPortCreate
notificationRunLoopSource =
            IONotificationPortGetRunLoopSource(notificationObject);
```

3. Add the run-loop source to your run loop (usually, your application's current run loop), using the Core Foundation `CFRunLoopAddSource` function:

```
CFRunLoopAddSource(CFRunLoopGetCurrent(), notificationRunLoopSource,
                    kCFRunLoopDefaultMode);
```

4. Call the `IOServiceAddMatchingNotification` function, passing it the following arguments:

   - The notification object you received from `IONotificationPortCreate`
   - A constant defining the type of event you want notification of, such as device registration or termination (these constants are defined in `IOKitKeys.h` in the I/O Kit framework)
   - The Core Foundation matching dictionary you've created to define the types of devices you're interested in
   - The function you want called when a matching device's state changes in the way you've identified
   - An optional reference constant for your callback function's use
   - An `io_iterator_t` object to access the list of matching devices

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

5. Call IOIteratorNext (discussed in "Examining Matching Objects" (page 36)) to access the matching devices that are already present in the I/O Registry and to arm the notification mechanism so you will receive notifications of future matching devices as your run loop runs.

6. Call the Core Foundation CFRunLoopRun function to start the run loop and receive notifications when new matching devices arrive.

Examining Matching Objects

After you've successfully created a matching dictionary and passed it to one of the I/O Kit look-up functions, you receive an iterator you can use to access the list of matching objects. (Of course, if you used IOServiceGetMatchingService, you instead receive a reference to the first matching object itself, not an iterator.) The iterator is an object of class IOIterator and this section describes how you use it to examine a set of matching objects.

When you get an iterator from IOServiceGetMatchingServices, you pass it to the I/O Kit function IOIteratorNext, which returns a matching object from the list. With the matching object in hand, you can directly examine the properties it publishes in the I/O Registry. You might do this to determine if an object does, in fact, represent the device you want to access or if you want to display information about the matching device to a user.

The IOIteratorNext function returns an object of type io_object_t, which the caller should release when it is finished. Each call to IOIteratorNext returns the next object in the list of matching objects or zero if there are no more objects or if the iterator is no longer valid. If you receive zero when you think there may still be more matching objects in the list, you can call the IOIteratorIsValid function to make sure the iterator is still valid. In the unlikely case that the iterator is invalid, it's usually because the I/O Registry has changed in some way. If your iterator has become invalid, the best thing to do is to call the I/O Kit function IOIteratorReset and begin iterating again.

Now that you have an io_object_t object representing one of the matching objects in the list, you can use other I/O Kit functions to examine it more closely. For example, you can call IOObjectGetClass to see the class name of the passed-in object. To look at a specific property, you can pass the corresponding property key to IORegistryEntryCreateCFProperty, which returns a Core Foundation representation of that property's value. See "The IOKitLib API" (page 41) for more object-examination functions.

Putting It All Together: Accessing a Device

Previous sections described how to:

- Find keys and values that identify a device's properties
- Use the keys and values to create a matching dictionary
- Use the matching dictionary to look up matching devices in the I/O Registry

This section builds on that information to show how to get a device interface (or the pathname for a device file) and begin communicating with a device.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Let's assume you used the IOServiceGetMatchingServices look-up function to find matching devices. This means that you received an iterator handle you can pass to the IOIteratorNext function to access each device object in turn. Whether you iterate over the entire list, examining each device, or just grab the first device in the list, the next step is communication with the device—a step that differs depending on whether you access the device through a device interface or a device file.

Getting a Device Interface

As described in "Device Access and the I/O Kit" (page 13), many I/O Kit families supply device interfaces that provide user-space access to the devices they support. This section presents information that is appropriate for many types of device families but may vary from the specifics of any particular family.

As always, be sure to check the documentation for your device's family before implementing the steps outlined in this section.

When you've determined that the io_object_t object you received from IOIteratorNext represents a device you want to access, you first create an intermediate Core Foundation plug-in interface for it, using the IOCreatePlugInInterfaceForService function (defined in IOCFPlugIn.h in the I/O Kit framework).

Before you get the intermediate plug-in, however, you must know which device interface type you need to get. I/O Kit families that support device interfaces define both an interface type that represents the collection of interfaces they provide and each individual interface in the type. You can get this information from your device family's header files. For example, Listing 3-5 (page 37) shows the USB family's definition of the kIOUSBDeviceUserClientTypeID (its library of device interface types) and the kIOUSBDeviceInterfaceID (one of the individual device interfaces it provides).

Listing 3-5 Device interface definitions from IOUSBLib.h

```
#define kIOUSBDeviceUserClientTypeID CFUUIDGetConstantUUIDWithBytes(NULL,   \
    0x9d, 0xc7, 0xb7, 0x80, 0x9e, 0xc0, 0x11, 0xD4,          \
    0xa5, 0x4f, 0x00, 0x0a, 0x27, 0x05, 0x28, 0x61)
...
#define kIOUSBDeviceInterfaceID CFUUIDGetConstantUUIDWithBytes(NULL,    \
    0x5c, 0x81, 0x87, 0xd0, 0x9e, 0xf3, 0x11, 0xD4,          \
    0x8b, 0x45, 0x00, 0x0a, 0x27, 0x05, 0x28, 0x61)
```

To get an intermediate interface, you pass the following information to IOCreatePlugInInterfaceForService:

- The device reference (the io_object_t you received from IOIteratorNext)
- The family's library type ID (from the family's header files)
- The intermediate plug-in type (always kIOCFPlugInInterfaceID, defined in IOCFPlugIn.h)
- The address of an IOCFPlugInInterface object (to contain the intermediate plug-in)
- The address of an integer variable (currently unused)

Listing 3-6 (page 38) shows an application using IOCreatePlugInInterfaceForService to get an intermediate plug-in interface for the FireWire family's IOFireWireLib (one of the FireWire family's device interface libraries).

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Listing 3-6 Getting an intermediate IOCFPlugInInterface object

```
IOCFPlugInInterface** cfPlugInInterface = 0;
IOReturn result;
SInt32 theScore;

// aDevice is the io_object_t from IOIteratorNext.
result = IOCreatePlugInInterfaceForService( aDevice, kIOFireWireLibTypeID,
            kIOCFPlugInInterfaceID, &cfPlugInInterface, &theScore );
```

Now that you have an intermediate interface, you use it to get the specific type of device interface you need. Again, see your device family's header files for the definitions of specific device interface names. To get a particular device interface, you use the `QueryInterface` function of the IOCFPlugInInterface, passing it the following arguments:

- A reference to the IOCFPlugInInterface (received from the call to `IOCreatePlugInInterfaceForService`)
- The UUID of the device interface (you use the Core Foundation function `CFUUIDGetUUIDBytes` to get the actual UUID from the family's device interface constant)
- The address of the device interface (to contain the new device interface)

Listing 3-7 (page 38) shows an application using an intermediate IOCFPlugInInterface object to get the SCSI Architecture Model family's MMC (Multimedia Commands) device interface (defined in `SCSITaskLib.h` in the I/O Kit framework).

Listing 3-7 Getting a specific device interface object

```
HRESULT herr;
MMCDeviceInterface **mmcInterface = NULL;

// plugInInterface is the IOCFPlugInInterface object from
// IOCFCreatePlugInInterfaceForService.
herr = ( *plugInInterface )->QueryInterface ( plugInInterface,
            CFUUIDGetUUIDBytes (
            kIOMMCDeviceInterfaceID ),
            ( LPVOID *) &mmcInterface );
```

The device interface provides you with a wide range of functions you can use to access your device. With an IOFireWireDeviceInterface object, for example, you can perform a FireWire bus reset or create FireWire command object interfaces to perform asynchronous read, write, and lock operations. When you're finished with the device interface you acquired, you should call the IOCFPlugIn `Release` function to release it.

After you are completely finished with the specific device interface object, you should get rid of the intermediate IOCFPlugInInterface object by calling the `IODestroyPlugInInterface` function, defined in `IOCFPlugIn.h`.

Getting a Device-File Path

Before you can access a serial or storage device, you use I/O Kit functions to extract specific properties from the I/O Registry object that represents it and use them to create the device-file path. As described in "Inside the Device-File Mechanism" (page 21), device-file paths are usually of the form `/dev/`*device_name*, where *device_name* is defined by the type of device.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Important: The process of getting the pathname of a serial or storage device differs slightly depending on the type of device you're accessing, however, so be sure to read the documentation for your device type to get specific information on where the device name and path components are defined.

In general, you create the device-file path either by getting a C-string representation of a particular property or by concatenating strings defined in various properties or constants. To get the C-string representation for a property, you first use the I/O Kit function `IORegistryEntryCreateCFProperty` to get the property value as a Core Foundation string object. Then, you use the Core Foundation function `CFStringGetCString` to transform the CFString object into a C string.

For example, Listing 3-8 (page 39) shows how to get the device name of a storage device from the value of the `kIOBSDNameKey` (defined in `IOBSD.h`) and use this string, along with a constant string defined in `paths.h`, to create the full device-file path.

Listing 3-8 Getting the device name of a storage device

