I/O Kit Fundamentals

2006-10-03

Contents

Figures, Tables, and Listings

Introduction to I/O Kit Fundamentals

This document explains the terminology, concepts, architecture, and basic mechanisms of the I/O Kit, Apple's object-oriented framework for developing device drivers for Mac OS X. It contains essential background information for anyone wanting to create device drivers for this platform.

Who Should Read This Document?

There are two general types of I/O Kit developers, and this document tries to be useful to both. The first type is the developer creating a device driver that is to be resident in the kernel; the second type is the application developer who is using an I/O Kit device interface to communicate with hardware.Some chapters contain information useful to both types of developers, and others contain information that is of interest only to writers of kernel-resident drivers.

Obviously there are things *I/O Kit Fundamentals* does not cover. It does not, for example, describe the use of the development tools or the use of specific driver APIs. But it does help you to understand the hows and whys of the I/O Kit, enabling you to obtain the most value from the more specific documentation and examples.

Organization of This Document

I/O Kit Fundamentals gives a broad, conceptual description of the I/O Kit and device-driver development on Mac OS X. It contains the following chapters:

- "What Is the I/O Kit?" (page 15)

  Describes the features and benefits of the I/O Kit, and also discusses the philosophy and decisions informing its design.

- "Architectural Overview" (page 25)

  Gives a high-level description of the I/O Kit's architecture, essential concepts, and basic mechanisms.

- "The I/O Registry" (page 39)

  Describes the I/O Registry, a dynamic database capturing the client/provider relationships among active driver objects.

- "Driver and Device Matching" (page 43)

  Explains the matching process by which the most appropriate client drivers are found for registered providers. It also summarizes the procedure processes in user space follow to find suitable devices and their drivers.

- "The Base Classes" (page 51)

  Describes the base classes that each driver object directly or indirectly inherits from. It includes discussions of object construction and disposal, driver objects as I/O Registry entries, and the driver life cycle.

- "Handling Events" (page 73)

  Explains the architecture and usage of work loops and event sources, mechanisms that the I/O Kit uses to process events such as interrupts and I/O requests in a protected single-threaded environment.

- "Managing Data" (page 87)

  Describes how to use memory cursors, memory descriptors, and related objects to handle I/O transfers. It also discusses how drivers should deal with hardware constraints, such as those imposed by DMA engines.

- "Managing Power and Device Removal" (page 97)

  Discusses the general strategy driver writers should follow in anticipation of unscheduled asynchronous events. Then the chapter explains the concepts and programming procedures related to two major types of such events: power management and responding to device removal (hot-swapping).

- "I/O Kit Family Reference" (page 117)

  Provides a class hierarchy chart for each family.

- "Base and Helper Class Hierarchy" (page 145)

  Provides a class hierarchy chart for all I/O Kit classes that are not members of a specific family.

- "Document Revision History" (page 147)

  Lists changes to this document.

- "Bibliography" (page 149)

  Lists additional sources for information on Mac OS X and related topics.

- "Glossary" (page 151)

  Defines key terms used in this document.

See Also

Once you've absorbed the information in *I/O Kit Fundamentals*, you should be able to forge ahead and actually create a device driver. Apple provides several documents and other sources of information to help you with your efforts:

- *I/O Kit Device Driver Design Guidelines* describes the general steps required to design, code, debug, and build a device driver that will be resident in the kernel.

- *Accessing Hardware From Applications* discusses how to use the I/O Kit's "device interface" feature; it also includes information on serial and storage I/O via BSD device files.
- *Kernel Extension Concepts* contains a collection of tutorials that introduce you to the development tools and take you through the steps required to create, debug, and package kernel extensions and I/O Kit drivers (a type of kernel extension). It also includes information on other aspects of kernel extensions.
- Documentation that provides in-depth information on writing drivers for specific driver families and related reference documentation is available in Hardware & Drivers Documentation.
- *Kernel Programming Guide* provides an overview of the architecture and components of the Mac OS X kernel environment (Mach, BSD, networking, file systems, I/O Kit). All developers who intend to program in the kernel (including device-driver writers) should read this document.
- *Mac OS X Technology Overview* provides an introduction to Mac OS X as a whole, which is useful for developers new to the platform.

Of course, you can always browse the header files shipped with the I/O Kit, which are installed in `Kernel.framework/Headers/iokit` (kernel-resident) and `IOKit.framework/Headers` (device interface).)

You can also view developer documentation in Xcode. To do this, select Help from the Xcode menu and then click Show Documentation Window.

You can browse the BSD man pages for more information on BSD and POSIX APIs in two ways: You can type `man`*function_name* in a Terminal window (for example, `man gdb`) or you can view an HTML version at Mac OS X Man Pages.

If you're ready to develop a universal binary version of a device driver to run in an Intel-based Macintosh, first read *Universal Binary Programming Guidelines, Second Edition*. Then, see *I/O Kit Device Driver Design Guidelines* for an overview of issues of particular interest to device driver developers. Related information that is specific to a particular device type is available in the documents listed at Hardware & Drivers Documentation.

Apple maintains several websites where developers can go for general and technical information on Mac OS X.

- Apple Developer Connection Reference Library (http://developer.apple.com/referencelibrary/index.html) contains a comprehensive collection of technical resources, including documentation, sample code, and Technical Notes.
- Apple Developer Connection: Mac OS X (http://developer.apple.com/macosx) offers SDKs, release notes, product notes and news, and other resources and information related to Mac OS X.
- The AppleCare Support site (http://www.apple.com/support) provides a search feature that enables you to locate technical articles, manuals, specifications, and discussions on Mac OS X and other areas.

What Is the I/O Kit?

The I/O Kit is a collection of system frameworks, libraries, tools, and other resources for creating device drivers in Mac OS X. It is based on an object-oriented programming model implemented in a restricted form of C++ that omits features unsuitable for use within a multithreaded kernel. By modeling the hardware connected to a Mac OS X system and abstracting common functionality for devices in particular categories, the I/O Kit streamlines the process of device-driver development.

This chapter talks about the inherent capabilities of the I/O Kit (and of the drivers developed with it), about the decisions informing its design, and about the I/O Kit when considered as a product. It also offers some caveats and guidelines for those considering developing kernel software such as device drivers.

Before You Begin

You might have developed device drivers for other platforms—Mac OS 9, perhaps, or BSD or another flavor of UNIX. One thing you'll discover reading this document is how different the approach is with the I/O Kit. Although writing drivers for Mac OS X requires new ways of thinking and different ways of programming, you are amply rewarded for shifting to this new approach. The I/O Kit simplifies driver development and supports many categories of devices. Once you get the basics of the I/O Kit down, you'll find it a relatively easy and efficient matter to create device drivers.

Before you attempt driver development with the I/O Kit, Apple highly recommends certain prerequisites. Because the framework uses an object-oriented programming model, which is implemented in a restricted subset of C++, it helps to know C++ or object-oriented concepts in general. Also, device drivers are not the same thing as applications because, being kernel-resident, they must abide by more restrictive rules. Knowledge of kernel programming is therefore very useful.

Indeed, programming in the kernel is discouraged except when it is absolutely necessary. Many alternatives for communicating with hardware and networks exist at higher levels of the system, including the "device interface" feature of the I/O Kit described in "Controlling Devices From Outside the Kernel" (page 36). See "Should You Program in the Kernel?" (page 22) for more on alternatives to kernel programming.

I/O Kit Features

From its inception, the fundamental goal for the I/O Kit has been to accommodate and augment native features and capabilities of Mac OS X, particularly those of the kernel environment. As the driver model for Mac OS X, the I/O Kit supports the following features:

- Dynamic and automatic device configuration (plug-and-play)
- Many new types of devices, including graphics acceleration and multimedia devices
- Power management (for example, "sleep" mode)
- The kernel's enforcement of protected memory—separate address spaces for kernel and user programs
- Preemptive multitasking
- Symmetric multiprocessing
- Common abstractions shared between types of devices
- Enhanced development experience—new drivers should be easy to write

The I/O Kit supports these kernel features with its new model for device drivers and adds some additional features:

- An object-oriented framework implementing common behavior shared among all drivers and types (families) of drivers
- Many families for developers to build upon
- Threading, communication, and data-management primitives for dealing with issues related to multiprocessing, task control, and I/O-transfers
- A robust, efficient match-and-load mechanism that scales well to all bus types
- The I/O Registry, a database that tracks instantiated objects (such as driver instances) and provides information about them
- The I/O Catalog, a database of all I/O Kit classes available on a system
- A set of device interfaces—plug-in mechanisms that allows applications and other software in "user space" to communicate with drivers
- Excellent overall performance
- Support for arbitrarily complex layering of client and provider objects

The I/O Kit's object-oriented programming model is implemented in a restricted subset of C++. Object-orientation just in itself is an advantage in driver development because of the code reusability it fosters. Once you are familiar with the I/O Kit, you can write device drivers much more quickly and efficiently than you can using a procedural model. In addition, code reusability decreases the memory footprint of drivers; drivers ported from Mac OS 9, for example, have been up to 75% smaller in Mac OS X.

Design Principles of the I/O Kit

Mac OS X is largely the product of two strains of operating-system technology: Mac OS 9 (and its predecessors) and BSD. Given this pedigree, one might have expected Apple to adopt the device-driver model of Mac OS 9 or FreeBSD. Instead, Apple chose to redesign the model. Several reasons motivated this decision.

First, neither the Mac OS 9 driver model nor the FreeBSD driver model offers a set of features rich enough to meet the needs of Mac OS X. The Mac OS X kernel is significantly more advanced than its Mac OS precursors; it handles memory protection, preemptive multitasking, multiprocessing, and other features not present in previous versions of Mac OS. Although FreeBSD is capable of handling these features, the BSD model does not offer other features expected in a modern operating system, including automatic configuration, driver stacking, power management, and dynamic loading of devices.

Thus the primary motivation behind the I/O Kit was the inadequacy of the currently available driver models. The redesign of the I/O architecture had to take advantage of and support the operating-system features of Mac OS X. Toward this end, the I/O Kit's designers settled on object-oriented programming model that abstracted the kernel capabilities and hardware of a Mac OS X system and provided a view of this abstraction to the upper layers of the operating system. The compelling part of this abstraction is the implementation of behavior common to all device drivers (or types of device drivers) in the classes of the I/O Kit.

As an example, consider virtual memory. In Mac OS 9, virtual memory is not a fundamental part of the operating system; it is an option. Because of this, a developer must always take virtual memory into account when creating a driver, and this raises certain complications. In contrast, virtual memory is an inherent capability of Mac OS X and cannot be turned off. Because virtual memory is a fundamental and assumed capability, knowledge of it is incorporated into system software and driver writers do not have to take it into account.

The I/O Kit functions as a kind of foundation and coordinator for device drivers. This is a departure from previous driver models. In Mac OS 9, all software development kits (SDKs) are independent of each other and duplicate common functionality. Mac OS X delivers the I/O Kit as part of a single kernel development kit (KDK); all portions of the KDK rest on common underpinnings. Mac OS X helps developers take advantage of hardware complexity without requiring them to encode software complexity into each new device driver. In most cases, they need only add the specific code that makes their drivers different.

Another part of the design philosophy of the I/O Kit is to make the design completely open. Rather than hiding APIs in an attempt to protect developers from themselves, all I/O Kit source code is available as part of Darwin. Developers can use the source code as an aid to designing (and debugging) new drivers.

Limitations of the I/O Kit

Although the I/O Kit supports most types of hardware on a Mac OS X system, it does not fully support all hardware. One category of such devices are those used for imaging, among them printers, scanners, and digital cameras. The I/O Kit provides only limited support for these devices, handling communication with these devices through the FireWire and USB families. Applications or other programs in user space are responsible for controlling the particular characteristics of these devices (see "Controlling Devices From Outside the Kernel" (page 36) for details). If your application needs to drive an imaging device, you should use the appropriate imaging software development kit (SDK).

Although the I/O Kit attempts to represent the hierarchy and dynamic relationships among hardware devices and services in a Mac OS X system, some things are difficult to abstract. It is in these gray areas of abstraction, such as when layering violations occur, that driver writers are more on their own. Even when the I/O Kit representation is clean and accurate, the reusability of I/O Kit family code can be limited. All hardware can have its own quirks and a driver's code must take these quirks into account.

Language Choice

Apple considered several programming languages for the I/O Kit and chose a restricted subset of C++.

C++ was chosen for several reasons. The C++ compiler is mature and the language provides support for system programming. In addition, there is already a large community of Macintosh (and BSD) developers with C++ experience.

The restricted subset disallows certain features of C++, including

- Exceptions
- Multiple inheritance
- Templates
- Runtime type information (RTTI)—the I/O Kit uses its own implementation of a runtime typing system

These features were dropped because they were deemed unsuitable for use within a multithreaded kernel. If you feel you need these features, you should reconsider your design. You should be able to write any driver you require using I/O Kit with these restrictions in place.

Using Namespaces in an I/O Kit Driver

Note that you can use namespaces in your I/O Kit driver. The use of namespaces can help you avoid name collisions and may make your code easier to read and more maintainable. Be sure to use reverse-DNS format for the namespace name (for example, `com.mycompany`) to avoid potential namespace collisions.

If you decide to use namespaces in your in-kernel I/O Kit driver, do not declare any subclass of OSObject in a namespace or your driver will not load. At present, the loader does not support OSObject-derived classes that require qualification, such as the one shown below:

```
namespace com.mycompany {
    class com.mycompany.driver.myClass : public IOService { // This is not
allowed.
        OSDeclareDefaultStructors (com.mycompany.driver.myClass);
    };
};
```

Using Static Constructors in an I/O Kit Driver

In Mac OS X v10.4, GCC 4.0 is the default compiler for all new projects, including I/O Kit drivers. This section describes a particular difference between GCC 3.3 and GCC 4.0 that may affect the compatibility of your in-kernel driver between Mac OS X v10.3.x and Mac OS X v10.4.x. For more information on the differences between GCC 3.3 (the default compiler in Mac OS X v10.3) and GCC 4.0, including porting guidance, see *GCC Porting Guide*.

If you perform static construction within a function in a C++ I/O Kit driver (or other KEXT) compiled with GCC 3.3 or earlier, be aware that the same KEXT compiled with GCC 4.0 will no longer load successfully. This is because GCC 4.0 is more strict about taking and releasing locks in the kernel environment. If you perform in-function static construction in your I/O Kit driver compiled with GCC 4.0, you will probably see the following error when you try to load it:

```
kld():Undefined symbols:
__cxa_guard_acquire
__cxa_guard_release
```

The solution to this problem is simple: move the static constructor to a global namespace. For example, suppose that your I/O Kit driver includes an in-function static construction, such as in the code shown below:

```
class com_mycompany_driver_mystaticclass;
void com_mycompany_driver_myclass::myfunction(void)
{
    static com_mycompany_driver_mystaticclass staticclass;
    staticclass.anotherfunction();
}
```

You can avoid loading errors by changing this code to avoid in-function static construction, as in the code shown below:

```
class com_mycompany_driver_mystaticclass;
static com_mycompany_driver_mystaticclasss staticclass;
void com_mycompany_driver_myclass::myfunction(void)
{
    staticclass.anotherfunction();
}
```

Note that you may be able to avoid the load errors associated with in-function static construction without changing your code if you compile your KEXT with GCC 4.0 using the `-fno-threadsafe-statics` compiler option, but this may lead to other problems. Specifically, unless you can guarantee thread safety in other ways, compiling your KEXT with this option may break your code.

The Parts of the I/O Kit

Physically and electronically, the I/O Kit is composed of many parts: frameworks and libraries, development and testing tools, and informational resources such as example projects, documentation, and header files. This section catalogs these parts and indicates where they are installed and how they can be accessed.

Frameworks and Libraries

The I/O Kit is based on three C++ libraries. All of them are packaged in frameworks, but only `IOKit.framework` is a true framework. The Kernel framework exists primarily to expose kernel header files, including those of libkern and IOKit. The code of these "libraries" is actually built into the kernel; however, drivers (when loaded) do link against the kernel as if it were a library.