```
io_object_t device; //This is the object IOIteratorNext returns
char deviceFilePath[MAXPATHLEN]; //MAXPATHLEN is defined in sys/param.h
size_t devPathLength;
CFTypeRef deviceNameAsCFString;
Boolean gotString = false;

deviceNameAsCFString = IORegistryEntryCreateCFProperty (
            device,
            CFSTR(kIOBSDNameKey),
            kCFAllocatorDefault,0);
if (deviceNameAsCFString) {
    deviceFilePath = '\0';
    devPathLength = strlen(_PATH_DEV); //_PATH_DEV is defined in paths.h
    strcpy(deviceFilePath, _PATH_DEV);
    //Add "r" before the BSD node name from the I/O Registry
    //to specify the raw disk node. The raw disk node receives
    //I/O requests directly and does not go through the
    //buffer cache.
    strcat(deviceFilePath, "r");
    gotString = CFStringGetCString( deviceNameAsCFString,
             deviceFilePath + strlen(deviceFilePath),
             MAXPATHLEN - strlen(deviceFilePath),
            kCFStringEncodingASCII);

    if (gotString)
        printf("Device file path: %s\n", deviceFilePath);
        //deviceFilePath will look something like /dev/rdisk1
}
```

Now that you have the device file pathname as a C string, you can use it with standard POSIX functions, such as `open`, `read`, and `close`, to access the device.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

The IOKitLib API

When you access a device from user space, whether you use a device interface or the POSIX API, you use a combination of I/O Kit functions and specific device-family functions to work with the I/O Kit objects that represent the device. The IOKitLib, located in the I/O Kit framework, contains the generic I/O Kit functions you use to implement user-space task access to in-kernel device objects.

This chapter surveys the IOKitLib functions, describing how to use them to get access to and manipulate in-kernel objects and providing insight into how the IOKitLib implements these functions.

Some of the functions in the IOKitLib are intended for developers of custom device interfaces rather than for developers of applications that use existing device interfaces. Although this chapter covers these functions at a high level, you should read "Making Hardware Accessible to Applications" in *I/O Kit Device Driver Design Guidelines* if you need to develop your own device interface–user client solution.

The IOKitLib functions can be divided into several categories, based on the type of service they provide. This chapter mirrors these groupings and is divided into the following sections, each of which covers a functional category:

- "Object Reference-Counting and Introspection" (page 41) introduces the types of objects you use to communicate with in-kernel entities and describes how to get information about them and keep track of their reference counts.
- "Device Discovery and Notification" (page 43) describes the functions you use to create matching dictionaries, look up and access matching devices in the I/O Registry, and receive notifications about a device's state change.
- "I/O Registry Access" (page 47) describes other IOKitLib functions you can use to get access to objects in the I/O Registry.
- "Device-Interface Development" (page 52) gives a brief overview of the IOKitLib functions you use when no I/O Kit device-family or third-party device interface exists and you need to develop your own device interface.

Object Reference-Counting and Introspection

Functions in the IOKitLib communicate with in-kernel objects using the Mach port transport mechanism to cross the user-kernel boundary. The file `IOTypes.h`, located in the I/O Kit framework, defines the objects you use with IOKitLib functions to communicate with such in-kernel entities as I/O Registry entries (objects of class IORegistryEntry) or iterators (objects of class IOIterator). Notice in Listing 4-1 (page 41), however, that `IOTypes.h` seems to define all these objects in the same way, specifically, as `mach_port_t` objects (`mach_port_t` is defined in `mach/port.h` in the Kernel framework).

Listing 4-1 Object definitions in IOTypes.h

```
typedef mach_port_t io_object_t;
```


2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

```
typedef io_object_t io_connect_t;
typedef io_object_t io_iterator_t;
typedef io_object_t io_registry_entry_t;
typedef io_object_t io_service_t;
typedef io_object_t io_enumerator_t;
```

To understand why this is so, recall that a Mach port is the communication transport mechanism an application uses to communicate with the I/O Kit. As far as the kernel is concerned, every time your application communicates with any in-kernel object using one of the object types defined in `IOTypes.h`, it's using the same generic mechanism (namely, a Mach port) to cross the user-kernel boundary.

From the application's point of view, however, the transport mechanism is unimportant and what matters is the type of object on the other side of the port. The fact that an `io_iterator_t` object, for example, encapsulates the association of a Mach port with an in-kernel IOIterator object is not as important to the application as the fact that an `io_iterator_t` object refers to an in-kernel object that knows how to iterate over the I/O Registry.

The `io_object_t` objects in user space not only refer to the in-kernel objects, they also reflect the in-kernel C++ class hierarchy. Because IOService is a subclass of IORegistryEntry, for example, you can use an `io_service_t` object with any IOKitLib function that expects an `io_registry_entry_t` object, such as `IORegistryEntryGetPath`.

With few exceptions (such as getting the I/O Kit master port to initiate communication with the I/O Kit and reference-counting of the `io_object_t` objects themselves) you should not concern yourself with the cross-boundary communication mechanism. Rather, you should focus on the specific in-kernel object you are working with, making sure you use each object type appropriately.

Reference Counting

The IOKitLib contains three reference-counting functions:

- `IOObjectGetRetainCount`
- `IOObjectRetain`
- `IOObjectRelease`

Each of these functions operates on the retain count of the underlying kernel object (the object on the other side of the Mach port) and not on the `io_object_t` object itself. In other words, if you want to retain, release, or get the retain count of the in-kernel IORegistryEntry object an `io_registry_entry_t` object represents, you use one of the IOKitLib reference-counting functions. For example, Listing 4-2 (page 42) shows a code fragment that releases the in-kernel IOIterator object underlying an `io_iterator_t` object after it's been used to find a serial port modem.

Listing 4-2 Releasing the underlying kernel object of an `io_iterator_t` object

```
io_iterator_t serialPortIterator = NULL;
char deviceFilePath[ MAXPATHLEN ];

kernResult = MyFindSerialPortModems(&serialPortIterator, &masterPort);
if (kernResult == kIOReturnSuccess) {
    kernResult = MyGetPathOfFirstModem (serialPortIterator, deviceFilePath,
            sizeof(deviceFilePath));
```


2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

```
    //Release the iterator since we want only the first modem.
    IOObjectRelease(serialPortIterator);
    //If MyGetPathOfFirstModem found a modem, communicate with it.
...
}
```

As a matter of good programming practice, you should use `IOObjectRelease` to release all `io_object_t` objects you create in your code when they're no longer needed.

If you suspect that you are leaking `io_object_t` objects, however, `IOObjectGetRetainCount` won't help you because this function informs you of the underlying kernel object's retain count (which is often much higher). Instead, because the retain count of an `io_object_t` object is essentially the retain count of the send rights on the Mach port, you use a Mach function to get this information. Listing 4-3 (page 43) shows how to use the Mach function`mach_port_get_refs` (defined in `mach/mach_port.h` in the Kernel framework) to get the retain count of an `io_object_t` object.

Listing 4-3 Getting the retain count of an `io_object_t` object

```
#include <mach/mach_port.h>

kern_return_t kr;
unsigned int count;
io_object_t theObject;

kr = mach_port_get_refs ( mach_task_self(), theObject, MACH_PORT_RIGHT_SEND,
            &count );
printf ("Retain count for object ID %#X is %d\n", theObject, count);
```

Introspection

The IOKitLib provides three object-introspection functions that operate on the in-kernel object the passed-in `io_object_t` object represents (not the `io_object_t` object itself):

- `IOObjectConformsTo`
- `IOObjectGetClass`
- `IOObjectIsEqualTo`

The first function, `IOObjectConformsTo`, simply performs the in-kernel object's `metaCast` method and returns the Boolean result. The `IOObjectGetClass` function calls the in-kernel object's `getMetaClass` method and then calls that class's `getClassName` method to return the class name as a C string. IOKitLib implements the `IOObjectIsEqualTo` function as a shallow pointer comparison of the two passed-in objects, returning the result as a Boolean value.

Device Discovery and Notification

Even if you use few other IOKitLib functions in your device-access application, you will certainly use the device-discovery functions. If you're working with hot-pluggable devices, such as USB or FireWire devices, you may also use the notification functions.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

The following sections cover the IOKitLib device-discovery and notification functions, describing how the IOKitLib implements:

- Matching dictionary–creation
- Device look-up
- Notification set-up
- Device iteration

Creating Matching Dictionaries

When you use IOKitLib functions to create a matching dictionary, you receive a reference to a Core Foundation dictionary object. The IOKitLib uses Core Foundation classes, such as CFMutableDictionary and CFString, because they closely corrrespond to the in-kernel collection and container classes, such as OSDictionary and OSString (defined in `libkern/c++` in the Kernel framework).

The I/O Kit automatically translates a CFDictionary object into its in-kernel counterpart when it crosses the user-kernel boundary, allowing you to create an object in user-space that is later used in the kernel. For more information on using Core Foundation objects to represent in-kernel objects, see "Viewing Properties of I/O Registry Objects" (page 49).

The IOKitLib defines the following functions to create matching dictionaries:

- `IOServiceMatching`
- `IOServiceNameMatching`
- `IOBSDNameMatching`

These functions create a mutable Core Foundation dictionary object containing the appropriate key and your passed-in value. Table 4-1 (page 44) shows the keys each of the dictionary-creation functions use, along with the I/O Kit framework files in which the keys are defined.

Table 4-1 Dictionary-creation functions and the keys they use

Function name	Key name	Key-definition file
`IOServiceMatching`	`kIOProviderClassKey`	`IOKitKeys.h`
`IOServiceNameMatching`	`kIONameMatchKey`	`IOKitKeys.h`
`IOBSDNameMatching`	`kIOBSDNameKey`	`IOBSD.h`

If, for example, you call `IOServiceNameMatching` with the argument *i2c-modem*, the resulting dictionary looks like this:

```
{
    IONameMatch = i2c-modem;
}
```


2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

All dictionary-creation functions return a reference to a CFMutableDictionary object. Usually, you pass the dictionary to one of the look-up functions (discussed next in "Looking Up Devices" (page 45)), each of which consumes one reference to it. If you use the dictionary in some other way, you should adjust its retain count accordingly, using `CFRetain` or `CFRelease` (defined in the Core Foundation framework).

Looking Up Devices

IOKitLib provides three look-up functions that look up registered objects in the I/O Registry that match a passed-in matching dictionary:

- `IOServiceGetMatchingServices`
- `IOServiceGetMatchingService`
- `IOServiceAddMatchingNotification`

IOKitLib implements its most general look-up function, `IOServiceGetMatchingServices`, by transforming your matching dictionary into an OSDictionary object and invoking the IOService method `getMatchingServices`. This method returns an IOIterator object that contains a list of matching IOService objects. `IOServiceGetMatchingServices` then releases the matching dictionary you passed in and returns to you an `io_iterator_t` object representing the in-kernel IOIterator object.

The `IOServiceGetMatchingService` function is simply a special case of `IOServiceGetMatchingServices`: It returns the first matching object in the list instead of an iterator that provides access to the entire list.

If you receive an `io_iterator_t` object from `IOServiceGetMatchingServices`, you should release it with `IOObjectRelease` when you're finished with it; similarly, you should use `IOObjectRelease` to release the `io_object_t` object you receive from `IOServiceGetMatchingService`.

The `IOServiceAddMatchingNotification` function not only looks up matching objects in the I/O Registry, it also installs a request for notification of matching objects that meet your passed-in criterion. The IOKitLib implements this function by invoking the IOService `addNotification` method, which creates a persistent notification handler that can be notified of IOService events.

To use `IOServiceAddMatchingNotification`, however, you must first use some other functions in the IOKitLib to set up the notification mechanism. For more information on these functions, see the next section, "Setting Up and Receiving Notifications" (page 45). For an outline of the steps your application takes to set up a notification mechanism using a run loop, see "Getting Notifications of Device Arrival and Departure" (page 35).

Setting Up and Receiving Notifications

The IOKitLib provides several functions you use to set up and manage notification objects:

- `IONotificationPortCreate`
- `IONotificationPortGetRunLoopSource`
- `IONotificationPortGetMachPort`
- `IODispatchCalloutFromMessage`

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

- IONotificationPortDestroy

Whether you choose to receive notifications on a run loop or a Mach port (using a run loop is the recommended approach), you must first create an object that can listen for I/O Kit notifications. To do this, you use the IONotificationPortCreate function, which returns an object of type IONotificationPortRef. IOKitLib implements this function by allocating a receive right on the current Mach port, giving it the name of the IONotificationPort object you pass in.

To receive notifications on a run loop, you first use the IONotificationPortGetRunLoopSource function to get a run-loop source you can install on your application's current run loop. In Mac OS X, a run loop registers input sources, such as Mach ports, and enables the delivery of events, such as IOService object status changes, through those sources. The run-loop source object you receive from IONotificationPortGetRunLoopSource function is of type CFRunLoopSourceRef. (For more information about how to install the run-loop source on your application's run loop, see "Getting Notifications of Device Arrival and Departure" (page 35).)

The IOKitLib provides two functions that allow you use a Mach port you create on which to listen for notifications, rather than a run loop. Although this method is available to you, it's recommended that you use the easier-to-implement and more automatic run-loop solution instead. The IONotificationPortGetMachPort function returns a Mach port on which the IONotificationPortRef object can listen for notifications. When the notification object receives a message, you pass the message to the IODispatchCalloutFromMessage function to generate the callback function associated with the notification.

The IONotificationPortDestroy function cleans up and destroys all rights named by the passed-in port name (the IONotificationPortRef object you received from IONotficationPortCreate). You should call this function when you no longer want to receive notifications.

Iterating Over Matching Devices

To access the matching objects a look-up function returns, you use the IOIteratorNext function. IOKitLib contains three functions that operate on io_iterator_t objects:

- IOIteratorNext
- IOIteratorIsValid
- IOIteratorReset

As a subclass of OSIterator, the in-kernel IOIterator object defines the methods getNextObject, isValid, and reset. IOKitLib invokes these methods in its implementation of IOIteratorNext, IOIteratorIsValid, and IOIteratorReset, respectively.

IOIteratorNext simply returns a reference to the current object in the list (an io_object_t object) and advances the iterator to the next object.

IOIteratorIsValid returns a Boolean value indicating whether the iterator is still valid. Sometimes, if the I/O Registry changes while you're using an iterator, the iterator becomes invalid. When this is the case, IOIteratorIsValid returns zero and you can reset the invalid iterator to the beginning of the list, using IOIteratorReset. Of course, you can use IOIteratorReset to start over at the beginning of the list for any reason, not only because the iterator is invalid.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

I/O Registry Access

The IOKitLib provides a range of functions that find and provide information about objects in the I/O Registry. Unlike the device-discovery functions, which focus on finding and accessing registered device objects that match your matching dictionary, the I/O Registry–access functions allow you to view objects more broadly as entries in the I/O Registry, providing ways to freely navigate the I/O Registry and get information about any object in it. If you're unfamiliar with the structure of the I/O Registry, you can read more about it in "The I/O Registry" in *I/O Kit Fundamentals*. The following paragraphs provide a summary.

The I/O Registry is a dynamic database that captures the connections of all driver and nub objects currently active in a running Mac OS X system. You can think of the I/O Registry as a tree-like structure rooted in the Platform Expert, the driver of the main logic board that knows the type of platform your Mac OS X system is running on.

The I/O Registry uses the concept of a plane to describe the different provider-client relationships between objects. The Service plane displays the instantiation hierarchy of the I/O Registry: Every object in the I/O Registry is a client of the services provided by its parent, so every object's connection to its ancestor in the I/O Registry tree is visible on the Service plane. Other planes express the provider-client relationships between objects that participate in a specific hierarchy, such as the flow of power from power provider to power client.

Many of the I/O Registry–access functions require you to pass in an argument designating a specific plane. In most cases, you will probably specify the all-inclusive Service plane, but there may be times when you want to narrow the focus a bit. The file `IOKitKeys.h` (in the I/O Kit framework) contains the plane name definitions you use in your code.

The IOKitLib provides I/O Registry–access functions that:

- Traverse the I/O Registry in some way
- Provide information about a particular object in the I/O Registry
- Allow you to view an I/O Registry object's properties
- Allow you to set properties in an I/O Registy object's property table

IOKitLib implements most of these functions by invoking similarly named IORegistryEntry methods (see `IORegistryEntry.h` in the `IOKit` directory of the Kernel framework for more). The following sections describe how you use the I/O Registry functions.

Traversing the I/O Registry

When you use the `IOServiceGetMatchingServices` look-up function, you get an iterator you can use to access each object in a list of matching I/O Registry objects. But what if you want to search the I/O Registry using some criteria other than a matching dictionary? Say, for example, you want to find all the children (clients) of a particular object, regardless of whether they are currently registered in the I/O Registry. The following functions give you access to the parents and children of an `io_registry_entry_t` object:

- `IORegistryEntryGetChildEntry`
- `IORegistryEntryGetChildIterator`
- `IORegistryEntryGetParentEntry`
- `IORegistryEntryGetParentIterator`

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Not surprisingly, the functions with the word "iterator" in their names supply you with an `io_iterator_t` object you can pass to `IOIteratorNext` to access each `io_registry_entry_t` object in the list. `IORegistryEntryGetParentIterator`, for example, looks in the specified plane to find all ancestors of the passed-in object and returns an iterator you can use to access them. As you do with the `IOServiceGetMatchingService` and `IOServiceAddMatchingNotification` functions, be sure to release the returned `io_iterator_t` object when you're finished with it (recall that releasing the iterator you receive from `IOServiceAddMatchingNotification` also disables that notification). The other two functions, `IORegistryEntryGetChildEntry` and `IORegistryEntryGetParentEntry`, return the first child or ancestor of the passed-in object in the specified plane.

For even broader searches, you can use the following functions to search the entire I/O Registry, from any point and to any depth:

- `IORegistryCreateIterator`
- `IORegistryEntryCreateIterator`
- `IORegistryIteratorEnterEntry`
- `IORegistryIteratorExitEntry`
- `IORegistryGetRootEntry`

The `IORegistryCreateIterator` function returns an `io_iterator_t` object that begins at the root of the I/O Registry and iterates over all its children in the specified plane. Similarly, `IORegistryEntryCreateIterator` returns an `io_iterator_t` object that begins at the passed-in `io_registry_entry_t` object and iterates over either all its ancestors or all its children in the passed-in plane.

Both these functions allow you to specify an option that tells the iterator to automatically recurse into objects as they are returned, or only when you call `IORegistryIteratorEnterEntry` to recurse into the current object. The `IORegistryIteratorExitEntry` function undoes the effect of `IORegistryIteratorEnterEntry` by resetting the iterator to the current object, allowing the iterator to continue from where it left off. As its name implies, `IORegistryGetRootEntry` returns an `io_registry_entry_t` object representing the root of the I/O Registry.

Getting Information About I/O Registry Objects

The IOKitLib provides a handful of functions, listed below, that return information about a particular I/O Registry object, such as its name or path.

- `IORegistryEntryGetName`
- `IORegistryEntryGetNameInPlane`
- `IORegistryEntryGetPath`
- `IORegistryEntryGetLocationInPlane`
- `IORegistryEntryInPlane`

IOKitLib implements these functions by invoking IORegistryEntry methods such as `getName` and `getLocation`. Each function returns a C string that contains the object's name or location. For example, Listing 4-4 (page 49) shows how to use some of these functions on the object representing an Apple USB keyboard.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Listing 4-4 Using the `IORegistryEntryGet` functions