Table 1-1 Frameworks and libraries of the I/O Kit

Framework or library	Description and location
Kernel/IOKit	The library used for developing kernel-resident device drivers. Headers location: `Kernel.framework/Headers/IOKit`
Kernel/libkern	The library containing classes useful for all development of kernel software. Headers location: `Kernel.framework/Headers/libkern`
IOKit	The framework used for developing device interfaces. Location: `IOKit.framework`

Applications and Tools

You use a handful of development applications to build, manage, debug, examine, and package device drivers. Table 1-2 (page 20) lists the applications used in driver development; these applications are installed in `/Developer/Applications`.

Table 1-2 Applications used in driver development

Application	Description
Xcode	The primary development application for Mac OS X. Xcode manages projects, provides a full-featured code editor, builds projects according to arbitrarily complex rules, provides a user interface for software configuration, and acts as a front end for debugging and documentation searches.
I/O Registry Explorer	Enables the graphical exploration of the contents and structure of the I/O Registry.
Package Maker	Creates an installation package for the Installer application; used for deployment of kernel extensions (including device drivers).

Table 1-3 (page 21) describes the command-line tools used in developing device drivers with the I/O Kit; all tools are located in `/usr/sbin/` or `/sbin`.

Note: You can view on-line documentation of these tools (called man pages in the UNIX world) by entering a command in the shell provided by the Terminal application. The command is `man`, and the main argument to the `man` command is the name of the tool for which you want to see documentation. For example, to see the man page for the `kextload` tool, enter the following line in Terminal:

```
man kextload
```

Table 1-3 Command-line tools used in driver development

Tool	Description and location
`ioreg`	Prints the contents of the I/O Registry (a command-line version of the I/O Registry Explorer application).
`kextload`	Loads a kernel extension (such as device driver) or generates a statically linked symbol file for remote debugging.
`kextunload`	Unloads a kernel extension (if possible).
`kextstat`	Prints statistics about currently loaded drivers and other kernel extensions.
`iostat`	Displays kernel I/O statistics on terminal, disk, and CPU operations.
`ioclasscount`	Displays instance count of a specified class.
`ioalloccount`	Displays some accounting of memory allocated by I/O Kit objects in the kernel.
`kextcache`	Compresses and archives kernel extensions (including drivers) so they can be automatically loaded into the kernel at boot time. See Loading Kernel Extensions at Boot Time.
`gcc`	Apple's version of the GNU C++ compiler; Xcode automatically invokes it with the correct set of flags for I/O Kit projects.
`gdb`	Apple's version of the GNU debugger; Xcode automatically invokes it with the correct set of flags for I/O Kit projects.

Other I/O Kit Resources

Several informational resources are included with the I/O Kit "product," particularly documentation and header files. Some of these resources are described in the preceding chapter, "Introduction to I/O Kit Fundamentals" (page 11).

The I/O Kit is part of the Darwin Open Source project. Apple maintains a website where you can find much information related to the I/O Kit and other Open Source projects managed by Apple. The following two locations are of particular interest:

- Open Source Projects—http://developer.apple.com/darwin/projects/

  Here you can find links to the Darwin and Darwin Streaming projects, among other projects. Also featured are links to documentation and tools.

- Mailing lists—http://developer.apple.com/darwin/mail.html

This page features links that will put you on the Darwin-Development and DarwinOS-Users mailing lists, among others.

Should You Program in the Kernel?

If you are thinking of writing code for the kernel environment, think carefully. Programming in the kernel can be a difficult and dangerous task. And often there is a way to accomplish what you want to do without touching the kernel.

Software that resides in the kernel tends to be expensive. Kernel code is “wired” into physical memory and thus cannot be paged out by the virtual memory system. As more code is put into the kernel, less physical memory is available to user-space processes. Consequently, paging activity will probably intensify, thereby degrading system performance.

Kernel code is also inherently destabilizing, much more so than application code. The kernel environment is a single process, and this means that there is no memory protection between your driver and anything else in the kernel. Access memory in the wrong place and the entire system can grind to a halt, a victim of a kernel panic.

Moreover, because kernel code usually provides services to numerous user-space clients, any inefficiencies in the code can be propagated to those clients, thereby affecting the system globally.

Finally, kernel software is a real pain to write. There are subtleties to grapple with that are unknown in the realm of application development. And bugs in kernel code are harder to find than in user-space software.

With all this in mind, the message is clear. It is in everyone's best interest to put as little code as possible into the kernel. And any code that ends up in the kernel should be honed and rigorously tested.

When Code Should Reside in the Kernel

A handful of situations warrant loading a driver or extension into the kernel environment:

- The software is used by the kernel environment itself.
- User-space programs will frequently use the software.
- The software needs to respond directly to primary interrupts (those delivered by the CPU's interrupt controller).

If the software you are writing does not match any of these criteria, it probably doesn’t belong in the kernel. If your software is a driver for a disk, a network controller, or a keyboard, it should reside in the kernel. If it is an extension to the file system, it should live in the kernel. If, on the other hand, it is used only now and then by a single user-space program, it should be loaded by the program and reside within it. Drivers for printers and scanners fall into this latter category.

Alternatives to Kernel-Resident Code

Apple provides a number of technologies that might let you accomplish what you want to do and stay out of the kernel. First are the higher-level APIs that give you some hardware-level access. For example, Open Transport is a powerful resource for many networking capabilities, and Quartz Compositor enables you to do some fairly low-level things with the graphics subsystem.

Second, and just as important, is the device-interface technology of the I/O Kit framework. Through a plug-in architecture, this technology makes it possible for your application to interact with the kernel to access hardware. In addition, you can—with a little help from the I/O Kit—use POSIX APIs to access serial, storage, or network devices. See "Controlling Devices From Outside the Kernel" (page 36) for a summary of device interfaces and see the document *Accessing Hardware From Applications* for a full discussion of this technology.

Note: Objective-C does not provide device-level I/O services. However, in your Cocoa application, you can call the C APIs for device-level functionality that the I/O Kit and BSD provide. Note that you can view the man pages that document BSD and POSIX functions and tools at Mac OS X Man Pages.

Architectural Overview

As you can with any complex system, you can look at the design of the I/O Kit from various angles and at different granularities. This chapter introduces you to the more important architectural elements and conceptual domains of the I/O Kit:

- Hardware modeling, the layering of driver objects, and the roles played by families, drivers, and nubs
- The runtime environment of device drivers
- The I/O Kit Registry and I/O Catalog
- Driver matching
- The I/O Kit class hierarchy
- Device interfaces

Keep in mind that this chapter is an overview, and so the discussion it devotes to each of these topics is intentionally brief. Later chapters cover most of these topics in more detail. In the case of device interfaces, the document *Accessing Hardware From Applications* describes the technology in great detail.

Driver Layering

Central to the design of the I/O Kit is a modular, layered runtime architecture that models the hardware of a Mac OS X system by capturing the dynamic relationships among the multiple pieces—hardware and software—involved in an I/O connection. The layers of the connection, comprising driver objects and the families these objects are members of, are stacked in provider-client relationships.

The chain of interconnected services or devices starts with a computer's logic board (and the driver that controls it) and, through a process of discovery and "matching," extends the connection with layers of driver objects controlling the system buses (PCI, USB, and so on) and the individual devices and services attached to these buses.

You can view the layering of driver objects in a running Mac OS X system using the I/O Registry Explorer application, included in the developer version of Mac OS X. The developer version also includes a command-line version of the application, `ioreg`, that you can run in a Terminal window to display current I/O Registry information.

This section examines the I/O Kit's layered architecture and describes the major constituent elements: families, drivers, and nubs.

Families and Drivers

An I/O Kit family is one or more C++ classes that implement software abstractions common to all devices of a particular type. The I/O Kit has families for bus protocols (such as SCSI Parallel, USB, and FireWire), for storage (disk) devices, for network services (including Ethernet), for human-interface devices (such as mice, keyboards, and joysticks), and for a host of other devices.

A driver becomes a member of a family through inheritance; the driver's class is almost always a subclass of some class in a family. By being a member of a family, the driver inherits the data structures (instance variables) and the behaviors that are common to all members of the family. For example, all SCSI controllers have certain things they must do, such as scanning the SCSI bus; the SCSI Parallel family defines and implements this scanning functionality. Thus, you do not need to include scanning code in your new SCSI controller driver (unless you require a different scanning implementation).

Most I/O Kit development involves writing specific driver classes, each of which inherits from the superclass in the family that provides the functionality the driver requires. A driver for an Ethernet controller, for example, inherits from the IOEthernetController superclass in the Network family. The bulk of a driver's interaction with its own family involves implementing member functions that the family invokes. These are typically client configuration and I/O requests. Some families also define objects and functions for your driver to use. The exact nature of these objects and functions depends on the family your driver works with.

However, a driver typically works with two families. In addition to the family a driver is a member of, the driver class must communicate with a nub published by the family for the bus or protocol the device is attached to. A nub (as "Drivers and Nubs" (page 26) explains in detail) is an object that defines an access point and communication channel for a given protocol. A family (usually representing a bus such as PCI or USB) acts as the driver's provider through the nub that it creates. The driver uses the nub to attach into the I/O Registry and communicate with its device. As an example, a PCI Ethernet driver would use an `IOPCIDevice` nub from the PCI family to attach to and communicate over the PCI bus. A driver's main interaction with the nub involves the issuing of requests or commands on whatever bus the nub is a client of. A SCSI device driver, for example, issues SCSI command blocks and checks results through the nub.

For more on families, particularly the nature and composition of superclasses in a family, see "The I/O Kit Family Classes" (page 35).

Drivers and Nubs

The I/O Kit supports two broad types of driver objects. The first is the nub, an object that defines an access point and communication channel, usually for a given protocol, such as PCI, USB, or Ethernet. The second type is the specific driver for an individual device or service. The specific driver communicates with the hardware, through a nub, to perform I/O operations. Both drivers and nubs in the I/O Kit must inherit from the IOService class.

A driver is an I/O Kit object that manages a specific piece of hardware. Drivers are written as kernel extensions and are usually installed in the Extensions folder (at `/System/Library/Extensions`.) See Kernel Extension Overview in *Kernel Programming Guide* for more information about creating and installing kernel extensions.

When a driver is selected for a device, but before it is loaded into the kernel (as a kernel extension), all required families—in terms of superclasses and their dependencies—are loaded to provide the common functionality for the driver and others of its type. (Of course, if these families have already been loaded, this step is not necessary.) After all requirements for the driver are met, the driver is loaded and instantiated as an object. See "The Anatomy of an I/O Connection" (page 27) for an illustration of this process.

A nub is an I/O Kit object that represents a communication channel for a device or logical service and mediates access to the device and service. For example, a nub could represent a bus, a disk, a disk partition, a graphics adaptor, or a keyboard. It might help to think of a nub as the software representation of a device slot or connector. Nubs also provide services such as arbitration, power management, and driver matching (see "The I/O Registry and the I/O Catalog" (page 31)).

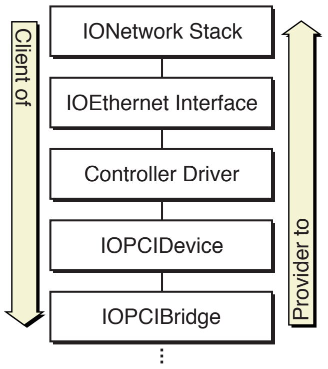
Nubs act as bridges between two drivers and, by extension, between two families. A driver communicates with a nub (and the nub's family) as its client and may, through its family, publish a nub which finds (by matching) a driver for which it is a provider. Usually a driver publishes one nub for each individual device or service it controls; however, when a driver supports a specific piece of hardware it can act as its own nub.

The Anatomy of an I/O Connection

The I/O Kit's layered architecture models the chain of connections between the system's hardware buses and devices, gathering common functionality into classes your driver can interact with. Each layer is a client of the layer below it and a provider of services to the layer above it. Broad groupings of layers, defined by the I/O Kit families, define the functionality common to a general type of I/O provider, such as networking or PCI bus devices.

Consider Figure 2-1 (page 27), which illustrates a typical layering of client and provider objects for a PCI-based Ethernet controller driver in the Network family.

Figure 2-1 Driver objects as clients and providers

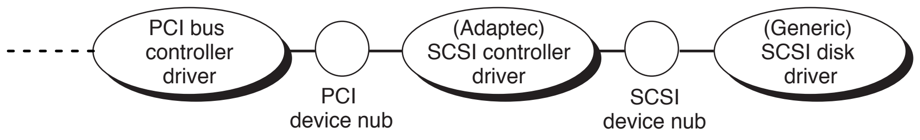


As this diagram shows, your driver typically fits between two families, inheriting from a class in the upper-layer family and using the services of the lower-layer family. In the case of the Ethernet controller, the driver participates in a stack of C++ objects comprising instances of classes from the networking and PCI families:

IONetworkStack (interface managing object)	Connects I/O Kit objects to the BSD networking facilities.
IOEthernetInterface (nub)	Manages device-independent data transmission and reception.
Controller Driver (driver)	Operates the Ethernet controller through the IOPCIDevice object. This object inherits from a networking family class called IOEthernetController.
IOPCIDevice (nub)	Match point for the controller; provides basic PCI bus interaction to the controller.
IOPCIBridge (driver)	Manages the PCI bus. (Other objects provide services to the IOPCIBridge; their specific identities depend on the hardware configuration.)

Another way to look at a stack of driver objects in a typical I/O connection is to consider the stack from a dynamic perspective. In other words, what happens when a Mac OS X system discovers a new device attached to it? How is the stack of driver objects constructed? For this, let's use the example of a SCSI disk drive; the general order of creation or discovery in Figure 2-2 (page 28) is left to right.

Figure 2-2 Driver objects in a connection for a SCSI disk driver


PCI bus controller driver
PCI device nub
(Adaptec) SCSI controller driver
SCSI device nub
(Generic) SCSI disk driver

This figure illustrates how a SCSI disk driver, a member of the Storage family, is connected to the PCI bus. As each individual connection is made, the newly created driver or nub is also added to the I/O Registry (described in "The I/O Registry and the I/O Catalog" (page 31)). The chain of connections takes place in several steps:

1. The PCI bus controller driver, a member of the PCI family, discovers a PCI device and announces its presence by creating a nub (`IOPCIDevice`).

2. The nub identifies (matches) an appropriate device driver—in this case, a SCSI controller driver—and requests that it be loaded. Loading the SCSI controller driver causes the SCSI Parallel family, and all families that it depends on, to be loaded as well. The SCSI controller driver is given a reference to the `IOPCIDevice` nub.

3. The SCSI controller driver, which is a client of the PCI family and a provider of SCSI Parallel family services, scans the SCSI bus for devices that might be clients of these services. Upon finding such a device (a disk), the driver announces the device's presence by creating a nub (`IOSCSIDevice`).

4. The nub, by going through the matching procedure, finds a device driver (a disk driver) that is appropriate for the device and requests that this driver be loaded. Loading the disk driver causes the Storage family, and all families that it depends on, to be loaded as well. The disk driver is now a client of the SCSI Parallel family and a member of the Storage family. The disk driver is given a reference to the `IOSCSIDevice` nub.

In many cases, applications and other "user space" programs can use the I/O Kit's device-interface technology to drive devices (including mass storage devices), obviating the need for a kernel-resident driver. See "Controlling Devices From Outside the Kernel" (page 36) for an overview of this plug-in technology.

The Runtime Environment of Device Drivers

The I/O Kit provides a runtime environment with several powerful features for driver writers, including:

- A dynamic, layered driver architecture that allows drivers to be loaded and unloaded at any time and delays reserving costly system resources until they're needed
- Standard facilities for managing data during common I/O operations
- A robust system for protecting access to driver resources during I/O operations, which frees driver writers from having to write their own code to enable and disable interrupts and manage locks on the driver's private resources
- Access to services in the libkern C++ library (on which the I/O Kit itself is based) to manage collections, perform atomic operations, and byte-swap values for use on different kinds of hardware

The following section summarizes each of these features.

Runtime Features

I/O Kit drivers can be loaded and unloaded or activated and deactivated at any time, through events initiated by software—as when networking stacks are brought up and down—and by hardware—as when a USB device is added to or removed from the bus. Nearly all drivers must work within the context of a dynamically changing system. The I/O Kit makes this easier by defining a standard life cycle for driver objects. By implementing a small set of functions, summarized in "The General I/O Kit Classes" (page 34), your driver can gracefully handle the addition and removal of devices and services, as well as changes induced by the power-management system.

Nearly all I/O operations require the same preparation in Mac OS X:

- Paging virtual memory into physical memory
- Wiring memory down so it can't be paged out during I/O operations
- Building scatter/gather lists that describe the data buffers to read or write

The I/O Kit provides a set of utility classes to help drivers prepare memory for I/O operations and to build scatter/gather lists, including the IOMemoryDescriptor and IOMemoryCursor classes. For more information on these facilities, see the chapter "Managing Data" (page 87).

Drivers running in a multithreaded system must be able to protect their resources from reentrant or concurrent access. The I/O Kit includes a small set of classes for this purpose. A work loop object runs a dedicated thread and manages a gating mechanism for exclusive access to data. Other objects, called event sources, use the gating mechanism to serialize function calls that access critical resources, closing the work loop gate before invoking the function. For more information on work loops and event sources, see the chapter "Handling Events" (page 73).

The libkern C++ library, on which the I/O Kit itself is based, provides services commonly needed by drivers, including:

- Arithmetic and logical operations that are guaranteed to be atomic
- Byte-swapping of values between big-endian and little-endian formats
- Classes for common collections of data such as strings, arrays, and dictionaries

For more information on the libkern classes see "The OS Classes" (page 33) as well as the libkern reference documentation installed with the Mac OS X Developer package.

Kernel Programming Constraints

Kernel code is always held resident in physical memory, and cannot be paged out by the virtual memory system. This makes kernel resources much more expensive than application program resources. Your driver should reside in the kernel if:

- It takes primary interrupts (in which case it must live in the kernel)
- Its primary client resides in the kernel; for example, mass storage drivers because file-system stacks reside in the kernel

Drivers for disks, network controllers, and keyboards, for example, reside in the kernel. If your driver is only occasionally used by one user-space program at a time, it should be loaded by the program and reside within it. Drivers for such devices as scanners and printers reside within user-space programs, using the I/O Kit's device-interface mechanism to communicate with devices. For more information on device interfaces, see "Controlling Devices From Outside the Kernel" (page 36).

Even if your driver resides in the kernel, you should minimize the amount of kernel-resident code and the amount of processing done by that code. For example, a dedicated application that controls an interrupt-driven hardware device should supply a driver that puts the minimum code in the kernel needed to service the interrupt, make the data available to its client, then return. For additional reasons to be cautious about programming in the kernel, see "Should You Program in the Kernel?" (page 22).

If your driver must reside in the kernel, you should be aware of the following issues:

- Most importantly, the kernel is a single program—there is no memory protection between your driver and the rest of the kernel. A kernel-resident driver that behaves badly can crash or hang the operating system.
- A more subtle issue is that function call stacks within the kernel are limited to 16 KB. Be careful not to declare large local variables in functions. Whenever possible, you should preallocate buffers and reuse them.

Kernel-resident drivers have full access to kernel programming interfaces. However, because of their low level of operation, drivers should use only Mach calls and not BSD calls. Many parts of the BSD kernel code aren't currently safe for multithreading or multiprocessing. In any case, drivers rarely need to perform Mach calls directly, as the I/O Kit provides interfaces to most of the kernel-level functionality needed by a driver.

The I/O Registry and the I/O Catalog

The I/O Registry is a dynamic database that records the network of driver objects participating in hardware connections on a Mac OS X system and tracks the provider-client relationships among those objects. A device driver must be recorded in the I/O Registry to participate in most I/O Kit services.

The I/O Registry is a critical part of the I/O Kit because it supports the dynamic features of the operating system, which allows users to add or remove devices (particularly FireWire or USB devices) to and from a running system and have them immediately available, without the need for a reboot. As hardware is added, the system automatically finds and loads the necessary drivers and updates the I/O Registry to reflect the new device configuration; when hardware is removed, the appropriate drivers are unloaded and the Registry is updated again. The Registry always resides in system memory and is not stored on disk or archived between boots.

The I/O Registry structures its data as an inverted tree. Each object in the tree descends from a parent node and can have one or more child nodes; if it is a "leaf" node, it has no children. Almost every node in the tree represents a driver object: a nub or an actual driver. These objects must inherit from the IORegistryEntry class (which is the superclass of IOService, the superclass of all driver classes). The central characteristic of IORegistryEntry objects is a list of associated properties. These properties reflect the personality used in driver matching (see "Driver Matching" (page 31)) and otherwise add information about a driver. The properties captured in the Registry derive from each driver's information property list, a file in the driver KEXT containing key-value pairs describing the driver's characteristics, settings, and requirements.

Another dynamic database, the I/O Catalog, works closely with the I/O Registry. The I/O Catalog maintains entries for all available drivers on a system. When a nub discovers a device, it requests a list of all drivers of the device's family from the I/O Catalog.

You can examine the I/O Registry using the I/O Registry Explorer application and the `ioreg` command-line tool, both included in the Mac OS X Developer package. You can also programmatically explore and manipulate the properties of Registry entries using the member functions of the IORegistryEntry class. From applications and other programs in user space, you can search for and access driver information in the I/O Registry using APIs in the I/O Kit framework.

For more about the I/O Registry and the I/O Catalog see the chapter "The I/O Registry" (page 39). For further information about the IORegistryEntry class see "Dynamic Driver Registration (IORegistryEntry)" (page 59) in the chapter ""The Base Classes" (page 51)."

Driver Matching

A primary function of nubs is to provide matching services, matching drivers to devices. Unlike in Mac OS 8 and 9, drivers are not loaded automatically simply because they are installed. In Mac OS X, a driver must first be matched to an existing device before that driver can be loaded.

Driver matching is an I/O Kit process in which a nub, after discovering a specific hardware device, searches for the driver or drivers most suited to that device. To support driver matching, each device driver defines one or more **personalities**that specify the kinds of devices it can support. This information is stored in XML dictionaries defined in the information property list in the driver's bundle. The dictionary values specify whether a driver is a candidate for a particular device.

When a nub detects a device, the I/O Kit finds and loads a driver for the nub in three distinct phases, using a subtractive process until a successful candidate is found. The phases of matching are:

1. **Class matching**—eliminates drivers of the wrong device class.

2. **Passive matching**—examines each remaining driver's personalities for properties specific to the device, eliminating those drivers that don't match.

3. **Active matching**—the remaining driver candidates probe the device to verify that they can drive it.

When a matching driver is found, its code is loaded and an instance of the principal class listed in the personality is created. At this point the driver's life cycle begins. See "Driver Object Life Cycle" (page 60) in the chapter ""The Base Classes" (page 51)" for details.

For a detailed discussion of driver personalities and the matching process, see the chapter "Driver and Device Matching" (page 43).

The I/O Kit Class Hierarchy

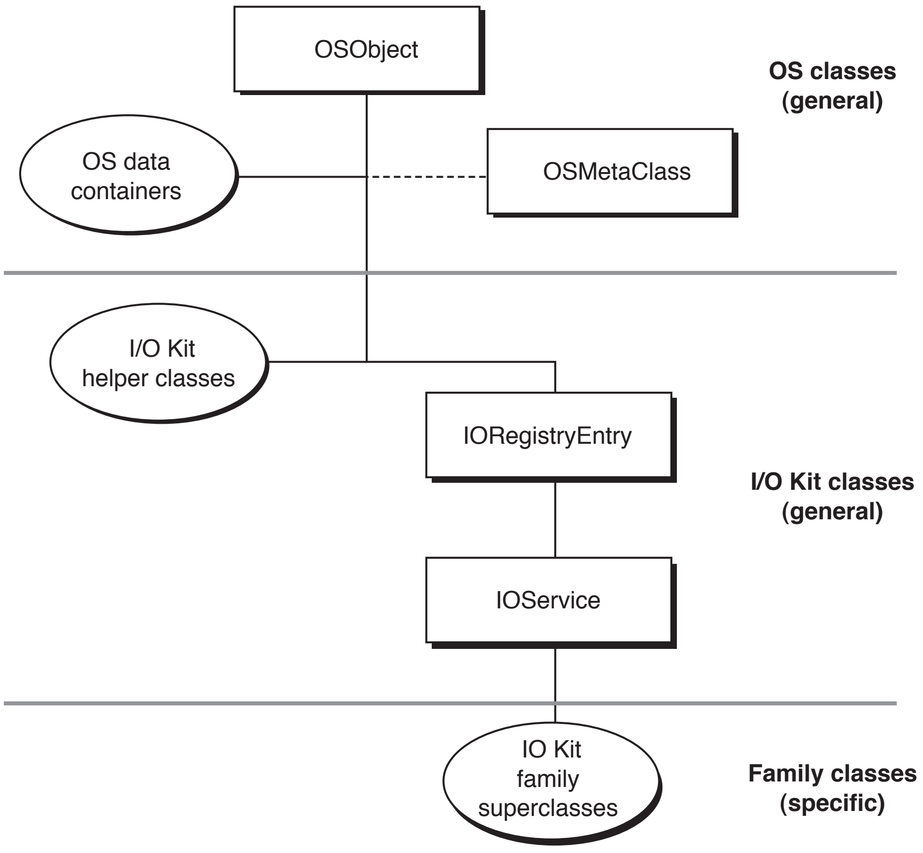
The I/O Kit encompasses dozens of C++ classes and is itself an extension of the libkern C++ library, the foundation for loadable kernel modules. Taken together, the I/O Kit and libkern would seem to form a forbiddingly large and complex hierarchy of classes. Yet the essential structure of that hierarchy is fairly simple, as Figure 2-3 (page 33) illustrates.

You can break down the extended I/O Kit class hierarchy into three broad groupings:

- The classes of libkern (sometimes called the OS classes because of their "OS" prefix)
- The I/O Kit base classes and helper classes
- The classes of the I/O Kit families

For information on where the binaries and header files of libkern and the I/O Kit library are installed, see "Frameworks and Libraries" (page 20). For detailed information on the features and interfaces of the base classes in this hierarchy—OSObject, OSMetaClass, IORegistryEntry, and IOService—see "The Base Classes" (page 51).

Figure 2-3 I/O Kit extended class hierarchy



Note: The appendix "Base and Helper Class Hierarchy" (page 145) contains a hierarchy chart for the I/O Kit base and helper classes; the appendix "I/O Kit Family Reference" (page 117)includes class hierarchy charts for most I/O Kit families.

The OS Classes

The I/O Kit is built on top of the libkern C++ library; that is to say, the root superclass for the I/O Kit–specific classes is IORegistryEntry, which inherits from libkern's OSObject. As is the I/O Kit, libkern is written in a subset of C++ suitable for use in loadable kernel modules. Specifically, the libkern C++ environment excludes the C++ exception-handling and the Runtime Type Information (RTTI) facilities. Instead the OS base classes implement a suitable equivalent of the RTTI feature, among other things.

At the root of the extended hierarchy is the OSObject class and closely related to this class is the OSMetaClass class. All other OS classes are "helper" classes for such things as collections and other data containers. The following summarizes the roles these classes play:

- OSMetaClass implements a runtime type information (RTTI) mechanism, enables some degree of object introspection, and supports runtime allocation of objects derived from OSObject by class name.

- OSObject features APIs for reference counting (`retain` and `release`), memory management of retained objects, and the automatic disposal of objects when they are no longer needed. OSObject also provides a dynamic default implementation of the `init` and `free` methods.
- The OS data containers are subclasses of OSObject whose instances encapsulate various types of data values (such as booleans, numbers, strings) and implement and iterate over collections such as arrays and dictionaries.

  The OS data containers approximately match their user-space counterparts, the Core Foundation containers, in both name and behavior. Because the characteristics of the OS and Core Foundation classes are so similar, the system can easily convert a Core Foundation type to an OS type and vice versa. For example, a CFArray object is transformed into an OSArray when crossing the user-kernel boundary.

The OS classes are generally useful for all code written for the kernel, not just device drivers. For example, kernel extensions implementing networking services or file systems can also take advantage of these classes. OSObject in particular is an essential common superclass for kernel code. For one kernel module (KMOD) to reference objects created by another KMOD, the objects must ultimately derive from OSObject. Most I/O Kit classes assume that the objects being passed around are derived from OSObject.

If you are porting existing C++ code to the I/O Kit, you are not required to use the OS classes. But if you decide to forgo the features that the these classes provide, such as reference counting or data containers, you'll probably need to implement them yourself.

For more on the OSObject and OSMetaClass classes, see "The libkern Base Classes" (page 52) in the chapter ""The Base Classes" (page 51)."

The General I/O Kit Classes

The middle group of the extended class hierarchy comprises the IO Kit base classes— IORegistryEntry and IOService—and a set of helper classes for resource management, data management, and thread and input control. This group of I/O Kit classes is designated "general" because all device-driver classes can potentially make use of them.

The root class of the I/O Kit hierarchy is IORegistryEntry; by virtue of inheritance from IORegistryEntry, an I/O Kit object can be a node in the I/O Registry and have one or more property tables (driver personalities) associated with it. IORegistryEntry implements a number of features:

- It manages connection into the Registry through a driver's `attach` and `detach` entry points
- It manages the property tables defining driver personalities using OSDictionary objects
- It implements locking in the Registry, allowing updates to the Registry to be made atomically

IOService is the sole direct subclass of IORegistryEntry. Almost all I/O Kit family superclasses inherit, directly or indirectly, from IOService. Most importantly, IOService specifies the life cycle of device drivers within a dynamic runtime environment. Through matching pairs of virtual functions—such as `init`/`free`, `start`/`stop`, and `open`/`close`—IOService defines how driver objects initialize themselves, attach themselves into the I/O Registry, perform all necessary allocations, and then reverse the effects of these actions in the proper order. To support its management of driver life cycles, IOService provides matching services (assisting with probing, for instance) and instantiates drivers based on the existence of a provider. In addition, IOService includes member functions that are useful for various purposes, including:

- Notification and messaging
- Power management
- Device memory (mapping and accessing)
- Device interrupts (registering, unregistering, enabling, causing, and so on)

For more on the IORegistryEntry and IOService classes, see "The I/O Kit Base Classes" (page 59) in the chapter ""The Base Classes" (page 51)."

Most I/O Kit helper classes have several functions related to the runtime environment of device drivers:

- Implementing work loops and event sources (interrupts, timers and commands) along with associated locks and queues
- Implementing memory cursors and memory descriptors for managing the data involved in I/O transfers

For more on the I/O Kit helper classes, see the chapters "Handling Events" (page 73) and "Managing Data" (page 87).

The I/O Kit Family Classes

Most drivers are instances of a subclass of a class in an I/O Kit family; the family classes, in turn, tend to be subclasses of IOService. Your driver class should be a direct subclass of the most appropriate family class for the driver you're trying to write. For example, if you are writing an Ethernet controller driver, your driver class should be a subclass of IOEthernetController, not IONetworkController (the superclass of IOEthernetController).

The I/O Kit has over a dozen families, each with its own set of classes; these families include the following:

- ADB
- ATA and ATAPI
- Audio
- FireWire
- Graphics
- HID (Human Interface Devices)
- Network
- PC Card
- PCI and AGP
- SBP-2
- SCSI Architectural Model
- SCSI Parallel
- Serial

- Storage
- USB

Apple will add additional families as they are developed. If you require a family for a device and it's currently not supported, you can try writing your own family classes. However, don't assume that a new family is required if one does not currently exist. In many cases, the IOService class provides all the services a driver requires; such "family-less" drivers can support many devices that are specific to certain vendors. For more information on families, see the chapter "I/O Kit Families" (page 65) and the appendix "I/O Kit Family Reference" (page 117).