```
io_service_t device;
io_name_t devName;
io_string_t pathName;

IORegistryEntryGetName(device, devName);
printf("Device's name = %s\n", devName);
IORegistryEntryGetPath(device, kIOServicePlane, pathName);
printf("Device's path in IOService plane = %s\n", pathName);
IORegistryEntryGetPath(device, kIOUSBPlane, pathName);
printf("Device's path in IOUSB plane = %s\n", pathName);
```

The code in Listing 4-4 (page 49) displays information like the following:

```
Device's name = Apple Extended USB Keyboard
Device's path in IOService plane = IOService:/MacRISC2PE/pci@f2000000/
    AppleMacRiscPCI/usb@18/AppleUSBOHCI/Apple Extended USB Keyboard@1211000
Device's path in IOUSB plane = IOUSB:/IOUSBHubDevice@1200000/Hub in Apple
    Extended USB Keyboard@1210000/Apple Extended USB Keyboard@1211000
```

If you're more interested in whether an object exists in a particular plane than in its path or name, you can use the `IORegistryEntryInPlane` function, which returns `TRUE` if the object has a parent in the plane and `FALSE` otherwise.

Viewing Properties of I/O Registry Objects

Each object in the I/O Registry has two dictionaries associated with it:

- The property table for the object—if the object is a driver, this is the matching dictionary that defines one of the driver's personalities (if the object is a nub, the device family publishes device properties in the nub's property table).
- The plane dictionary—this dictionary describes how the object is connected to other objects in the I/O Registry.

These dictionaries are instances of OSDictionary. The libkern C++ library defines container classes, which hold primitive values, such as numbers and strings, and collection classes, which hold groups of both container objects and other collection objects. To view the contents of these objects in an application, you must use IOKitLib functions to convert the in-kernel collection or container object (such as OSDictionary or OSString) to the equivalent Core Foundation object. Currently, only the following libkern classes are available as Core Foundation analogues:

- OSDictionary
- OSArray
- OSSet
- OSSymbol
- OSString
- OSData
- OSNumber
- OSBoolean

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

The following IOKitLib functions allow you to access an object's properties as Core Foundation types:

- `IORegistryEntryCreateCFProperty`
- `IORegistryEntryCreateCFProperties`
- `IORegistrySearchCFProperty`

The `IORegistryEntryCreateCFProperty` function creates a Core Foundation container representing the value associated with the passed-in property key. For example, you can get a CFTypeRef to a storage device's device-file name by passing a handle to the device (an `io_object_t` object) and the key `kIOBSDNameKey` (defined in `IOBSD.h`) to `IORegistryEntryCreateCFProperty`:

```
CFTypeRef deviceFileName;
deviceFileName = IORegistryEntryCreateCFProperty(deviceHandle,
                    CFSTR(kIOBSDNameKey), kCFAllocatorDefault, 0);
```

The `IORegistryEntryCreateCFProperties` function creates a Core Foundation dictionary object containing the passed-in object's property table. After your application gets the CFDictionary representation of an object's property table, it can call Core Foundation functions (such as `CFDictionaryGetValue` and `CFNumberGetValue`) to access the dictionary's values.

The `IORegistryEntrySearchCFProperty` function searches for the passed-in property, beginning with a specified `io_registry_entry_t` object's property table, and continuing with either its parents or its children in the specified plane. When it finds the property you passed in, it returns a Core Foundation container object representing the property.

When you're finished with the Core Foundation container object you receive from these functions, you should call `CFRelease` on it.

Because the I/O Registry is a dynamic database that is constantly being updated, the properties a function such as `IORegistryEntryCreateCFProperties` returns represent a snapshot of the I/O Registry's state at a single instant. For this reason, you should not assume that the properties you receive are static—if you call these functions multiple times in your application, you may get different results each time.

Setting Properties of I/O Registry Objects

In addition to viewing an object's properties in an application, you can also use IOKitLib functions to place new properties into an object's property table.

> **Important:** This is not an action most applications need to take. Because these functions rely on the implementation of the `setProperties` method in the corresponding in-kernel driver, you should not use them unless your device family specifically recommends it or unless you control both the user and kernel side of a connection.

Setting properties from user space is suitable for some types of device control, in particular, single downloads of data, such as the loading of firmware. This type of user space–kernel communication works because both the application and the in-kernel object have access to the device object's property table in the I/O Registry. Of course, when you use IOKitLib functions to set properties in an object's property table, it's important to realize that you're only manipulating the OSDictionary representation of an object's information property list, not the property list itself.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Before you consider setting an object's properties in your application, make sure your situation meets the following conditions:

- Your device family implements the `setProperties` method or you control the in-kernel driver, and in it, you implement the `setProperties` method.
- The driver does not have to allocate permanent resources to complete the transaction.
- The data sent either causes no change in the driver's state, or causes a single, permanent change.
- If the application transfers data by copy, it sends only a limited amount, such as a page or less. (If the application sends data by reference, it can send an arbitrary amount of data.)

The IOKitLib provides four functions that allow you to set properties in an object's I/O Registry property table:

- `IORegistryEntrySetCFProperty`
- `IORegistryEntrySetCFProperties`
- `IOConnectSetCFProperty`
- `IOConnectSetCFProperties`

The first two functions, `IORegistryEntrySetCFProperty` and `IORegistryEntrySetCFProperties`, are generic functions that set either a single property value or a collection of property values (typically, in a CFDictionary object) in the specified `io_registry_entry_t` object. The in-kernel object interprets the Core Foundation container or collection object as it chooses.

If you're developing your own device interface–user client solution, you're more likely to use the `IOConnectSetCFProperty` and `IOConnectSetCFProperties` functions to set properties. This is because you are working with the `io_connect_t` object that represents the connection you opened to the in-kernel driver object with `IOServiceOpen` and these functions provide more access control than `IORegistryEntrySetProperty` and `IORegistryEntrySetProperties`. For a brief overview of how to create a connection to a user client, see "Device-Interface Development" (page 52).

Determining Busy States

The IOKitLib includes a handful of functions that give you information about the busy state of various IOService objects and allow you to wait until these objects are no longer busy:

- `IOServiceGetBusyState`
- `IOKitGetBusyState`
- `IOServiceWaitQuiet`
- `IOKitWaitQuiet`

Because activities such as registration, matching, and termination are asynchronous, an IOService object in the process of one of these activities is considered busy and its `busyState` value is increased by one. When any of these activities concludes, the IOService's `busyState` value is decreased by one. Additionally, any time an IOService object's `busyState` value changes, its provider's `busyState` value changes to match, which means that an IOService object is considered to be busy when any of its clients is busy.

To get the busy state of an individual `io_service_t` object, you use the `IOServiceGetBusyState` function, which IOKitLib implements by invoking the object's `getBusyState` method.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

You can get the busy state of all IOService objects by calling the IOKitGetBusyState function. As the provider of all IOService objects in the I/O Registry, the root of the Service plane is busy when any of its clients (in other words, any IOService object) is busy. The IOKitGetBusyState function returns the busy state of the Service plane root, effectively informing you of the busy state of the I/O Registry as a whole.

You can wait until objects cease to be busy by calling either the IOServiceWaitQuiet or IOKitWaitQuiet functions. Both functions allow you to specify a maximum time to wait and block the calling process until either the time runs out or the object becomes nonbusy.

Device-Interface Development

If you need to implement both an in-kernel user client and a device-interface library, you should read "Making Hardware Accessible" in *I/O Kit Device Driver Design Guidelines* for in-depth information on how to do this. This section covers only the user-space side of this process, briefly describing the IOKitLib functions involved.

Important: The IOKitLib contains several functions that are intended for developers of custom user clients and device interfaces. If you are using an I/O Kit family (or third-party) device interface to access a device, you will not need to use these functions; in fact, using these functions may circumvent the family's (or third party's) device-interface functions and produce undesired results.

Although you may not directly use the IOKitLib functions this sections describes, knowing how they work will give you some insight into how device interfaces and user clients cooperate.

Creating a User Space–Kernel Connection

The IOKitLib provides several functions that allow you to create and manipulate connections to in-kernel objects, typically user clients. As with the other functions described in this section, you should use them only when your device family does not provide a device-interface solution and you've decided to implement both the in-kernel user client and the user-space device interface.

For an application to communicate with a device, the first thing it must do is create a connection between itself and the in-kernel object representing the device. To do this, it creates a user client object.

After it gets the io_service_t object representing the device driver (by calling IOServiceGetMatchingServices, for example), the application calls the IOKitLib IOServiceOpen function to create the connection. The IOServiceOpen function invokes the io_service_t object's newUserClient method, which instantiates, initializes, and attaches the user client. As a result of calling IOServiceOpen, the application receives an io_connect_t object (representing the user-client object) that it can use with the IOKitLib IOConnect functions.

For example, in the *SimpleUserClient* example project, the application portion of the project creates a user client with code like that shown in Listing 4-5 (page 52).

Listing 4-5 Creating a user client