Controlling Devices From Outside the Kernel

Perhaps one of the more compelling features of Mac OS X is the inviolable separation it enforces between the virtual address spaces of processes. Unless laborious arrangements are made for shared memory, one process cannot directly touch data mapped to another process's address space. This separation enhances the stability and reliability of the system by preventing memory trashers and similar annoyances from bringing processes down.

Even more important is the separation between the address spaces of the kernel and of all other processes, which are sometimes said (from the perspective of the kernel) to inhabit "user space." If an application or other program in user space somehow violates the address space of the kernel, the whole system can come crashing down. To make this separation between kernel and user space even more airtight, programs in user space cannot even directly call kernel APIs. They must make system calls to (indirectly) access kernel APIs.

Sometimes, however, a program in user space needs to control or configure a device, and thus needs access to I/O Kit services in the kernel. For example, a game might need to set monitor depth or sound volume, or a disk-backup program might need to act as the driver for a tape drive. Other examples of applications that must somehow interact with the kernel to drive hardware include those running or interpreting data from scanners, joysticks, and digital cameras.

To answer this requirement, the I/O Kit includes two mechanisms: device interfaces and POSIX device nodes. Through a plug-in architecture and well-defined interfaces, the device-interface mechanism enables a program in user space to communicate with a nub in the kernel that is appropriate to the type of device it wishes to control. Through the nub the program gains access to I/O Kit services and to the device itself. For storage, serial, and networking devices, applications can obtain the information they need from the I/O Kit to access and control these devices using POSIX APIs.

Keep in mind that there are some family services that the I/O Kit refuses to export to user space as device interfaces; these services are available only inside the kernel. An example is the PCI family. For reasons of stability and security, external access to PCI resources is forbidden. The appendix "I/O Kit Family Reference" (page 117) identifies the families that export device interfaces.

This section summarizes information in *Accessing Hardware From Applications*. Refer to this document for a complete description of device interfaces and how to use them.

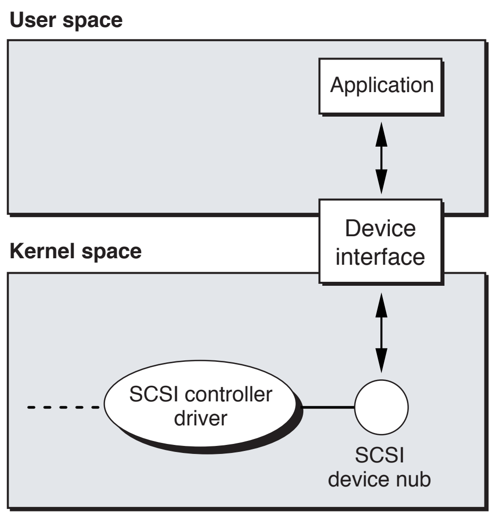
The Device-Interface Mechanism

A device interface is a plug-in interface between the kernel and a process in user space. The interface conforms to the plug-in architecture defined by Core Foundation Plug-in Services (CFPlugIn), which, in turn, is compatible with the basics of Microsoft's Component Object Model (COM). In the CFPlugIn model, the kernel acts as the plug-in host with its own set of well-defined I/O Kit interfaces, and the I/O Kit framework provides a set of plug-ins (device interfaces) for applications to use.

Conceptually, a device interface straddles the boundary between user space and the kernel. It handles negotiation, authentication, and similar tasks as if it were a kernel-resident driver. On the user-space side, it enables communication with the application (or other program) through its exported programmatic interfaces. On the kernel side it enables communication with an appropriate I/O Kit family through a nub created by a driver object of that family. From the kernel's perspective, a device interface appears to be a driver and is known as a "user client." From the application's perspective, the device interface appears as a set of functions that it can call and through which it can pass data to the kernel and receive data back from it. That's because, at an elemental level, a device interface is a pointer to a table of function pointers (although it can also include data fields). Applications, once they obtain an instance of a device interface, can call any of the functions of the interface.

Figure 2-4 (page 37) illustrates the architecture of a device interface, showing an application that has acquired access to a SCSI hard disk through a device interface. It is best to view this diagram as a variation of Figure 2-2 (page 28), which shows the series of driver-object connections made for a kernel-resident SCSI disk driver.

Figure 2-4 An application controlling a SCSI device through a device interface.



At the start, the same series of actions—device discovery, nub creation, matching, driver loading—occurs from the PCI bus driver to the SCSI device nub. But then the SCSI device nub matches and loads the device interface as its driver instead of a kernel-resident driver.

Before an application can use the device-interface mechanism to access a device, it must find the device. It accomplishes this through a process called device matching. In device matching, an application creates a "matching dictionary" that specifies the properties of the target device, then calls an I/O Kit function, passing in the dictionary. The function searches the I/O Registry and returns one or more matching driver objects that the application can then use to load an appropriate device interface. For more on this topic, see "Device Matching" (page 48).

If you develop a custom driver that is a not a subclass of a class in an I/O Kit family, and you want applications to be able to access the driver, you have to write your own device interface. Any code that communicates between user space and the kernel must use of one or more of the following facilities:

- BSD system calls
- Mach IPC
- Mach shared memory

The I/O Kit uses primarily Mach IPC and Mach shared memory. In contrast, the networking and file-system components of Mac OS X use primarily BSD system calls.

POSIX Device Files

BSD, a central component of the Mac OS X kernel environment, exports a number of programmatic interfaces that are consistent with the POSIX standard. These interfaces enable communication with serial, storage, and network devices through device files. In any UNIX-based system such as BSD, a device file is a special file located in `/dev` that represents a block or character device such as a terminal, disk drive, or printer. If you know the name of a device file (for example, `disk0s2` or `mt0`) your application can use POSIX functions such as `open`, `read`, `write`, and `close` to access and control the associated device.

The I/O Kit dynamically creates the device files in `/dev` as it discovers devices. Consequently, the set of device files is constantly changing; different devices might be attached to the device files in `/dev` at any one time, and the same devices might have different device-file names at different times. Because of this, your application cannot hard-code device file names. For a particular device, you must obtain from the I/O Kit the path to its device file through a procedure involving device matching. Once you have the path, you can use POSIX APIs to access the device.

Note that you can access networking services from user space using the BSD socket APIs. However, you should generally use sockets only if the higher-level networking APIs in the Carbon and Cocoa environments do not provide you with the features you require.

The I/O Registry

The I/O Registry is a dynamic database that describes a collection of “live” objects (nubs or drivers) and tracks the provider-client relationships between them. When hardware is added or removed from the system, the Registry is immediately updated to reflect the new configuration of devices. A dynamic part of the I/O Kit, the Registry is not stored on disk or archived between boots. Instead, it is built at each system boot and resides in memory.

The I/O Registry is made accessible from user space by APIs in the I/O Kit framework. These APIs include powerful search mechanisms that allow you to search the Registry for an object with particular characteristics. You can also view the current state of the Registry on your computer using applications provided with the developer version of Mac OS X.

This chapter describes the I/O Registry architecture and the planes the Registry uses to represent relationships between objects. It also provides an overview of device matching and introduces applications that allow you to browse the Registry.

I/O Registry Architecture and Construction

It is most useful to think of the I/O Registry as a tree: Each object is a node that descends from a parent node and has zero or more child nodes. The Registry follows the definition of a tree in nearly all respects, with the exception of a small minority of nodes that have more than one parent. The primary example of this situation is a RAID disk controller where several disks are harnessed together to appear as a single volume. Exceptional cases aside, however, viewing the Registry as a tree will help you visualize how it is constructed and updated.

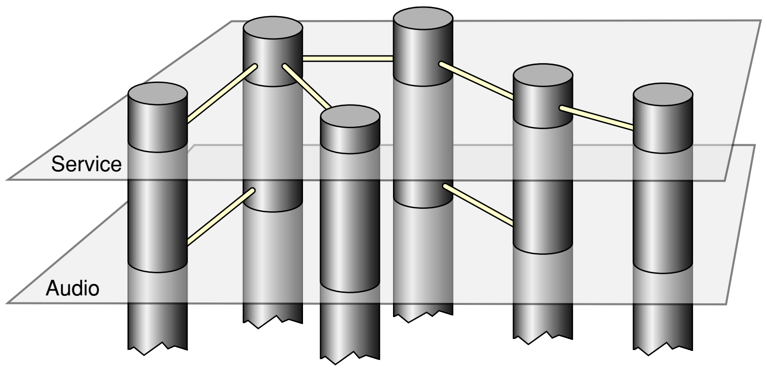
At boot time, the I/O Kit registers a nub for the Platform Expert, a driver object for a particular motherboard that knows the type of platform the system is running on. This nub serves as the root of the I/O Registry tree. The Platform Expert nub then loads the correct driver for that platform, which becomes the child node of the root. The Platform driver discovers the buses that are on the system and it registers a nub for each one. The tree continues to grow as the I/O Kit matches each nub to its appropriate bus driver, and as each bus driver discovers the devices connected to it and matches drivers to them.

When a device is discovered, the I/O Kit requests a list of all drivers of the device’s class type from another dynamic database, the I/O Catalog. Whereas the I/O Registry maintains the collection of objects active in the currently running system, the I/O Catalog maintains the collection of available drivers. This is the first step in a three-step process known as driver matching that is described in “Driver and Device Matching” (page 43).

Information such as class type is kept in the driver's information property list, a file containing XML-structured property information. The property list describes a driver's contents, settings, and requirements in the form of a dictionary of key-value pairs. When read into the system, this information is converted into OS containers such as dictionaries, arrays, and other types. The I/O Kit uses this list in driver matching; a user application can search the I/O Registry for objects with specific properties in a process known as device matching. You can also view the property lists of your computer's currently loaded drivers using I/O Registry Explorer, an application that displays the Registry.

Keeping the tree-like structure of the I/O Registry in mind, now visualize each node extending into the third dimension like a column. The two-dimensional Registry tree, with the Platform Expert nub at its root, is now visible on a plane that cuts perpendicularly through these columns. The I/O Kit defines a number of such planes (you can think of them as a set of parallel planes cutting through the columns at different levels). See Figure 3-1 (page 40) for an illustration of this structure.

Figure 3-1 Two planes in the I/O Registry



There are six planes defined in the I/O Registry:

- Service
- Audio
- Power
- Device Tree
- FireWire
- USB

Each plane expresses a different provider-client relationship between objects in the I/O Registry by showing only those connections that exist in that relationship. The most general is the Service plane which displays the objects in the same hierarchy in which they are attached during Registry construction. Every object in the Registry is a client of the services provided by its parent, so every object's connection to its ancestor in the Registry tree is visible on the Service plane.

The other planes show more specific relationships:

- The Audio plane provides a representation of the audio signal chain that Core Audio framework and its plug-ins use to discover information about the audio signal paths between the system's audio devices.

- The Power plane shows the power interdependencies between I/O Registry objects, allowing you to trace the flow of power from provider to client and discover which objects might be affected if a particular device is powered down.
- The Device Tree plane represents the Open Firmware device hierarchy.
- The FireWire and USB planes each represent the internal hierarchies defined by those standards.

It is important to remember the following points about planes in the I/O Registry:

- All I/O Registry objects exist on all planes, but on any individual plane, only those objects connected by the relationship defined by that plane are visible.
- A driver does not get attached to the Registry on any one particular plane. Instead it may participate in a plane's connections if its provider-client relationships with other objects fit that plane's definition.

The I/O Registry Explorer

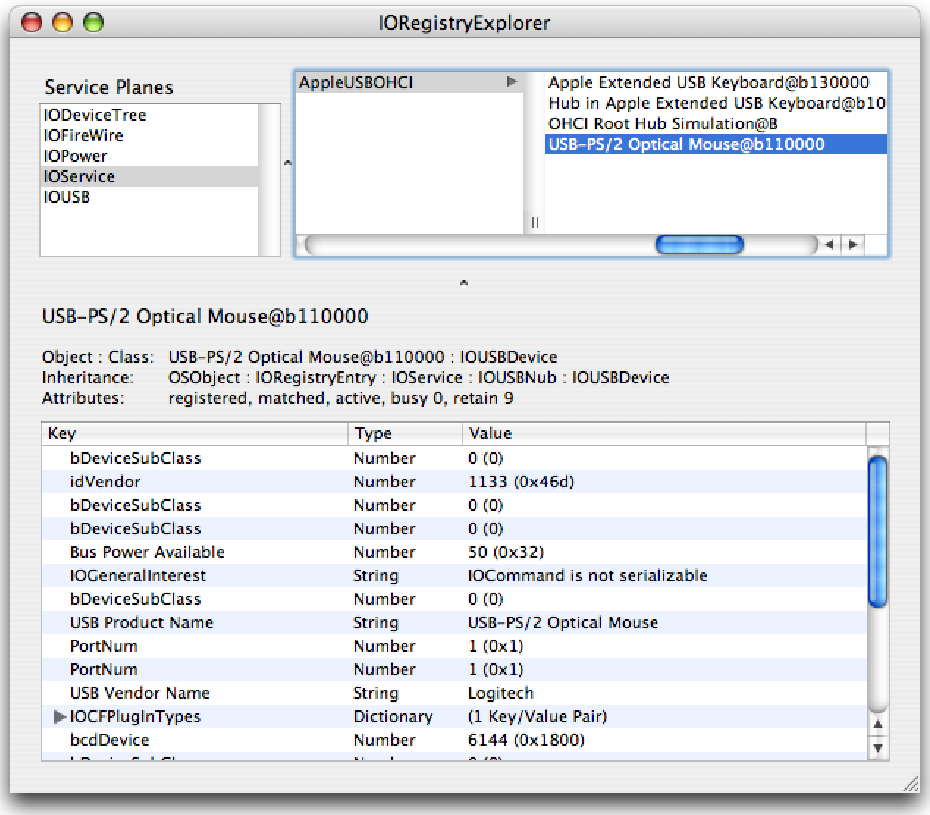
The developer version of Mac OS X provides an application called the I/O Registry Explorer that you can use to examine the configuration of devices on your computer. I/O Registry Explorer provides a graphical representation of the I/O Registry tree. By default, it displays the Service plane, but you can choose to examine any plane. The command-line equivalent, `ioreg`, displays the tree in a Terminal window. This tool has the advantage of allowing you to cut and paste sections of the tree if, for example, you want to send that information in an email message. You can get a complete description of the usage of `ioreg` by typing `man ioreg` at the shell prompt in the Terminal application.

When you open I/O Registry Explorer, a divided window appears with I/O Registry objects in the upper right, the six planes in the upper left, and the property list of the selected object in the lower half of the window. An object followed by a disclosure triangle indicates that it is a parent node. You can traverse the I/O Registry tree by clicking a parent node and dragging the scroller to the right to display its children. Figure 3-2 (page 42) shows an example of a property list in the I/O Registry Explorer window.

Commands in the Tools menu help you search the I/O Registry and examine its contents:

- **Dump Registry Dictionary to Output** places the I/O Registry contents into the console log (viewable through the Console application in `/Applications/Utilities`) if the I/O Registry Explorer was opened from the Finder.
- **Inspector** displays the property list of the currently selected object in ASCII form. Selecting a particular property in the main window causes its value to be displayed in the Inspector window.
- **Force Registry Update** updates I/O Registry Explorer's picture of the I/O Registry to reflect any changes that may have occurred since you first opened the application.
- **Find** performs a case insensitive search on your input string and, if successful, displays the path to the occurrence of the string with object names separated by colons.

Figure 3-2 A sample I/O Registry Explorer window


IORegistryExplorer
Service Planes
IODeviceTree
IOFireWire
IOPower
IOService
IOUSB
AppleUSBOHCI
Apple Extended USB Keyboard@b130000
Hub in Apple Extended USB Keyboard@b10
OHCI Root Hub Simulation@B
USB-PS/2 Optical Mouse@b110000

USB-PS/2 Optical Mouse@b110000

Object : Class: USB-PS/2 Optical Mouse@b110000 : IOUSBDevice
Inheritance: OSObject : IORegistryEntry : IOService : IOUSBNub : IOUSBDevice
Attributes: registered, matched, active, busy 0, retain 9

Key	Type	Value
bDeviceSubClass	Number	0 (0)
idVendor	Number	1133 (0x46d)
bDeviceSubClass	Number	0 (0)
bDeviceSubClass	Number	0 (0)
Bus Power Available	Number	50 (0x32)
IOGeneralInterest	String	IOCommand is not serializable
bDeviceSubClass	Number	0 (0)
USB Product Name	String	USB-PS/2 Optical Mouse
PortNum	Number	1 (0x1)
PortNum	Number	1 (0x1)
USB Vendor Name	String	Logitech
▶ IOCFPlugInTypes	Dictionary	(1 Key/Value Pair)
bcdDevice	Number	6144 (0x1800)

Driver and Device Matching

Before a device—or any service provider—can be used, a driver for it must be found and loaded into the kernel. The I/O Kit defines a flexible, three-phase matching process that narrows a pool of candidate drivers down to one or more drivers. The final candidate (or, if multiple candidates, the most eligible one) is then loaded and given the first opportunity to manage the device or service provider.

The matching process makes use of the matching dictionaries defined as XML key-value pairs in a driver's information property list. Each matching dictionary specifies a personality of the driver, which declares its suitability for a device or service of a particular type.

This chapter discusses driver personalities and the matching language that describes them. It then describes the matching process, which uses the information in the driver personalities to identify the most appropriate driver for a detected device. The chapter also briefly discusses the device-matching procedure that applications use for loading device interfaces. See *Accessing Hardware From Applications* for the complete details.

Driver Personalities and Matching Languages

Each device driver, considered as a loadable kernel extension (KEXT), must define one or more personalities that specify the kinds of devices it can support. This information is stored in XML matching dictionaries defined in the information property list (`Info.plist`) in the driver’s KEXT bundle. A dictionary in this sense is a collection of key-value pairs where the XML tags `<key>` and `</key>` enclose the key. Immediately following the key are the tags enclosing the value; these tags indicate the data type of the value; for example,

```
<integer>74562</integer>
```

would define an integer value.

Each matching dictionary is itself contained within the information property list’s `IOKitPersonalities` dictionary.

The dictionary values of a personality specify whether a driver is a candidate for a particular device. All values in the personality must match in order for the driver to be selected for the device; in other words, a logical AND is performed on the values. Some of the keys may take a list of space-delimited values, which are generally examined in an OR fashion. Thus you might have a “model” key for a certain PC card driver personality that takes a list of model numbers, each identifying a supported model from a specific card vendor.

The specific keys that are required depend on the family. A driver for a PCI card, for example, can define a value that is checked against the PCI vendor and device ID registers. Some families, such as the PCI family, provide fairly elaborate matching strategies. For instance, consider this key-value pair:

```
<key>IOPCIMatch</key>
<string>0x00789004&0x00ffffff 0x78009004&0xff00ffff</string>
```

This expression, which is used to match various Adaptec SCSI cards, consists of two compound values, each of which can be a valid match. To evaluate these values, the driver family reads the 32-bit vendor and device ID is read from the PCI card and masks it with the value to the right of each ampersand. The result of that operation is then compared with the value to the left of the ampersand to determine if there is a match.

Listing 4-1 (page 44) shows a partial listing of a driver personality from the XML file for an Ethernet controller driver.

Listing 4-1 A partial listing of an XML personality for an Ethernet controller

```
 <key>IOKitPersonalities</key>
    <dict>
        <dict>
            <!-- Each personality has a different name. -->
            <key>Name</key>      <string>PCI Matching</string>

            <!-- ... some keys not shown ... -->

            <!-- The name of the class IOKit will instantiate when probing. -->
            <key>IOClass</key>   <string>ExampleIntel82558</string>

            <!-- IOKit matching properties
              -- All drivers must include the IOProviderClass key, giving
              -- the name of the nub class that they attach to. The provider
              -- class then determines the remaining match keys. A personality
              -- matches if all match keys do; it is possible for a driver
              -- with multiple personalities to be instantiated more than once
              -- if several personalities match.
              -->
            <key>IOProviderClass</key>
                <string>IOPCIDevice</string>

            <!-- IOPCIDevice matching uses any of four possible PCI match
             -- criteria. This personality just uses IOPCIMatch to check the
              -- device/vendor ID.
              -->
            <key>IOPCIMatch</key>
                <string>0x12298086</string>

            <!-- The initial match score for this personality.-->
            <key>IOProbeScore</key>    <integer>400</integer>
        </dict>

        <dict>
            <!-- Can have additional personalities. -->
            <!-- ... (not shown) -->
        </dict>
    </dict>
```

Because a driver can contain multiple matching dictionaries, each one defining a different personality for the driver, the same driver code can be loaded for different devices. For purposes of competition, the I/O Kit treats each personality as if it were a driver. If, in any single personality, all of the properties required by the family match, the driver's code is loaded and given a chance to run for that device.

Your driver can have more than one personality for a variety of reasons. It could be that the driver (as packaged in the KEXT) supports more than one type of device, or more commonly, multiple versions of the same type of device. Another reason might be that the driver supports similar devices, each of which is attached to the system on different buses; for example, Zip drives can be attached to USB, FireWire, SCSI, ATAPI, and other buses. Because each of these attaches to a different nub class, it has different matching values. The personalities of a driver can also range from device-generic to device-specific. The personalities of the AppleUSBAudio driver (Listing 4-2 (page 45)) illustrate this.

Listing 4-2 Driver personalities for the AppleUSBAudio driver

```
<key>IOKitPersonalities</key>
<dict>
    <key>AppleUSBAudioControl</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.driver.AppleUSBAudio</string>
        <key>IOClass</key>
        <string>AppleUSBAudioDevice</string>
        <key>IOProviderClass</key>
        <string>IOUSBInterface</string>
        <key>bInterfaceClass</key>
        <integer>1</integer>
        <key>bInterfaceSubClass</key>
        <integer>1</integer>
    </dict>
    <key>AppleUSBAudioStream</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.driver.AppleUSBAudio</string>
        <key>IOClass</key>
        <string>AppleUSBAudioDMAEngine</string>
        <key>IOProviderClass</key>
        <string>IOUSBInterface</string>
        <key>bInterfaceClass</key>
        <integer>1</integer>
        <key>bInterfaceSubClass</key>
        <integer>2</integer>
    </dict>
    <key>AppleUSBTrinityAudioControl</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.driver.AppleUSBAudio</string>
        <key>IOClass</key>
        <string>AppleUSBTrinityAudioDevice</string>
        <key>IOProviderClass</key>
        <string>IOUSBInterface</string>
        <key>bConfigurationValue</key>
        <integer>1</integer>
        <key>bInterfaceNumber</key>
        <integer>0</integer>
        <key>idProduct</key>
```

```
            <integer>4353</integer>
            <key>idVendor</key>
            <integer>1452</integer>
        </dict>
    </dict>
```

This matching dictionary defines three personalities: `AppleUSBAudioControl`, `AppleUSBAudioStream`, and `AppleUSBTrinityAudioControl`. In matching for a detected USB Trinity audio-control device, the `AppleUSBTrinityAudioControl` would be chosen; for any other audio-control device, the generic personality (`AppleUSBAudioControl`) would match.

One common property of personalities is the probe score. A probe score is an integer that reflects how well-suited a driver is to drive a particular device. A driver may have an initial probe-score value in its personality and it may implement a `probe` function that allows it to modify this default value, based on its suitability to drive a device. As with other matching values, probe scores are specific to each family. That's because once matching proceeds past the class-matching stage, only personalities from the same family compete. For more information on probe scores and what a driver does in the `probe` function, see "Device Probing" (page 47).

Driver Matching and Loading

At boot time and at any time devices are added or removed, the process of driver matching occurs for each detected device (or other service provider). The process dynamically locates the most suitable driver in `/System/Library/Extensions` for the device or service.

As described in "Driver Matching" (page 31) in the chapter "Architectural Overview" (page 25), the matching process is triggered when a bus controller driver scans its bus and detects a new device attached to it. For each detected device the controller driver creates a nub. The I/O Kit then initiates the matching process and obtains the values from the device to use in matching (for example, examining the PCI registers). Once a suitable driver is found for the nub, the driver is registered and loaded. That driver, in turn, may create its own nub (possibly through behavior inherited from its family), which initiates the matching process to find a suitable driver.

Driver Matching

When a nub detects a device, the I/O Kit finds and loads a driver for the nub in three distinct phases, using a subtractive process. In each phase, drivers that are *not* considered to be likely candidates for a match are subtracted from the total pool of possible candidates until a successful candidate is found.

The matching process proceeds as follows:

1. In the **class matching** step, the I/O Kit narrows the list of potential drivers by eliminating any drivers of the wrong class for the *provider* service (that is, the nub). For example, all driver objects that descend from a SCSI class can be ruled out when the search is for a USB driver.

2. In the **passive matching** step, the driver's personality (specified in a driver's XML information property list) is examined for properties specific to the provider's family. For example, the personality might specify a particular vendor name.

3. In the **active matching** step, the driver's `probe` function is called with reference to the nub it is being matched against. This function allows the driver to communicate with the device and verify that it can in fact drive it. The driver returns a probe score that reflects its ability to drive the device. See "Device Probing" (page 47) for more information. During active matching, the I/O Kit loads and probes all candidate drivers, then sorts them in order of highest to lowest probe score.

The I/O Kit then chooses the remaining driver with the highest probe score and starts it. If the driver successfully starts, it is added to the I/O Registry and any remaining driver candidates are discarded. If it does not start successfully, the driver with the next highest probe score is started, and so on. If more than one driver is in the pool of possible candidates, the more generic driver typically loses out to the more specific driver if both claim to be able to drive the device.

Device Probing

During the active matching phase, the I/O Kit requests each driver in the pool of remaining candidates to probe the device to determine if they can drive it. The I/O Kit calls a series of member functions defined in the IOService class and overridden in some cases by the driver's class. These functions, and the order in which they are called, are

```
init()
attach()
probe()
detach()
free() /* if probe fails */
```

These functions comprise the first part of a driver's life cycle (see "Driver Object Life Cycle" (page 60) in the chapter "The Base Classes" (page 51) for the full story). Note that four of these functions form complementary pairs, one nested inside the other: `init` and `free` are one pair, and `attach` and `detach` are the other.

During active matching, the code of a candidate driver is loaded and an instance of the principal class listed in the personality is created. The first function invoked is `init`, which is the `libkern` equivalent of the constructor function for the class. For I/O Kit drivers, this function takes as its sole argument an OSDictionary object containing the matching properties from the selected driver personality. The driver can use this to identify what specific personality it's been loaded for, determine what level of diagnostic output to produce, or otherwise establish basic operating parameters. I/O Kit drivers typically don't override the `init` function, performing their initialization in later stages.

However, if you do override the `init` function—or almost any other function of the driver life cycle—you must take care to do two things. The first is to invoke your superclass's implementation of the function. When you do this depends on the function; for example, in implementing `init` you should invoke the superclass's implementation as the first thing, and in `free` you should invoke it as the last statement of the function. The second general rule is that you should undo in the second function of a pair what you've done in the first function; thus, if you allocate memory for any reason in `init`, you should free that memory in `free`.

Next, the `attach` function (which is bracketed with the `detach` function) is called. The default implementation of `attach` attaches the driver to the nub through registration in the I/O Registry; the default implementation of `detach` detaches the driver from the nub. A driver can override the default implementations, but rarely needs to do so.

After `attach` the `probe` function is invoked. The I/O Kit always calls a driver's `probe` function if the driver's matching dictionary passively matches the provider (the nub). A driver may choose not to implement `probe`, in which case IOService's default implementation is invoked, which simply returns `this`.

The `probe` function takes as arguments the driver's provider and a pointer to a probe score. The probe score is a signed 32-bit integer initialized to a value specified in the driver's personality (or to zero if not explicitly initialized). The driver with the highest initial probe score is given the first chance to start operating the device. The purpose of the `probe` function is to offer drivers an opportunity to check the hardware and to modify their default probe scores as assigned in their personalities. A driver can check device-specific registers or attempt certain operations, adjusting its probe score up or down based on how well suited it is for the device it is examining. Whatever it finds, each driver must leave the hardware in the same state it was in when `probe` was invoked so the next driver can probe the hardware in its original state.

A driver, in its `probe` function, returns a driver object (`IOService *`) if the probe was successful and returns zero otherwise. The returned object is usually the driver itself, but the driver can return another driver that is more suited to the provider. The probe score is an in-out parameter, which `probe` can modify based on what it discovers about the device.

Driver Loading

After all drivers have probed the device, the one with the highest probe score is attached and its `start`function, which must be implemented by all drivers, is invoked. The `start` function initializes the device hardware and prepares it for operation. If the driver succeeds in starting, it returns `true`; the remaining candidate driver instances are discarded and the driver that started successfully continues operating. If the driver cannot initialize the hardware it must leave the hardware in the state it was in when `start` was invoked and return `false`. The failing driver is then detached and discarded, and the candidate driver with the next highest probe score is given a chance to start.

Some time after this occurs, all loaded drivers that are not currently in use are unloaded.

Device Matching

A user application that requires access to a device must first search for that device and then acquire the appropriate device interface to communicate with it. This process is known as device matching. Unlike driver matching, device matching searches the I/O Registry for a driver that is already loaded.

To perform device matching, follow these basic steps:

1. Establish a connection with the I/O Kit by obtaining a Mach port.

2. Define a dictionary that specifies the type of device to search for in the I/O Registry. The search can be refined by setting additional values in the dictionary. For example, a search for IOMedia objects can be narrowed down to find all ejectable media. You can find the values to match in the device header files (such as `IOSCSIDevice.h` or `IOATADevice.h`), by referring to the family-specific documentation, or by looking at the information property lists displayed in output from the I/O Registry Explorer application.

3. Obtain a list of all objects in the Registry that match your dictionary and choose the appropriate device.

4. Access the device you have chosen by obtaining a device interface for it. This step is explained more fully in "Controlling Devices From Outside the Kernel" (page 36).

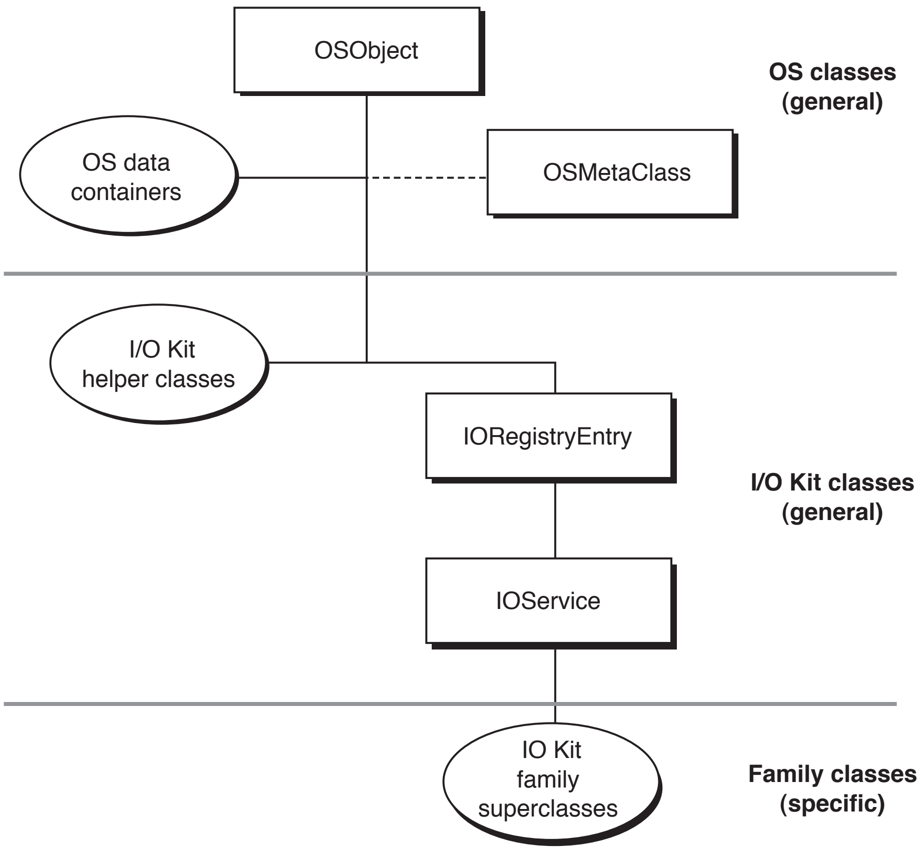
See the document *Accessing Hardware From Applications* for a full description of device matching.