```
//Code to get the I/O Kit master port, create a matching dictionary, and get
//an io_service_t object representing the in-kernel driver is not shown here.
io_service_t serviceObject;
io_connect_t dataPort;
```


2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

```
kern_return_t kernResult;

kernResult = IOServiceOpen(serviceObject, mach_task_self(), 0, &dataPort);
```

An application destroys a connection to a user client with a call to the `IOServiceClose` function, which invokes the `clientClose` method in the user client. If, for some reason, the application terminates before it's able to call `IOServiceClose`, the user client invokes the `clientDied` method. Typically, the user client responds to both methods by invoking `close` on its provider (usually the device nub).

If you're developing your own user client and device-interface library, the IOKitLib functions you're most likely to use are the four `IOConnectMethod` functions (which call the user client's external methods) and, if your user client can map hardware registers into your application's address space, `IOConnectMapMemory` and `IOConnectUnmapMemory`.

IOConnectMethod Functions

The `IOConnectMethod` functions use arrays of structures containing pointers to methods to invoke in a user-client object. The user client defines the list of methods it implements in an `IOExternalMethod` array. Typically, these methods include the user client's `open` and `close` methods and methods that pass data between the in-kernel driver and the application. To use an `IOConnectMethod` function, an application passes in the `io_connect_t` object, the index into the `IOExternalMethod` array, and, if the application is passing or receiving data, arguments that describe the general data type (scalar or structure), number of scalar parameters, size of data structures, and direction (input or output).

To open the user client, the application calls any of the `IOConnectMethod` functions, passing in just the `io_connect_t` object representing the user client and the `IOExternalMethod` array index corresponding to the user client's `open` method. Because the application is not passing or receiving any data at this point, the remaining two arguments passed to the `IOConnectMethod` function are zero. For example, the code below shows how the application in the SimpleUserClient project uses the `io_connect_t` object it received in Listing 4-5 (page 52) to open its user client:

```
//kMyUserClientOpen is one of the enum constants the SimpleUserClient
//project uses as indexes into the IOExternalMethod array.
kernResult = IOConnectMethodScalarIScalarO(dataPort, kMyUserClientOpen,
                0, 0);
```

To pass untyped data back and forth across the user-kernel boundary, an application uses one of the `IOConnectMethod` functions in Table 4-2 (page 53), depending on the type of data and the direction of data flow.

Table 4-2 `IOConnectMethod` functions

Function	Description
`IOConnectMethodScalarIScalarO`	One or more scalar input parameters, one or more scalar output parameters
`IOConnectMethodScalarIStructureO`	One or more scalar input parameters, one structure output parameter
`IOConnectMethodScalarIStructureI`	One or more scalar input parameters, one structure input parameter

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Function	Description
IOConnectMethodStructureIStructureO	One structure input parameter, one structure output parameter

The IOConnectMethod functions are designed to accept variable argument lists, depending on the data type and direction you choose. Following the first two arguments—the io_connect_t object and the method array index—is some combination of arguments that identify:

- The number of scalar input or output values
- The size of the input or output structure
- Scalar input or output values
- A pointer to an input or output structure

The application in the SimpleUserClient, for example, uses the IOConnectMethodScalarIStructureI function as shown in Listing 4-6 (page 54).

Listing 4-6 Requesting I/O with the IOConnectMethodScalarIStructureI function

```
//MySampleStruct is defined in the header file both the user client
//and the application include and consists of two integer variable
//declarations.
MySampleStruct sampleStruct = {586, 8756}; //Random numbers.
int sampleNumber = 15; //Random number.
IOByteCount structSize = sizeof(MySampleStruct);
kern_return_t kernResult;

kernResult = IOConnectMethodScalarIStructureI(dataPort, // from IOServiceOpen
                kMyScalarIStructImethod, // method index
                1, // number of scalar input values
                structSize, // size of input structure
                sampleNumber, // scalar input value
                &sampleStruct // pointer to input structure
                );
```

Memory-Mapping Functions

If your device driver has full PIO (Programmed Input/Output) memory management and your device does not require the use of interrupts, you can write a dedicated application that moves large amounts of data to and from the device, using the IOConnectMapMemory and IOConnectUnmapMemory functions. For example, a frame-buffer application can use these functions to handle the large amounts of on-board memory that needs to be accessible from user space. The IOConnectMapMemory function allows a user process to share memory with an in-kernel driver by mapping the same memory into both processes.

When a user process calls IOConnectMapMemory, it passes in, among other arguments, a pointer to an area in its own address space that will contain the mapped memory. The call to IOConnectMapMemory causes the invocation of the user client's clientMemoryForType method. The user client implements this method by creating an IOMemoryDescriptor object that backs the mapping to the hardware registers. The user process

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

receives a virtual memory `vm_address_t` object containing the address of the mapped memory and a `vm_size_t` object that contains the size of the mapping. At this point, the user process can freely write to and read from the hardware registers.

When the user process or device interface is finished performing I/O with the mapped memory, it should call `IOConnectUnmapMemory` to remove the mapping.

Managing the User Space–Kernel Connection

The remaining `IOConnect` functions in IOKitLib are primarily intended to help you manage your custom user space–kernel connection. As with the other `IOConnect` functions, you should not use them with I/O Kit family-supplied or third-party device interfaces because they operate directly on the `io_connect_t` object representing the connection, circumventing the existing device interface.

The IOKitLib contains the following functions to manipulate the `io_connect_t` object:

- `IOConnectAddClient`
- `IOConnectAddRef`
- `IOConnectRelease`
- `IOConnectGetService`
- `IOConnectSetNotificationPort`

The `IOConnectAddClient` function creates a connection between two user-client objects in the kernel by invoking the first user client's `connectClient` method. You can use this function in the unlikely event that you have two user clients that need to be able to communicate.

As their names suggest, `IOConnectAddRef` and `IOConnectRelease` adjust the reference count on the passed-in `io_connect_t` object.

The `IOConnectGetService` function returns an `io_service_t` object representing the IOService object on which the passed-in `io_connect_t` object was opened. IOKitLib implements this function by invoking the user client's `getService` method.

To identify a Mach port on which to receive family-specific notifications (notification types not interpreted by the I/O Kit), you use the `IOConnectSetNotificationPort` function. You pass in, among other arguments, the `io_connect_t` object representing the user client connection and a `mach_port_t` object representing the Mach port on which you want to receive notifications. The IOKitLib implements this function by invoking the user client's `registerNotificationPort` method.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Handling Errors

All programming tasks carry the potential for error. Aside from standard errors of syntax or logic, however, applications that access hardware may encounter specific types of errors that are outside the experience of many application developers.

This chapter covers some of the errors you might see while developing an application that uses the I/O Kit to access hardware on a Mac OS X system. It describes how to "read" the I/O Kit's error codes and then provides some information on the exclusive-access error.

Interpreting I/O Kit Error Return Values

Because you'll be using I/O Kit functions extensively while looking up devices in the I/O Registry and examining the objects that represent them, you should be familiar with the structure of the I/O Kit's error return values. This section describes how your application can interpret error values returned by I/O Kit functions.

The I/O Kit uses an error return mechanism, defined by the kernel framework, in which a 32-bit, unsigned return value supplies information in three separate bit fields, as shown in Figure 5-1 (page 57). That is:

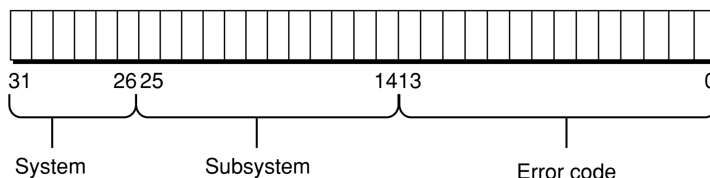
- The high 6 bits specify the system in which the error occurred.
- The next 12 bits specify the subsystem.
- The final 14 bits specify the error code itself.

To work with these error return values, the header file `IOReturn.h` defines the following type:

```
typedef kern_return_t IOReturn;    //kern_return_t is an int
```

You can use `kern_return_t` and `IOReturn` to obtain return values from I/O Kit functions that use either type.

Figure 5-1 Bit layout for kernel and I/O Kit error return values



Listing 5-1 (page 58) shows some common system error values that are defined in the header `Kernel.framework/Headers/mach/error.h`.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Listing 5-1 Error.h system error values

```
#define     err_kern    err_system(0x0)     /* kernel */
#define     err_us      err_system(0x1)     /* user space library */
#define     err_server  err_system(0x2)     /* user space servers */
#define     err_ipc     err_system(0x3)     /* old ipc errors */
#define     err_mach_ipc err_system(0x4)    /* mach-ipc errors */
#define     err_dipc    err_system(0x7)     /* distributed ipc */
#define     err_local   err_system(0x3e)    /* user defined errors */
#define     err_ipc_compat err_system(0x3f) /* (compatibility) mach-ipc errors
 */
#define     err_max_system 0x3f
```

This header also defines macros for defining system and subsystem error values and for extracting system, subsystem, and code values from an error return value, as shown in Listing 5-2 (page 58).

Listing 5-2 Error.h macros for working with error return values