The Base Classes

The I/O Kit is an object-oriented framework consisting primarily of dozens, if not hundreds, of C++ classes. These classes can be organized by virtue of their inheritance relationships in a class hierarchy. As with all class hierarchies, the I/O Kit's can be depicted as an inverted tree, with childless nodes—classes without any subclasses—as the leaves of the tree. Carrying the analogy further, the classes at the trunk and, especially, the root of the tree are those that most classes of the hierarchy inherit from. These are the base classes.

Figure 5-1 (page 51) shows the general outline of the I/O Kit's class hierarchy and the positions of the base classes within this hierarchy.

Figure 5-1 The base classes of the I/O Kit class hierarchy



As the diagram illustrates, the base classes specific to the I/O Kit are IOService and IORegistryEntry; also included as base classes—through inheritance—are the libkern library's OSObject and (in a special sense) OSMetaClass.

Given the centrality of these classes, it is apparent how important it is to understand them. They provide not just the behavior and data structures that all other classes of the I/O Kit inherit. They define the *structure* of behavior for kernel and driver objects: how objects are created and disposed of, how metaclass information is captured and revealed, how driver objects should behave within a dynamic runtime environment, and how the client/provider relationships among driver objects are dynamically established. If you're writing device drivers using the I/O Kit, you're going to have to deal with the base classes in your code early on and frequently thereafter, so it's a good idea to become familiar with them.

This chapter also gives an overview of some generally useful functions and data types. Even though these functions and types do not properly belong in a discussion of base classes (since they are not affiliated with any class), their utility in a variety of circumstances makes them almost as central as any of the base classes.

The libkern Base Classes

The I/O Kit is built on top of the libkern C++ library, which is written in a subset of C++ suitable for use in loadable kernel modules. Specifically, the libkern C++ environment does not support multiple inheritance, templates, the C++ exception-handling facility, and runtime type information (RTTI). The C++ RTTI facility is omitted because it doesn't support dynamic allocation of classes by name, a feature required for loading kernel extensions. RTTI also makes considerable use of exceptions. However, the libkern C++ environment defines its own runtime typing system, which does support dynamic loading.

Exceptions are forbidden in the kernel for reasons of both cost and stability. They increase the size of the code, thereby consuming precious kernel memory, and introduce unpredictable latencies. Further, because I/O Kit code may be invoked by many client threads, there's no way to guarantee that an exception will be caught. Using `try`, `throw`, or `catch` in any kernel extension is not supported and will result in a compilation error. Although you can't use exceptions in an I/O Kit driver, your driver should always check return codes where appropriate.

Apple highly recommends that you base all kernel C++ code, including that for device drivers, on the libkern base classes, OSObject and OSMetaClass, and observe the conventions prescribed by those classes (see "Type Casting, Object Introspection, and Class Information" (page 56)). Classes that are completely private to your driver need not be based on OSObject and need not follow these conventions. Such classes, however, will be limited in their interaction with libkern classes. For example, all libkern collection classes store objects that inherit from OSObject. Custom classes that don't inherit from OSObject can't be stored in libkern collections such as OSDictionary or OSArray objects.

Important: At present, the loader does not allow the use of any OSObject subclass that requires qualification, such as a nested class or a class declared within a namespace (for an example of a namespace declaration, see "Language Choice" (page 18)). For example, the following nested class declaration in an I/O Kit driver would prevent the driver from loading:

```
class com.mycompany.driver.myClass {
    class myNestedClass : public IOService {}; // This is not allowed.
};
```

Object Creation and Disposal (OSObject)

OSObject is at the root of the extended I/O Kit hierarchy. It inherits from no (public) superclass, and all other libkern and I/O Kit classes (except for OSMetaClass) inherit from it. OSObject implements the dynamic typing and allocation features needed to support loadable kernel modules. Its virtual functions and overridden operators define how objects are created, retained, and disposed of in the kernel. OSObject is an abstract base class, and therefore cannot itself be instantiated or copied.

Object Construction

The standard C++ constructors cannot be used in libkern because these constructors use exceptions to report failures; as you may recall, the restricted form of C++ chosen for libkern excludes exceptions. So the main purpose of the OSObject class (and also of the OSMetaClass class) is to reimplement object construction.

For constructing objects, OSObject defines the `init`function and overrides the `new` operator. The `new` operator allocates memory for an object and sets the object's reference count to one. After it uses the `new` operator, the client must call the `init` function on the new object to perform all initializations required to make it a usable object. If the `init` call fails, then the client must immediately release the object.

In support of OSObject's `init` and `new`, the OSMetaClass class implements macros related to object construction. These macros bind a class into the kernel's runtime typing facility and automatically define functions that act as the constructor and destructor for the class. See "Runtime Type Information (OSMetaClass)" (page 54) for more information on these macros and OSMetaClass's implementation of RTTI.

Subclasses of OSObject do not explicitly implement their constructors and destructors since these are essentially created through the OSMetaClass macros. Moreover, you typically invoke neither constructor and destructor functions, nor the C++ `new` and `delete`operators. These functions and operators are reserved for use by the dynamic typing and allocation facilities, which implicitly define them for a class. In their place, OSObject defines a convention for creating and initializing objects. Subclasses do, however, typically override the `init` function to perform initializations specific to the class.

Most libkern and I/O Kit classes define one or more static functions for creating instances. The naming convention varies from class to class, but the name is usually either the base name of the class itself (with a lowercase first letter), or some form of `with...` where the name describes the initialization arguments. For example, OSArray defines the static creation functions `withCapacity`, `withObjects`, and `withArray`; IOTimerEventSource defines `timerEventSource`; and IOMemoryCursor defines `withSpecification`. If a class doesn't have static creation functions, you must use `new` and then invoke the initialization method that takes the place of the C++ constructor, as shown in Listing 5-2 (page 58).

For an overview of the boilerplate code you need to specify your class's constructor and destructor functions, see "Type Casting, Object Introspection, and Class Information" (page 56).

Object Retention and Disposal

OSObject defines a reference-counting and automatic-deallocation mechanism to support the safe unloading of kernel extensions. For this mechanism it uses three virtual member functions—`retain`, `release`, and `free`—and overrides the `delete`operator. Of these, the only functions you should call in your code are `retain` and `release`, and you should follow certain conventions that dictate when to call them.

Newly created objects and copied objects have a reference count of one. If you have created or copied a libkern object and have no need to keep it beyond the current context, you should call `release` on it. This decrements the object's reference count. If that count is zero, the object is deallocated; specifically, the `release` method invokes the alternative destructor, named `free`, and finally invokes the `delete` operator. If you don't own an object—that is, you did not create or copy it—and you want to keep it past the current context, call `retain` on it to increment its reference count. If you did not create, copy, or call `retain` on an object, you should never call `release` on it.

In addition, some functions that return objects pass ownership to the caller, meaning the caller must release the object when it is finished with it, while others don't. See the reference documentation for a given function to find out if your code needs to retain or release an object it receives.

Never invoke the `delete` operator explicitly to free an object. Also, never call `free` directly to free an object; however, you may (and should, in most circumstances) override the `free` function to deallocate memory allocated in your `init` function.

Runtime Type Information (OSMetaClass)

Although libkern's restricted form of C++ excludes the native runtime type information (RTTI) facility, OSMetaClass implements an alternative runtime typing facility that does support dynamic allocation of classes by name. OSMetaClass is not a base class in the true sense; no public libkern or I/O Kit class inherits from it. However, OSMetaClass provides APIs and functionality that are essential for object construction and destruction. OSMetaClass itself is an abstract class and cannot be directly constructed.

The functionality that OSMetaClass offers all libkern-based code includes the following:

- A mechanism for tracking the class hierarchy dynamically
- Safe loading and unloading of kernel modules

  The runtime typing facility enables the system to track how many instances of each libkern (and I/O Kit) class are currently extant and to assign each of these instances to a kernel module (KMOD).
- Automatic construction and deconstruction of class instances
- Macros and functions for dynamic type casting, type discovery, membership evaluation, and similar introspective behavior
- Dynamic allocation of libkern class instances based on some indication of their class type, including C-string names

In libkern's runtime typing facility, one static metaclass instance (derivative of OSMetaClass) is created for every class in a kernel module (KMOD) loaded into the kernel. The instance encapsulates information on the class's name, size, superclass, kernel module, and the current count of instances of that class. The process of loading a kernel module takes place in two phases, the first initiated by the `preModLoad` member function and the second by the `postModLoad` function. During the `preModLoad` phase, OSMetaClass statically constructs, within the context of a single, lock-protected thread, a metaclass instance for each class in the module. In the `postModLoad` phase, OSMetaClass links together the inheritance hierarchy of constructed metaclass objects, inserts the metaclass instances into the global register of classes, and records for each instance the kernel module it derived from. See the OSMetaClass reference documentation for more on `preModLoad`, `postModLoad`, and related functions.

The created store of metaclass information forms the basis for the capabilities of OSMetaClass listed above. The following sections explore the more important of these capabilities in some detail.

Object Construction and Dynamic Allocation

One of the features of OSMetaClass is its ability to allocate libkern objects based upon some indication of class type. Subclasses of OSMetaClass can do this dynamically by implementing the `alloc`function; the class type is supplied by the OSMetaClass subclass itself. You can also allocate an instance of any libkern class by calling one of the `allocClassWithName`functions, supplying an appropriate identification of class type (OSSymbol, OSString, or C string).

Freshly allocated objects have a retain count of 1 as their sole instance variable and are otherwise uninitialized. After allocation, the client should immediately invoke the object's initialization function (which is `init` or some variant of `init`).

OSMetaClass defines a number of runtime type-declaration macros and object-construction macros based on the `alloc` function. Based on the type of class (virtual or otherwise), you must insert one of these macros as the first statement in class declarations and implementations:

`OSDeclareDefaultStructors`
: Declares the data and interfaces of a class, which are needed as runtime type information. By convention this macro should immediately follow the opening brace in a class declaration.

`OSDeclareAbstractStructors`
: Declares the data and interfaces of a virtual class, which are needed as runtime type information. By convention this macro should immediately follow the opening brace in a class declaration. Use this macro when the class has one or more pure virtual methods.

`OSDefineMetaClassAndStructors`
: Defines an OSMetaClass subclass and the primary constructors and destructors for a non-abstract subclass of OSObject. This macro should appear at the top of the implementation file just before the first function is implemented for a particular class.

`OSDefineMetaClassAndAbstractStructors`
: Defines an OSMetaClass subclass and the primary constructors and destructors for a subclass of OSObject that is an abstract class. This macro should appear at the top of the implementation file just before the first function is implemented for a particular class.

`OSDefineMetaClassAndStructorsWithInit`
: Defines an OSMetaClass subclass and the primary constructors and destructors for a non-abstract subclass of OSObject. This macro should appear at the top of the implementation file just before the first function is implemented for a particular class. The specified initialization routine is called once the OSMetaClass instance has been constructed at load time.

`OSDefineMetaClassAndAbstractStructorsWithInit`
: Defines an OSMetaClass subclass and the primary constructors and destructors for a subclass of OSObject that is an abstract class. This macro should appear at the top of the implementation

file just before the first function is implemented for a particular class. The specified initialization routine is called once the OSMetaClass instance has been constructed at load time.

See "Type Casting, Object Introspection, and Class Information" (page 56) for more information on using these macros, including examples of usage.

Type Casting, Object Introspection, and Class Information

OSMetaClass defines many macros and functions that you can use in almost any situation. They help you safely cast from one type to another, discover an arbitrary object's class, determine if an object inherits from a given superclass, find out how many instances of a given class are still allocated, and yield other useful information. Table 5-1 (page 56) summarizes these macros and functions.

Table 5-1 OSMetaClass type-casting and introspection APIs

Function or macro	Description
`OSTypeID`	This macro returns the type ID of a class based on its name.
`OSTypeIDInst`	This macro returns the type ID of the class a given instance is constructed from.
`OSCheckTypeInst`	This macro checks if one instance is of the same class type as another instance.
`OSDynamicCast`	This macro dynamically casts the class type of an instance to a suitable class. It is basically equivalent to RTTI's `dynamic_cast`.
`isEqualTo`	This function verifies if the invoking OSMetaClass instance (which represents a class) is the same as another OSMetaClass instance. The default implementation performs a shallow pointer comparison.
`metaCast` (multiple)	This set of functions determines if an OSMetaClass instance (which represents a class) is, or inherits from, a given class type. The type can be specified as OSMetaClass, OSSymbol, OSString, or a C string.
`modHasInstance`	Returns whether a kernel module has any outstanding instances. This function is usually called to determine if a module can be unloaded.
`getInstanceCount`	This function returns the number of instances of the class represented by the receiver.
`getSuperClass`	This function returns the receiver's superclass.
`getClassName`	This function returns the name (as a C string) of the receiver.
`getClassSize`	This function returns the size (in bytes) of the class represented by the receiver.

Defining C++ Classes in libkern

When implementing a C++ class based on OSObject, you invoke a pair of macros based upon the OSMetaClass class. These macros tie your class into the libkern runtime typing facility by defining a metaclass and by defining the constructor and destructor for your class that perform RTTI bookkeeping tasks through the metaclass.

The first macro, `OSDeclareDefaultStructors` declares the C++ constructors; by convention you insert this macro as the first element of the class declaration in the header file. For example:

```
class MyDriver : public IOEthernetController
{
    OSDeclareDefaultStructors(MyDriver);
    /* ... */
};
```

Your class implementation then uses the companion macro, `OSDefineMetaClassAndStructors`, to define the constructor and destructor, as well as the metaclass that provides the runtime typing information. `OSDefineMetaClassAndStructors` takes as arguments the name of your driver and the name of its superclass. It uses these to generate code that allows your driver class to be loaded and instantiated while the kernel is running. For example, `MyDriver.cpp` might begin like this:

```
#include "MyDriver.h"

// This convention makes it easy to invoke superclass methods.
#define super    IOEthernetController

// You cannot use the "super" macro here, however, with the
//  OSDefineMetaClassAndStructors macro.
OSDefineMetaClassAndStructors(MyDriver, IOEthernetController);
```

The definition of the `super`macro allows convenient access to superclass methods without having to type the whole name of the superclass every time. This is a common idiom of libkern and I/O Kit class implementations.

In place of the C++ constructor and destructor, your class implements an initialization method and a `free`method. For non–I/O Kit classes, the initialization method takes whatever arguments are needed, can have any name (although it usually begins with `init`), and returns a C++ `bool` value. The `free` method always takes no arguments and returns `void`.

The initialization method for your driver class should invoke the appropriate superclass initialization method before doing anything else, as shown in Listing 5-1 (page 57). If the superclass returns `false`, your class's initialization method should abort, release any allocated resources, and return `false`. Otherwise your class can perform its initialization and return `true`. When the libkern C++ runtime system creates an instance of a class, it zero-fills all of the member variables, so you don't need to explicitly initialize anything to zero, false, or null values.