```
#define     err_system(x)       (((x)&0x3f)<<26)
#define     err_sub(x)          (((x)&0xfff)<<14)

#define     err_get_system(err) (((err)>>26)&0x3f)
#define     err_get_sub(err)    (((err)>>14)&0xfff)
#define     err_get_code(err)   ((err)&0x3fff)
```

Additional system values, as well as subsystem and code values, are defined in header files for particular systems. For example, the header file `Kernel.framework/Headers/IOKit/IOReturn.h` defines the values shown in Listing 5-3 (page 58) for the I/O Kit.

Listing 5-3 IOReturn.h error return values

```
#ifndef sys_iokit
#define sys_iokit                       err_system(0x38)
#endif /* sys_iokit */
#define sub_iokit_common                err_sub(0)
#define sub_iokit_usb                   err_sub(1)
#define sub_iokit_firewire              err_sub(2)
#define sub_iokit_reserved              err_sub(-1)
#define iokit_common_err(return)        (sys_iokit|sub_iokit_common|return)

#define kIOReturnSuccess         KERN_SUCCESS            // OK
#define kIOReturnError           iokit_common_err(0x2bc) // general error
#define kIOReturnNoMemory       iokit_common_err(0x2bd) // can't allocate memory
#define kIOReturnNoResources     iokit_common_err(0x2be) // resource shortage
#define kIOReturnIPCError        iokit_common_err(0x2bf) // error during IPC
#define kIOReturnNoDevice        iokit_common_err(0x2c0) // no such device
// ... (many more individual error codes)
```

Your application may be able to use these error values directly, without having to extract system, subsystem, or code values. For example, you could use code like the following to check for a no device error:

```
IOReturn returnVal;

returnVal = IOKitSomeFunction(...);

if (returnVal == kIOReturnNoDevice)
{
    // "No device returned" error in I/O Kit system, common subsystem.
```


2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

```
}
```

However, in some cases you may want to isolate an error code value or determine which system or subsystem an error value came from. From the definitions shown in Listing 5-3 (page 58), and a little bit of calculation, you can see that the error return value for a no device error (`kIOReturnNoDevice`) in the I/O Kit system and the I/O Kit common subsystem would have the following bit-field values:

- The high 6 bits (31–26) have the value `11 1000`, or `0x38`.
- The next 12 bits (25–14) have the value `00 0000 0000 00`, or `0x0`.
- The final 14 bits (13–0) have the value `00 0010 1010 0000`, or `0x2c0`.

The fully assembled bit representation is `1110 0000 0000 0000 0000 0010 1011 1100`, resulting in a hex value of `0xe00002c0`. To extract the system, subsystem, or code value from such an error return value, you use the macros shown in Listing 5-2 (page 58) along with the constants defined in Listing 5-3 (page 58). For example:

```
IOReturn returnVal;

returnVal = IOKitSomeFunction(...);

if (err_get_system(returnVal) == err_get_system(sys_iokit))
{
    // The error was in the I/O Kit system
    UInt32 codeValue = err_get_code(returnVal);
    // Can now perform test on error code, display it to user, or whatever.
}
```

Handling Exclusive-Access Errors

Many types of devices are designed to allow only one process at a time to access the device. A scanner, for example, supports access by only one application at a time. Accordingly, I/O Kit families enforce the access policy for their devices, whether exclusive or shared. In addition, Classic (the Mac OS 9 environment that you can run in Mac OS X) may expect its drivers to have exclusive access to some devices, such as USB devices. In the course of developing an application that accesses hardware, you may receive an exclusive-access error, even when you believe yours is the only process trying to open the device. When this happens, if there is no higher-level arbitration you can employ, you may need to present a dialog to the user and either try accessing another device of the same class or try to access the same device later or under different circumstances.

Defined in `IOReturn.h` (in the I/O Kit framework), the `kIOReturnExclusiveAccess` error tells your application that the device it is attempting to access has already been opened by another entity. In most cases, the user client enforces exclusive access in its `open` method. When an application uses a device interface's `open` function, the device interface issues an `open` command to the user client. The user client responds by trying to open its provider (the device nub) and, if it fails, it returns the `kIOReturnExclusiveAccess` error. (If the user client finds that the provider is terminated, it will probably return the `kIOReturnNotAttached` error.)

Different I/O Kit device families handle the exclusive-access issue in different ways. The FireWire family, for example, enforces exclusive access to its devices, but also allows multiple device interfaces to open different objects in the FireWire driver stack. It does this by employing the concept of a session reference to refer to existing user space–kernel connections. Consider an application that uses a FireWire family device interface

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

to open an AV/C unit on a FireWire device. If that application also wants to access the FireWire unit object that supports the AV/C unit, it can get a session reference from the AV/C device interface and use it to get the FireWire unit device interface. For more information on this process, see *FireWire Device Interface Guide*.

The SCSI Architecture Model family, on the other hand, does not allow multiple device interfaces to simultaneously open different objects in the driver stack, but it does allow applications to get information about devices that in-kernel drivers currently have open. It also allows an application to gain exclusive access to an authoring or media-mastering device by tearing down the upper layers of the stack and requiring the in-kernel logical unit driver to yield control to the application. For more information about this process, see *SCSI Architecture Model Device Interface Guide*.

For serial and storage devices you can access through device files, your application may fail to open the device for the following reasons:

- Another process has already opened the device.
- The file system itself has opened a storage device.
- Your application does not have the correct file-system permissions to access the device.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

I/O Kit Family Device-Access Support

This appendix lists the current I/O Kit device families, identifies which families support device interfaces and which can be accessed through device files and POSIX functions, and points to documentation for working with specific devices.

For more information on I/O Kit families, see *I/O Kit Fundamentals*. For reference documentation of I/O Kit family support of user-space device access, see the Device Drivers Reference Library.

Documentation for working with additional device families will be provided as it becomes available.

- **ADB family.** This family provides an interface (defined in `IOADBLib.h`) for reading and writing registers on ADB devices. The interface permits only polled mode operations. Interrupt operations are only supported for kernel-resident clients.
- **ATA and ATAPI family.** This family does not supply any device interfaces. Access to ATA/ATAPI devices is provided by clients of this family, most commonly the Storage family.

  The SCSI Architecture Model family provides device interfaces for ATAPI devices that comply with the SCSI Architecture Model SCSI Primary Commands specification. See*SCSI Architecture Model Device Interface Guide* for more information.
- **Audio family.** The Audio family does not export device interfaces for applications to access audio hardware directly. However, it does provide a device interface that the Audio Hardware Abstraction Layer (Audio HAL) uses to access drivers derived from the Audio family.
- **FireWire family.** The FireWire family provides a general-purpose device interface that is suitable for any FireWire device except those that require an in-kernel driver, such as disk drivers that mount file systems. However, the device interfaces provided by the FireWire DV and FireWire SBP-2 families (described next) provide mechanisms that may be more convenient for working with some devices.

  For the latest information on Apple's FireWire support, including access to development kits with device interface support, see http://developer.apple.com/hardwaredrivers/firewire/.

  For more information about accessing FireWire devices (including AV/C and SBP-2 units) from an application, see *FireWire Device Interface Guide*.
- **FireWire DV family.** The FireWire DV (digital video) family provides complete driver support so that you can use standard QuickTime functions to access DV devices such as cameras and camcorders. No specific knowledge of FireWire is required and you can use the same functions for Mac OS X and MacOS8and9. Additional documentation for DV support of lower-level device control may be provided at a later date.
- **FireWire SBP-2 family.** This family provides a device interface to communicate with devices that support SBP-2 (Serial Bus Protocol 2).

  The SCSI Architecture Model family provides device interfaces for FireWire SBP-2 devices that comply with the SCSI Architecture Model SCSI Primary Commands specification. See*SCSI Architecture Model Device Interface Guide* for more information.
- **Graphics family.** The Graphics family includes Quartz 2D and other graphics libraries that provide high-level graphics-rendering services.

  For information on Quartz 2D and Quartz Compositor (the Mac OS X window server), see "Hardware-Access Options" (page 9).

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

- **HID family.** Through the HID Manager, the HID family provides a device interface for accessing a variety of devices, including joysticks and other game devices, audio devices, non-Apple displays, and UPS (uninterruptible power supply) devices. Note that you can use the Carbon Event Manager and NSEvent interfaces described in "Hardware-Access Options" (page 9) to monitor mouse and keyboard actions.

  For more information on using the HID Manager to access HID class devices, see *HID Class Device Interface Guide*. You can also find sample code for using the Mac OS X HID Manager in the Games Human Interface Device & Force Feedback Sample Code Library.

- **Network family.** Mac OS X provides networking functions in Carbon and Cocoa that should handle most standard networking requirements for applications. You can also use the BSD sockets API to obtain network services. A recommended network programming book is *Unix Network Programming, Volume 1, Second Edition*, by W. Richard Stevens, Prentice-Hall PTR, 1998.

- **PC Card family.** There are no direct device interfaces for either CardBus or 16-bit PC Card devices. Direct access to PC Card bus hardware by applications or other code running outside the kernel is not permitted for security reasons. Applications that must communicate with a PC Card must do so through an in-kernel driver.

- **PCI and AGP family.** There are no device interfaces for PCI and AGP devices. Direct access to PCI bus hardware by applications or other code running outside the kernel is not permitted for security reasons. Applications that must communicate with a PCI or AGP card must do so through an in-kernel driver.

  In most cases, applications should interact with higher-level services, such as those provided by the USB storage family or other client families. Applications can access graphics devices through the Quartz Compositor, which is described briefly in "Hardware-Access Options" (page 9), or other high-level APIs.

- **SCSI family.** In versions of Mac OS X prior to 10.2, the SCSI family supported user-space access to parallel SCSI devices through the device-interface functions in the `IOSCSILib.h` (in the `cdb` directory of the I/O Kit framework). Your application may still be able to use these functions to find and communicate with parallel SCSI devices. However, if you're looking up a parallel SCSI device that is not accessible with the SCSI Architecture Model family's device interfaces and if your application requires compatibility with versions of Mac OS X prior to 10.2, you should employ the device look-up functions of both the new SCSI Parallel family and the old SCSI family. This is because after a user has installed new HBA (host bus adaptor) drivers developed with the new SCSI Parallel family, the device-interface functions of the SCSI family will no longer be supported. By using the functions of both families to look for the device, however, your application has the widest compatibility.

  See *SCSI Architecture Model Device Interface Guide* for more information.

- **SCSI Architecture Model family.** Devices that support the SCSI Architecture Model SCSI Primary Commands specification can be controlled by SCSI tasks, which are a means of executing command descriptor block (CDB) commands. The SCSI Architecture Model family provides device interfaces for accessing compliant ATAPI, USB mass storage, FireWire SBP-2, and, in some cases, parallel SCSI devices.

  See *SCSI Architecture Model Device Interface Guide* for more information.

- **SCSI Parallel family.** The SCSI Parallel family is new in Mac OS X version 10.2 and is designed to support SCSI controllers. If a user has installed new HBA (host bus adapter) drivers developed with the SCSI Parallel family, your application can access all SCSI devices that do not declare a peripheral device type of $00, $05, $07, or $0E using the device-interface functions of the SCSI Architecture Model family.

  See *SCSI Architecture Model Device Interface Guide* for more information.

- **Serial family.** Applications can access devices in this family through the device-file mechanism. You use the I/O Kit to obtain a path to the device files in the `/dev` directory. The filenames start with `cu` or `tty` , such as `cu.modem`, `tty.modem`, `ttyp1`, `ttyp2`, and so on, so that a full device-file name would look

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

like `/dev/cu.modem`. You then perform traditional UNIX serial port access using POSIX `termios` functions. Your application can read and write data using these device files. Data is also routed through to PPP via these device files. For related information, see "Inside the Device-File Mechanism" (page 21).

For information on how to use the device-file mechanism to access a serial device from an application, see *Device File Access Guide for Serial Devices*.

- **Storage family.** Mac OS X provides file-management APIs in Carbon and Cocoa that allow applications to access files and folders on physical storage devices. Applications can also get raw access to media objects in this family through the device file system. You use the I/O Kit to obtain a path to device files in the `/dev` directory, then use traditional UNIX file-system access through POSIX functions. For related information, see "Inside the Device-File Mechanism" (page 21).

  For information on how to use the device-file mechanism to access storage media from an application, see *Device File Access Guide for Storage Devices*.

- **USB family.** This family provides device interface support for generic Universal Serial Bus (USB) serial devices. Support for USB input devices is provided by the HID family. For more information on accessing USB devices from user space, see *USB Device Interface Guide*.

  The SCSI Architecture Model family provides device interface support for USB mass storage class devices that comply with the SCSI Architecture Model SCSI Primary Commands specification. See *SCSI Architecture Model Device Interface Guide* for more information.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Document Revision History

This table describes the changes to *Accessing Hardware From Applications*.

Date	Notes
2007-02-08	Made minor corrections.
2006-11-07	Fixed minor errors.
2006-05-23	Made minor corrections and added information about the use of the NVRAM variable pmuflags while debugging.
2005-11-09	Fixed minor typos.
2005-09-08	Made minor bug fixes.
2005-07-07	Made minor bug fixes.
2005-06-04	Added CFNetwork API and NSEvent support for tablet events to list of APIs for hardware access.
2005-04-08	Fixed typos. Added note that Objective-C does not supply I/O Kit interfaces. Added link to man page documentation.
2005-01-11	Reworded bullet point in Appendix A.
2004-04-22	Updated documentation references.
2003-10-10	Added information about getting power-source information in an application.
2003-05-15	Updated for WWDC 2003. Changed emphasis from API “cookbook” to developer guide.
	Removed “Working With Parallel SCSI Devices” chapter. This information can be found in*SCSI Architecture Model Device Interface Guide*.
	Removed “Working With SCSI Architecture Model Devices” chapter. This information can be found in *SCSI Architecture Model Device Interface Guide*.
	Removed “Working With Serial I/O” chapter. This information can be found in *Device File Access Guide for Serial Devices*.