Listing 5-1 Implementing an init method

```
bool MyDriver::init(IOPhysicalAddress * paddr)
{
    if (!super::init()) {
        // Perform any required clean-up, then return.
        return false;
    }
    physAddress = paddr;  // Set an instance variable.
    return true;
}
```

To create an instance using the initialization method, you write code such as this:

Listing 5-2 Creating an instance and calling its init method

```
MyDriver * pDrv = new MyDriver; // This invokes the predefined constructor
                                //  of MyDriver itself

if (!pDrv) {
    // Deal with error.
}

if (!pDrv->init(memAddress)) {
    // Deal with error.
    pDrv->release();    // Dispose of the driver object.
}
```

Because this makes creating instances more cumbersome, you may want to write a convenience method in the manner of many of the kernel C++ classes, as for example:

```
MyDriver * MyDriver::withAddress(IOPhysicalAddress *paddr)
{
    MyDriver * pDrv = new MyDriver;


    if (pDrv && !pDrv->init(paddr)) {
        pDrv->release();
        return 0;
    }
    return pDrv;
}
```

Using this convenience method, you can create an instance of your driver with code like the following:

```
MyDriver * pDrv = MyDriver::withAddress(paddr);

if (!pDrv) {
    // Deal with error of not being able to create driver object.
}
else {
    // Go on after successful creation of driver object.
}
```

A class's `free` method should release any resources held by the instance and then invoke the superclass's `free` method, as in Listing 5-3 (page 58):

Listing 5-3 Implementing the free function

```
void MyDriver::free(void)
{
    deviceRegisterMap->release();
    super::free();
    return;
}
```

Again, note that your code should never invoke `free` or the `delete` operator directly with objects based on the OSObject class. Always call `release` on such objects to dispose of them.

The I/O Kit Base Classes

All driver objects based on the I/O Kit inherit from the two base classes IORegistryEntry and IOService. The second of these classes, IOService, directly inherits from IORegistryEntry and all driver objects ultimately inherit from IOService. The IORegistryEntry class defines a driver object as a node in the I/O Registry, and IOService defines the life cycle of a driver object as well as implementing other behavior common to drivers.

The close inheritance relationship between IORegistryEntry and IOService might invite speculation as to why these classes weren't designed as one class. The reason is performance. Having IORegistryEntry as a superclass of IOService is an optimization because, in terms of memory footprint, the IORegistryEntry object is much more lightweight.

Dynamic Driver Registration (IORegistryEntry)

An IORegistryEntry object defines a node (or entry) in the I/O Registry. As the chapter "The I/O Registry" (page 39) explains in detail, the I/O Registry is a dynamic database that captures the current graph of "live" driver objects, tracking the client/provider relationships among these objects and recording the properties that describe their personalities. The I/O Registry plays an essential role in the dynamic features of Mac OS X; when users add or remove hardware, the system uses the Registry in the driver-matching process and immediately updates it to reflect the new configuration of devices.

Each IORegistryEntry object has two dictionaries (that is, instances of OSDictionary) associated with it. One is the property table for the object, which is typically also a driver object. This property table is the matching dictionary that specifies one of the driver's personalities. (See "Driver Personalities and Matching Languages" (page 43) for information on personalities.) The other dictionary of an IORegistryEntry object is the plane dictionary, which specifies how the object is connected to other objects in the registry.

In addition to reflecting all client/provider relationships among driver objects, the I/O Registry identifies subsets of these relationships. Both the totality of the Registry tree and the subsets of it are called **planes**. Each plane expresses a different provider/client relationship between objects in the I/O Registry by showing only those connections that exist in that relationship. Often the plane relationship is one of a dependency chain. The most general plane is the Service plane which displays the total hierarchy of registry entries. Every object in the Registry is a client of the services provided by its parent, so every object's connection to its ancestor in the Registry tree is visible on the Service plane. In addition to the Service plane, there are the Power, Audio, Device, FireWire, and USB planes. For more information on planes, see "I/O Registry Architecture and Construction" (page 39).

It is possible to have an IORegistryEntry object that is not also an IOService object. Such an object could be used purely for holding information associated with that node in the Registry. However, there is little actual need for such objects.

The IORegistryEntry class includes many member functions that driver objects might find useful; these functions fall into several categories:

- Property-table functions allow you to set, get, and remove properties of an IORegistryEntry object's property table as well as serializing property tables. Some `getProperty`functions perform a synchronized, recursive search through the Registry for the property of a given key.

- Positional functions let an IORegistryEntry object manipulate its position in the Registry tree. It can locate, identify, and attach to or detach from another IORegistryEntry object.
- Iteration functions enable your code to traverse the entire Registry tree, or a portion or it, and optionally invoke an "applier" callback function on IORegistryEntry objects encountered.

See the reference documentation for IORegistryEntry for details.

Basic Driver Behavior (IOService)

Every driver object in the I/O Kit is an instance of a class that ultimately inherits from the IOService class. IOService most importantly defines, through complementary pairs of virtual functions, a driver's life cycle within a dynamic runtime environment. It manages the matching and probing process, implements default matching behavior, and registers drivers and other services. But the IOService class also provides a wealth of functionality for many other purposes, including:

- Accessing a driver's provider, clients, state, and work loop
- Posting notifications and sending messages to other driver objects or services
- Managing power in devices
- Implementing user clients (device interfaces)
- Accessing device memory
- Registering and controlling interrupt handlers

This section first describes the life cycle of a driver object and IOService's role in that life cycle. Then it summarizes each of the other major IOService APIs.

Driver Object Life Cycle

An I/O Kit driver can be loaded and unloaded, or activated and deactivated, at any time. Every driver's life cycle follows the same pattern, as laid out in a set of functions defined by the standard driver superclass, IOService. Some of these functions must be implemented by the driver; others are implemented by IOService, but can be overridden by the driver for additional flexibility.

Figure 5-2 Driver object life-cycle functions

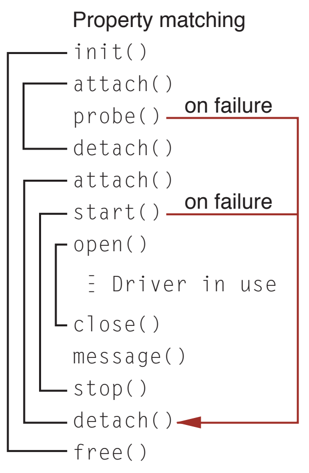


Figure 5-2 (page 61) shows the sequence of functions that gets invoked during the life of a driver object. The bracketing lines show how these functions are grouped into complementary pairs. A driver object class can override any of these functions, but must be sure to invoke the superclass's implementation of that same function at the appropriate point in its own implementation. For example, when you override the opening function of a complementary pair, such as `init` or `start`, your version must invoke the corresponding function of its superclass before doing its own initialization, as shown in Listing 5-1 (page 57). When you override a closing function, such as `free` or `stop`, you should perform your own cleanup before invoking the corresponding function in the superclass, as shown in Listing 5-3 (page 58).

Driver Matching and Loading

The first group of functions—`init`, `attach`, `probe`, and `detach`—is invoked during the process of driver matching and loading. This process occurs at boot time and at any time devices are added or removed. The following paragraphs summarize the matching process, paying special attention to the functions involved; see "Driver Matching and Loading" (page 46) for an extended discussion of the process.

The matching process is kicked off when a service provider detects a device. Usually this provider is the controller driver for a bus (such as a PCI bus), which detects the device by scanning its bus. The provider (usually through its family) then creates and registers any required nubs by calling the IOService function, `registerService`; this call, in turn, triggers the matching process.

As orchestrated by IOService, the I/O Kit finds and loads a driver for a nub in three distinct phases, using a subtractive process. In each phase, drivers that are *not* considered to be likely candidates for a match are subtracted from the total pool of possible candidates until a successful candidate is found. The phases are:

1. **Class matching**—the I/O Kit eliminates any drivers of the wrong provider (nub) class.

2. **Passive matching**—the I/O Kit examines the remaining drivers' personalities for family-specific properties.

3. **Active matching**—IOService calls each of the remaining drivers' `probe` functions with reference to the object the driver is being matched against. This function allows the driver to communicate with the device and verify that it can in fact drive that device. A probe score is returned that reflects how well suited the driver is to drive the device.

When a matching driver is found, its code is loaded and an instance of the principal class listed in the personality is created.

In the first two phases, none of the driver's life-cycle functions is called. It is only during the third stage of active matching, when the driver is asked to probe a device for suitability, that the first group of functions is invoked.

Whether a driver is loaded to drive a device or is merely asked to probe it, the first life-cycle function invoked is `init`, which is the libkern equivalent of the constructor function for the class. For I/O Kit drivers, this function takes as its sole argument an OSDictionary containing the matching properties from the personality in the XML file. The driver can use this to determine what specific personality it's been loaded for, determine the level of diagnostic output to produce, or otherwise establish basic operating parameters. However, I/O Kit drivers typically don't override the `init` function, performing their initialization in later stages, as described below. For more on `init`, and the related `free` function, see "Object Creation and Disposal (OSObject)" (page 53).

Before a driver object can either probe or start, it must be attached into the I/O Registry. To do this, the nub invokes the driver's `attach` function, which attaches the driver to the nub through the I/O Registry. The complementary function `detach` removes the driver from its nub. IOService gives both of these functions default implementations. A driver can override them, but rarely needs to do so.

If active matching is occurring, the nub next invokes the driver object's probe function. The probe function returns an IOService. This is usually the driver object itself, but the driver can return an instance of a different class, such as a specialized subclass included in the driver's kernel extension bundle. IOService's default implementation of probe simply returns the this pointer without altering the probe score. Overriding probe is optional; most drivers get enough information from property matching and don't need to override it. If you do override probe, however, you must make sure that the probe is not destructive, leaving the device in the state it found it. Hardware specifications generally define how to conduct non-destructive probes.

A driver's `start`function, just as with implementations of `probe`, should perform only the minimum necessary allocation of system resources to verify that it can operate the hardware. This conservative approach delays consumption of kernel resources until they're actually needed.

Each family, such as PCI, USB, or storage, defines a pair of activation and deactivation functions to indicate that the driver should prepare to service I/O requests and that the driver's services are no longer needed. These two functions are typically named `open` and `close`. Most drivers implement these functions to allocate and deallocate all of the necessary buffers and other structures in preparation for I/O processing.

Some families define additional levels of activation and deactivation. A networking driver, for example, does very little in `open` and `close`, instead performing setup and teardown in the `enable` and `disable` functions. Whatever the specific activation and deactivation functions, they can be invoked many times during a driver's life span; a driver should be able to function no matter how many times it gets activated or deactivated.

Driver Status Change

Another function that can be invoked many times during a driver's life span is the `message`function. This function informs the driver of important system status changes, such as when a disk is forcibly removed, when a power management change (sleep, wake-up) is occurring, or when the driver is being shut down. IOService's implementation of this function does nothing and returns an "unsupported" result code. For more on the notification and messaging functionality provided by IOService, see "Notification and Messaging" (page 63).

Driver Shutdown

When a driver is going to be permanently shut down, its `message` function is invoked with a terminate message (`kIOMessageServiceIsTerminated`). If the driver accepts the termination, its `stop` function is then invoked. The driver should implement its `stop`function to close, release, or free any resources it opened or created in its `start` function, and to leave the hardware in the state the driver originally found it. Assuming the driver implements the activation and deactivation functions, there is usually little to do in the `stop` function. The final stage of driver shutdown is invocation of `free`, which occurs when the driver object's reference count reaches zero. In this function the driver can dispose of any resources it created in its `init` function.

Provider Matching

If you are implementing a provider driver object (that is, a subclass that is a member of an I/O Kit family) you may want to override IOService's `matchPropertyTable` member function. When IOService performs matching for a driver object, it calls this method so the provider class can implement its own specific matching criteria in addition to the generic ones provided by IOService. The provider should examine the matching dictionary passed to see if it contains properties the family understands for matching and use them to match with the specific driver object if it does understand them.

Notification and Messaging

IOService provides two mechanisms for driver objects to communicate with each other and with the I/O Kit: notifications and messaging. Notifications are delivered to interested clients when a certain event occurs with an active service or driver that has properties matching a given dictionary. Messages are more targeted and flow in one direction, from provider to client. Any provider can send a message to any of its clients to notify it of some change in the runtime environment.

As discussed earlier in "Driver Object Life Cycle" (page 60), driver clients implement the `message`function to receive and respond to messages from their providers. This function allows them to adapt to changes in the runtime environment. The messages can inform them of changes in system status, such as changes in power state, suspension of service, or impending service terminations. Providers implement the `messageClient`(or `messageClients`) functions to send messages by invoking their client's `message` methods. The I/O Kit defines some messages while others may be defined by families. See the header file `Kernel.framework/Headers/IOKit/IOMessage.h` for the generic messages that the `messageClient` and `messageClients` functions can deliver to a driver.

The broadcasting of notifications is a bit more complicated. Any driver object can install a notification handler through the `addNotification` or `installNotification` functions. The notification handler is set up to be invoked when a specific driver object (identified by a dictionary of matching properties) experiences a specific type of state change, such as when a driver is first published, matched at any time, or is terminated. Each notification handler is also given a priority number in case multiple notifications of the same type and for the same object are triggered at the same time.

The notification handler (of type `IOServiceNotificationHandler`) is invoked if any driver object whose personality matches the supplied matching dictionary changes to the specified state. For example, when a service provider calls `registerServices`, that not only starts the registration process but it also delivers notifications to all registered clients interested in the publication of the provider. The notification request is identified by an instance of an IONotifier object, through which the notification can be enabled, disabled, or removed.

Driver Accessors

IOService includes, as a convenience, a number of accessor member functions giving quick access to a driver object's state and the objects that are closely related to it. These functions return the following objects or values:

- The driver's state (`getState`), a bitfield indicating whether the driver is inactive, registered, matched, and so on
- The work loop being used by the driver (`getWorkLoop`) (see "Handling Events" (page 73)) for further information)
- The driver's primary provider (`getProvider`), as well as an OSIterator object for iterating over the driver's providers, if multiple (for example, a RAID device)
- The driver's primary client (`getClient`), as well as an OSIterator object for iterating over the driver's clients, if multiple

Other IOService Features

IOService incorporates functionality (other than that summarized above) that is useful for many categories of device driver. Most notably, this functionality includes the following features:

- **User client**. The `newUserClient`function creates an IOUserClient-based connection for communication with a non-kernel client; the client invokes this function by calling the `IOServiceOpen`function of the I/O Kit framework.
- **Device memory**. Several IOService member functions get, map, and set the physical memory ranges allocated to a memory-mapped device. These functions are intended for driver objects that are clients of PCI devices.
- **Interrupt handling**. IOService provides low-level functions for registering, unregistering, manipulating, and accessing interrupt handlers that are called at primary-interrupt time for a device's interrupt. The functions provide a mechanism for installing interrupt handlers that is not based on a work loop.

I/O Kit Families

In the I/O Kit, families are collections of classes that define and implement the abstractions common to all devices of a particular type. They provide the programmatic interfaces and generic support code for developing drivers that are members (providers) or clients of such families.

This chapter describes a number of concepts related to I/O Kit families:

- The relation of drivers to families
- Families as libraries, including the versioning and loading of libraries
- The programmatic structure of families and naming conventions

In addition, this chapter offers some tips for those who want to write their own I/O Kit families. For a reference to the current set of the I/O Kit families provided by Apple, see the appendix "I/O Kit Family Reference" (page 117).

Drivers and Families

An I/O Kit family is a library that implements some bus protocol (for example, PCI or USB) or some common set of services. But the support that a family provides is generic. A family does not include any of the details for getting at hardware because it cannot make assumptions about the specific hardware under the general layer it represents. It's the driver writer's responsibility to write code that bridges between the concrete and the abstract—that is, between the hardware and the abstraction defined by the family. A driver must extend a family to support specific hardware or to acquire specific features.