	Removed “Working With Device Files for Storage Devices” chapter. This information can be found in *Device File Access Guide for Storage Devices*.

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Document Revision History

© 2001, 2007 Apple Inc. All Rights Reserved.

Glossary

device Computer hardware, typically excluding the CPU and system memory, that can be controlled and can send or receive data. Examples of devices include monitors, drives, bus controllers, and keyboards.

device file A special file the I/O Kit creates in the `/dev` directory for each serial and storage device it discovers.

device interface A plug-in interface, provided by an I/O Kit family, that conforms to the CFPlugIn architecture. Code running on Mac OS X can call the functions in a device interface to access the in-kernel object representing a device. A device interface transmits an application's commands to the device object via a user client. See also user client (page 68).

device matching The I/O Kit process of searching the I/O Registry for objects representing one or more specific kinds of device.

driver A unit of software that manages a specific piece of hardware. A driver written with the I/O Kit is an object that implements the appropriate I/O Kit abstractions for controlling that hardware.A driver can serve as a nub, but this is rare. See also nub (page 67).

driver matching The I/O Kit process of locating a suitable driver for a device.

driver personality A dictionary of key/value pairs that specify device property values, such as family type, vendor name, or product name. A driver is suitable for any device whose properties match one of the driver's personalities.

family A collection of software abstractions that are common to all devices of a particular category. Families provide functionality and services to drivers. The I/O Kit defines families for bus protocols (such as SCSI, USB, and FireWire), storage devices, human interface devices, and many others.

framework A type of bundle that packages a dynamic shared library with the resources the library requires, including header files and reference documentation.

I/O Kit An object-oriented framework for developing device drivers on Mac OS X. The I/O Kit provides many features, from a set of object classes that model system software and streamline the task of writing device drivers, to a dynamic model for identifying, loading, and unloading drivers and other services in a running system.

I/O Kit framework The framework that includes `IOKitLib.h` and makes the I/O Registry, user client plug-ins, and other I/O Kit services available to applications and other code. Stored on disk as `IOKit.framework`.

I/O Registry A dynamic database that describes a collection of "live" objects, each of which represents an I/O Kit entity, such as a family, driver, or nub. As hardware is added to or removed from the system, the I/O Registry is modified to reflect the changes.

kernel space The protected memory partition in which the kernel resides. See also user space (page 68).

matching dictionary A dictionary of key/value pairs that describe the properties of a device or other service. The values in a matching dictionary are compared against those in a driver personality during device matching.

nub An I/O Kit object that represents a detected, controllable entity, such as a device or logical service. A nub may represent a bus controller, a disk, a

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

graphics adaptor, or any number of similar entities. When it supports a specific piece of hardware, a nub is also a driver (although this is rare).A nub supports dynamic configuration by providing a connection match point between two drivers (and, by extension, between two families). A nub can also provide services to code running in user space through a device interface. See also driver (page 67).

plug-in An object module that can be dynamically added to a running system or application.Core Foundation Plug-in Services uses the basic code-loading facility of Core Foundation Bundle Services to provide a standard plug-in architecture, known as the Core Foundation plug-in model, for Mac OS X applications.

SCSI See Small Computer System Interface (SCSI) (page 68).

SCSI Architecture Model A specification, approved as ANSI standard X3.270-1996, that defines a common interface standard between computers and devices such as disk drives, printers, and scanners.

Small Computer System Interface (SCSI) An industry standard parallel data bus that provides a consistent method of connecting computers and peripheral devices.

service A service is an I/O Kit entity, based on a subclass of IOService, that has been published with the `registerService` method and provides certain capabilities to other I/O Kit objects. In the I/O Kit's layered architecture, each layer is a client of the layer below it and a provider of services to the layer above it.

user client An in-kernel object that inherits from the IOService class and provides a connection between an in-kernel device driver or device nub and an application or process in user space. See also device interface (page 67).

user space Memory outside the protected partition in which the kernel resides. Applications, plug-ins, and other types of modules typically run in user space. See also kernel space (page 67).

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

Index

A

ADB family 61
ATA/ATAPI families 61
Audio family 61
Audio HAL (Hardware Abstraction Layer) 10

B

BSD
- and device files 21–23
- and networking API 22
- file systems 9
- networking 9

`busyState` values 51–52

C

Carbon Event Manager 9
Carbon Printing Manager 9
`CFDictionaryAddValue` function 32
`CFDictionaryCreateMutable` function 32
CFNetwork 10
`CFRelease` function 32
`CFRunLoopAddSource` function 35
`CFRunLoopRun` function 36
`CFStringGetCString` function 39
Component Object Model (COM) 19
Core Audio 10
Core Foundation plug-in model 19
Core Graphics Services. *See* Quartz Compositor
custom device interfaces. *See* developing custom device-access solutions
custom user clients. *See* developing custom device-access solutions

D

developing custom device-access solutions 52–55
device families. *See* I/O Kit, family device-access support
device files
- and device interfaces 21–22
- defined 15
- getting a device-file path for 38–39
- in driver stack 18
- overview 21–23

device interfaces
- creating custom. *See* developing custom device-access solutions
- and device files 21–22
- overview 19–21
- steps for getting 37–38

device interfaces
- defined 14
- in driver stack 17

device look-up functions 34–35
device matching
- and communicating with the I/O Kit 26–27
- and device files 33
- and notifications 35–36
- creating matching dictionaries for 31–33
- defined 15
- device look-up functions for 34–35
- examining matching objects in 36
- getting keys and values for 27–31
- steps to implement 26

driver 14
driver matching 15
driver personalities
- defined 14
- examining for matching keys and values 30–31
- keys in 27–30

E

error return values
- interpreting 57–59

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

macros for handling 58
exclusive-access errors 59–60

F

family 13
File Manager 10
file systems, BSD 9
finding devices. *See* device matching
FireWire DV family 61
FireWire family 61
FireWire SBP-2 family 61
functions
CFDictionaryAddValue 32
CFDictionaryCreateMutable 32
CFRelease 32
CFRetain 34
CFRunLoopAddSource 35
CFRunLoopRun 36
CFStringGetCString 39
close 22
host_get_io_master 23
IOBSDNameMatching 31, 44
IOConnectAddClient 55
IOConnectAddRef 55
IOConnectGetService 55
IOConnectMapMemory 54
IOConnectMethodScalarIScalarO 53
IOConnectMethodScalarIStructureI 53
IOConnectMethodScalarIStructureO 53
IOConnectMethodStructureIStructureO 54
IOConnectRelease 55
IOConnectSetCFProperties 51
IOConnectSetCFProperty 51
IOConnectSetNotificationPort 55
IOConnectUnmapMemory 54
IOCreatePlugInInterfaceForService 20, 37
IODestroyPlugInInterface 20, 38
IODispatchCalloutFromMessage 46
IOIterator 35
IOIteratorIsValid 36, 46
IOIteratorNext 36, 46
IOIteratorReset 36, 46
IOKitGetBusyState 51
IOKitWaitQuiet 51
IOMasterPort 23, 26, 27
IONotificationPortCreate 35, 46
IONotificationPortDestroy 46
IONotificationPortGetMachPort 46
IONotificationPortGetRunLoopSource 35, 46
IOObjectConformsTo 43
IOObjectGetClass 36, 43
IOObjectGetRetainCount 42
IOObjectIsEqualTo 43
IOObjectRelease 42
IOObjectRetain 42
IORegistryEntryCreateCFProperties 50
IORegistryEntryCreateCFProperty 36, 39, 50
IORegistryEntryGetLocationInPlane 48
IORegistryEntryGetName 48
IORegistryEntryGetNameInPlane 48
IORegistryEntryGetParentEntry 35
IORegistryEntryGetPath 48
IORegistryEntryInPlane 48
IORegistryEntrySearchCFProperty 50
IORegistryEntrySetCFProperties 51
IORegistryEntrySetCFProperty 51
IOServiceAddMatchingNotification 34–35, 45
IOServiceClose 53
IOServiceGetBusyState 51
IOServiceGetMatchingService 34–35, 45
IOServiceGetMatchingServices 27, 34–35, 45
IOServiceMatching 31, 33, 44
IOServiceNameMatching 31, 44
IOServiceOpen 52
IOServiceWaitQuiet 51
mach_port_deallocate 27
mach_port_get_refs 43
open 22
QueryInterface 20, 38
SCSIAction 11

H

HID family 62
host_get_io_master function 23

I

I/O Kit
and driver-stack building 16–19
communicating with 23
family device-access support 61–63
framework 15
terms and definitions 13–15
I/O Registry Explorer application 14, 30
I/O Registry
defined 14
object property-setting functions 50–51
object property-viewing functions 49–50
object-introspection functions 48–49
planes in 47

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

traversal functions 47–48
Info.plist file 14, 30
IOBSDNameMatching function 31, 44
IOCFPlugInInterface plug-in 20
IOConnectAddClient function 55
IOConnectAddRef function 55
IOConnectGetService function 55
IOConnectMapMemory function 54
IOConnectMethodScalarIScalarO function 53
IOConnectMethodScalarIStructureI function 53
IOConnectMethodScalarIStructureO function 53
IOConnectMethodStructureIStructureO function 54
IOConnectRelease function 55
IOConnectSetCFProperties function 51
IOConnectSetCFProperty function 51
IOConnectSetNotificationPort function 55
IOConnectUnmapMemory function 54
IOCreatePlugInInterfaceForService function 20, 37
IODestroyPlugInInterface function 20, 38
IODispatchCalloutFromMessage function 46
IOIterator objects
and the IOIteratorNext function 36
from device look-up functions 34
reference counting of 34
using to examine matching objects 36
IOIteratorIsValid function 36, 46
IOIteratorNext function 36, 46
IOIteratorReset function 36, 46
IOKitGetBusyState function 51
IOKitLib API
and in-kernel objects 41–42
busy-state functions in 51–52
custom device interface development functions in 52–55
device look-up functions in 45
device-notification functions in 45
I/O Registry object information functions in 48–49
I/O Registry-traversal functions in 47–48
iteration functions in 46
matching dictionary-creation functions in 44
object-introspection functions in 43
property-setting functions in 50–51
property-viewing functions in 49–50
reference-counting functions in 42–43
IOKitWaitQuiet function 51
IOMasterPort function 23, 26
IONotificationPortCreate function 35, 46
IONotificationPortDestroy function 46
IONotificationPortGetMachPort function 46
IONotificationPortGetRunLoopSource function 35, 46
IOObjectConformsTo function 43
IOObjectGetClass function 36, 43
IOObjectGetRetainCount function 42
IOObjectIsEqualTo function 43
IOObjectRelease function 42
IOObjectRetain function 42
IOProviderClass key 28
ioreg tool 14, 30
IORegistryEntryCreateCFProperties function 50
IORegistryEntryCreateCFProperty function 36, 39, 50
IORegistryEntryGetLocationInPlane function 48
IORegistryEntryGetName function 48
IORegistryEntryGetNameInPlane function 48
IORegistryEntryGetParentEntry function 35
IORegistryEntryGetPath function 48
IORegistryEntryInPlane function 48
IORegistryEntrySearchCFProperty function 50
IORegistryEntrySetCFProperties function 51
IORegistryEntrySetCFProperty function 51
IOServiceAddMatchingNotification function 34–35, 45
IOServiceClose function 53
IOServiceGetBusyState function 51
IOServiceGetMatchingService function 34–35, 45
IOServiceGetMatchingServices function 27, 34–35, 45
IOServiceMatching function 31, 33, 44
IOServiceNameMatching function 31, 44
IOServiceOpen function 52
IOServiceWaitQuiet function 51
io_iterator_t object 42
io_object_t type 36, 41–42
io_registry_entry object 42
io_service_t object 42
iterators. *See* IOIterator objects

K

kernel space 13
kIOMasterPortDefault constant 23
kIOReturnExclusiveAccess error 59
kIOReturnNotAttached error 59

L

looking up devices. *See* device look-up functions

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.

M

Mach ports, using for I/O Kit communication 23, 41–42
mach_port_get_refs function 43
matching dictionaries
  creating and modifying 31–33
  defined 15
  for device files 33
  getting property keys for 27–31
  reference counting of 34
  using Core Foundation functions to modify 32–33
  using I/O Kit functions to create 31–32

N

Network family 62
networking services in user space 9
notifications of device arrival and departure. *See* device matching, getting notifications
NSEvent object 10
nub 14

O

Open Transport 10

P

PC Card family 62
PCI and AGP families 62
personality dictionaries, examining for keys and values 30–31
POSIX (Portable Operating System Interface) API 15, 21, 22, 39
power-source information, getting from user space 10
property keys
  finding with I/O Registry Explorer 30
  in header files 28–30
  in information property lists 30
  IOProviderClass key 28

Q

Quartz 2D 10
Quartz Compositor 10
QueryInterface function 20, 38
QuickDraw 10
QuickTime 11

R

reference counting
  functions in IOKitLib 42–43
  of io_object_t objects 43
  of IOCFPlugInInterface objects 20
  of IOIterator objects 34
  of matching dictionaries 32, 34

S

SCSI Architecture Model family 62
SCSI family 62
SCSI Manager 4.3 11
SCSI Parallel family 11, 62
SCSIAction function 11
serial devices, setting up a matching dictionary for 33
Serial family 62
service 14
storage devices, setting up a matching dictionary for 33
Storage family 63

U

UPS (uninterruptible power supply) devices 10
USB family 63
user clients
  creating custom. *See* developing custom device-access solutions
  defined 14
  in driver stack 17
user space 13
UUID (universal unique identifier) 20

X

XML (Extensible Markup Language) 14

2007-02-08 | © 2001, 2007 Apple Inc. All Rights Reserved.