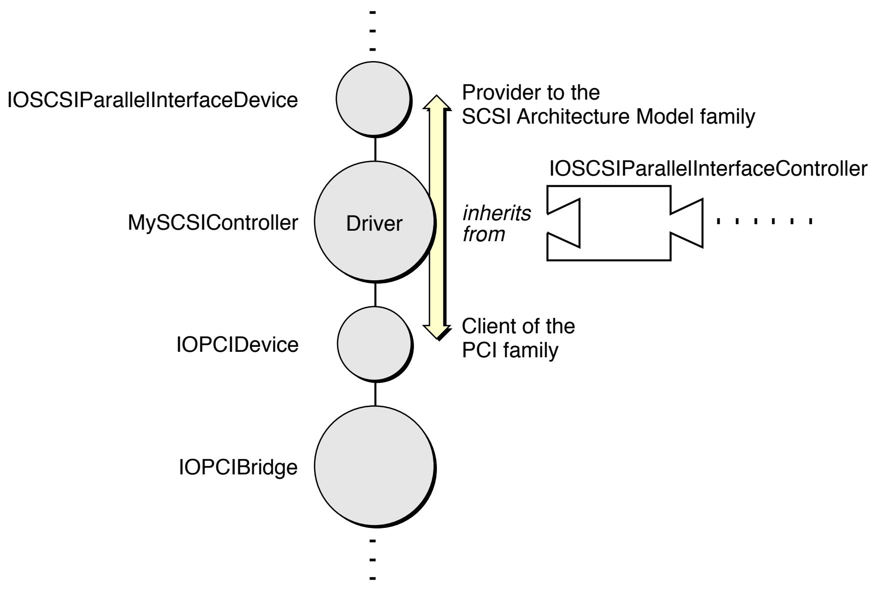
Take the SCSI Parallel family as an example. The SCSI Parallel family encapsulates the SCSI Parallel Interface-5 specification, which is well-defined. One of the things the specification describes is how to go about scanning the bus and detecting devices. Because this is an expensive operation, many SCSI Parallel controllers include firmware that can cache information about detected devices. To take advantage of this caching optimization, you could design your controller driver—member of the SCSI Parallel family—so that it overrides the scanning functionality to interact with the firmware.

Families commonly perform certain generic tasks, such as scanning buses, querying clients, queuing and validating commands, recovering from errors, and matching and loading drivers. Drivers do the tasks that impinge on hardware in some way. To continue with the example of the SCSI Parallel family, the primary job of the SCSI Parallel controller driver, as member of the SCSI Parallel family, is to receive SCSI commands from its family, execute each command on the hardware, and send a notification when the command completes.

Some I/O Kit families are clearly delimited by the specifications they encapsulate. Other families, such as the Audio family, are not as easily defined because there is no single specification prescribing what the family should include. In cases such as these, Apple carefully chose the set of abstractions to incorporate in the family to make it flexible and comprehensive enough. All families must advertise their capabilities and it is up to the higher levels of the driver stack to manage these capabilities.

A driver is both a provider and a client in its relationships to I/O Kit families. A driver that is a provider for a family (through its nub) is also a member of that family; it should inherit from a particular class in the family that describes the service it exports (however augmented). On the other hand, a driver is a client of the family whose service it imports (through a nub of the family). For example, a SCSI disk driver would inherit from the storage family rather than the SCSI Parallel family, to which it would be a client (see Figure 6-1 (page 66)). A USB mouse driver would inherit from the HID (Human Interface Devices) family and would be a client of the USB family. A PCI Audio card driver would inherit from the Audio family and would be a client of the PCI family.

Figure 6-1 A driver's relationships with I/O Kit families



Families As Libraries

Families are implemented as libraries packaged as kernel extensions (KEXTs). They specify their defining attributes in an information property list and are installed in `/System/Library/Extensions`. Families are, mechanically, little different than ordinary drivers.

Two related characteristics distinguish a family from a driver. First, a driver expresses a dependency on a family using the `OSBundleLibraries` property; second, a family is loaded only as a byproduct of a driver listing it as a library. A driver specifies the libraries on which it depends as elements of the `OSBundleLibraries` dictionary. The I/O Kit guarantees that these libraries will be loaded into the kernel before it loads the driver and links it with its families. Note that libraries themselves declare the libraries (kernel extensions and the kernel itself) on which they depend using the `OSBundleLibraries` property.

You specify a library as a key-value pair in the `OSBundleLibraries` dictionary where the key is the bundle identifier (`CFBundleIdentifier`) of the library and the value is the earliest version of the library that the driver is compatible with. All versions are expressed in the `'vers'` resource style. Listing 6-1 (page 67) gives an example from the information property list of the AppleUSBAudio driver.

Listing 6-1 The OSBundleLibraries property

```
<key>OSBundleLibraries</key>
    <dict>
        <key>com.apple.iokit.IOAudioFamily</key>
        <string>1.0.0</string>
        <key>com.apple.iokit.IOUSBFamily</key>
        <string>1.8</string>
    </dict>
```

Although the I/O Kit loads the libraries before it loads the driver that specifies these dependencies, and loads libraries in proper dependency order, there is no guarantee about the order in which it loads libraries that have no interdependencies.

Generally, developers should declare dependencies for their device driver or any other kernel extension. (If the KEXT doesn't have an executable, dependency declaration is unnecessary.) What dependencies they need to declare depends on which symbols need to get resolved. If you include a header file of a family or other library, or if a header indirectly ends up including a library, you should declare that dependency. If you unsure whether a dependency exists, declare it anyway.

Library Versioning

To be available for loading and linking into the kernel, a family or other library has to declare its compatibility information using two properties: `CFBundleVersion` and `OSBundleCompatibleVersion`. The `CFBundleVersion` property defines the forward limit of compatibility—that is, the current version. The `OSBundleCompatibleVersion` property defines the backward limit of compatibility by identifying the last `CFBundleVersion`-defined version of the library that broke binary compatibility with prior versions.

Every time you revise a driver or a family, you should increment your `CFBundleVersion` value appropriately. You reset the `OSBundleCompatibleVersion` value (to the current `CFBundleVersion`) only when the revision makes the binary incompatible with prior versions, as when you remove a function or other symbol, or change a class such that the vtable layout changes. If you are writing an I/O Kit family, make sure that you specify an `OSBundleCompatibleVersion` property for your library; otherwise, drivers and other kernel extensions cannot declare a dependency on it and thus cannot link against it.

For both drivers and families (and, indeed, all kernel extensions), make sure that you also set the version in the kernel module and that this value is equivalent to the `CFBundleVersion` in the information property list. You set the version in the executable through the `MODULE_VERSION` setting in Xcode, in the target's Customized Settings list (you find this in the target's Build view).

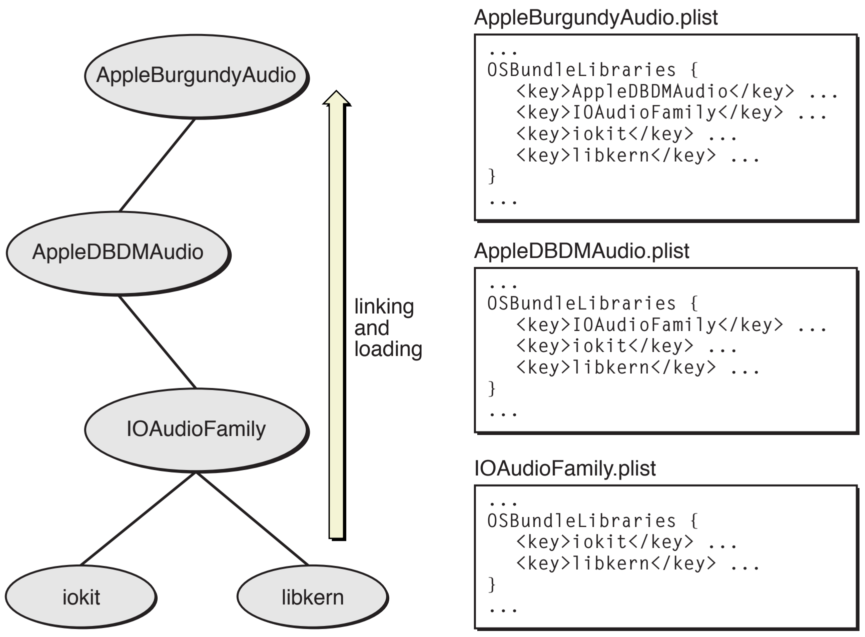
Library Loading

The KEXT manager functions as the kernel loader and linker. At boot time or whenever the system detects a newly attached device, the I/O Kit kicks off the matching process to find a suitable driver for a device. When such a driver is found, it is the KEXT manager's job to load the driver into the kernel and link it with the libraries on which it depends.

But before it can do this, the KEXT manager must ensure that those libraries, and all the other libraries on which those libraries depend, are loaded first. To do this, the manager builds a dependency tree of all libraries and other kernel modules required for the driver. It builds this tree using the contents of the `OSBundleLibraries` property, first of the driver and then of each required library.

After it builds the dependency tree, the KEXT manager checks if the libraries that are the most remote from the driver in the tree are already loaded. If any of these libraries is not loaded, the manager loads it and calls its start routine (the routine varies according to type of KEXT). It then proceeds up the dependency tree in similar fashion—linking, loading, and starting—until all required libraries have been linked and loaded. See Figure 6-2 (page 68) for an illustration of this procedure.

Figure 6-2 OSBundleLibraries and the dependency tree



AppleBurgundyAudio.plist

```
...
OSBundleLibraries {
   <key>AppleDBDMAudio</key> ...
   <key>IOAudioFamily</key> ...
   <key>iokit</key> ...
   <key>libkern</key> ...
}
...
```

AppleDBDMAudio.plist

```
...
OSBundleLibraries {
   <key>IOAudioFamily</key> ...
   <key>iokit</key> ...
   <key>libkern</key> ...
}
...
```

IOAudioFamily.plist

```
...
OSBundleLibraries {
   <key>iokit</key> ...
   <key>libkern</key> ...
}
...
```

If the KEXT manager encounters a problem initializing a library, or it doesn't find a library with a compatible version (based on the value of `OSBundleCompatibleVersion`), it stops and (usually) returns a failure code. The modules already loaded stay loaded for awhile. Generally, unloading of modules does not happen immediately when they are not used. The I/O Kit includes a feature that tracks idle time and unloads modules after a certain period of idleness.

Important: The only way to load a kernel extention explicitly is to use the `kextload` command-line utility.

The Programmatic Structure of Families

Although I/O Kit families tend to be quite different from each other, they have some structural elements in common. First, IOService is the common superclass for all I/O Kit families; at least one important class in each family, and possibly more, inherits from IOService (see "The I/O Kit Base Classes" (page 59) for more information). And each family has one or more classes that present an interface to drivers.

Typical Classes

A family typically defines two classes for drivers:

- A class describing the nub interface for drivers that are clients of the family
- A superclass for drivers that are members of the family, and thus providers to its nubs

In other words, I/O Kit families usually define an upward interface and a downward interface. These interfaces are required for the layering of driver objects involved in an I/O connection. The upward interface—the nub interface—presents to the rest of the system the hardware abstractions and rule definitions encapsulated by the family. The downward interface provides the subclassing interface for member drivers. Together, the interfaces define the up calls into the family and the down calls that member drivers are expected to make.

In addition to these two classes, families typically define a number of utility classes and support classes. The appendix "I/O Kit Family Reference" (page 117) describes some of these classes.

Some families specify subclasses for particular varieties of client or member drivers. The Storage family, for example, defines a generic block storage class for nub objects (IOBlockStorageDevice) and then also provides specific subclasses for certain varieties: IOCDBlockStorageDevice and IODVDBlockStorageDevice. In addition, families can include classes for device interfaces (as subclasses of IOUserClient) as well as commands specific to the family (as subclasses of IOCommand). Families can also have various helper classes and header files for family-specific type definitions.

Some families do not include a public nub or provider class for drivers when there is little need for such drivers. And Apple has not provided families for all types of hardware. If you find that the I/O Kit does not have a family or interface for your needs, you can always create a driver that inherits directly from IOService. Such "family-less" drivers are sometimes necessary if the potential applications for the driver are few. They must incorporate the abstractions and range of functionality found in families as well as the hardware-specific code typical of drivers. Besides directly inheriting from IOService, family-less drivers frequently make use of the I/O Kit helper classes such as IOWorkLoop, the event-source classes, IOMemoryCursor, and IOMemoryDescriptor.

Naming and Coding Conventions

Generally, Apple's position on class naming within families is that the name should indicate what the class represents. Often, this name is dictated by the specification for the hardware. For example, the PCI family defines the IOPCIBridge class for drivers that are providers for the family. The reason for this name is simple: the PCI bridge (as the specfication makes clear) is what the PCI controller drivers control. When there is no clear naming precedent for a family's classes, the I/O Kit follows a naming convention of IO*FamilyName*Device for nub (client) classes and IO*FamilyName*Controller for provider classes.

Important: The general guideline of naming classes for what they represent applies equally to drivers. Drivers should be named for the device they control (but should not have the redundant suffix "Driver").

If you are writing your own I/O Kit family, Apple recommends that you follow the same naming guidelines for your classes. And there are a few other general naming conventions to be aware of. Each class, function, type, and so on should have prefix that designates the vendor writing the software. Be sure not to use any of the prefixes that Apple reserves for itself (Table 6-1 (page 70)).

Table 6-1 API prefixes reserved by Apple

Prefix	Meaning
OS, os	libkern or other kernel service
IO, io	I/O Kit or I/O Kit family
MK, mk, mach_	Mach kernel
Apple, APPLE, apple, AAPL, aapl, com_apple_	Apple hardware support (for example, Apple-provided drivers)

In addition, private, internal symbols should have an underscore "_" prefix, following the convention used by Apple. Do not access these private APIs from a KEXT. As with drivers, use of reverse DNS notation (substituting underscores for periods) is highly recommended to avoid naming conflicts.

Creating An I/O Kit Family

There might be occasions when you deem it worthwhile to write your own I/O Kit family. Usually this happens when there is a standard or protocol for which no family exists, and you discern a need for interoperability among drivers for devices based on this protocol or standard. An example might be the IEEE488 standard for plotters and lab equipment.

If you decide to implement a family, here are a few guidelines to help you:

- At the beginning, write the family and driver code together; don't worry yet about the division of functionality and interface between driver and family. Just concentrate on coming up with a good object-oriented design, determining what objects are necessary and what relationships they should have.

- After you have a working driver and have solved the stack for a particular device, separate the family code from the hardware-specific code. One approach that might be useful for locating family-generic code, especially for complex families, is to write two or more drivers for different hardware and then abstract away the common code.
- Define what the family's nub objects look like to drivers—that is, the APIs your clients will see. To do this, look at the specification and encapsulate the important features (it's not necessary to include rarely or never-used features). Keep in mind that the nubs of most families do very little. Most often they encapsulate addressing and arbitration details.
- Define the superclass for drivers that will be members of your family.
- Keep the layering separation of a family airtight. A family should not include headers from any other family or driver and should not define the superclass of clients.

There can also be situations that might call for the creation of a "superfamily": a family that extends an existing family in a way similar to a subclass, but with a big difference; its aim is generality rather than specificity. Third-party vendors might want to have a superfamily to contain the code common to drivers based on different bus protocols. This would eliminate the need to load code that isn't needed. For example, a mouse vendor might have a driver capable of driving both USB and ADB mice. If a system requires a USB mouse, you don't want to have the ADB-specific code loaded as well. Thus the vendor might write a superfamily that acts as a service library; it would separate out the layers of code specific to a bus protocol into subfamilies and put the remaining code into the superfamily. Only the code specific to the currently used bus would be loaded.

Handling Events

A device driver works in perhaps the most chaotic of environments within an operating system. Many devices require a sequence of commands to perform a single I/O operation. However, multiple threads can enter the driver's code at any place and at any time, whether through client I/O requests, hardware interrupts, or timeout events; on a multiprocessor system, threads can even be running concurrently. The driver must be prepared to handle them all.

This flurry of incoming threads, each with its own event, poses some problems for a driver. The driver needs a way to protect its data structures from access by different threads because such simultaneous access can lead to data corruption, or worse. It needs to guarantee exclusive access to a thread for any single command or operation that must complete in order to preserve the integrity of the driver's data, or to prevent a deadlock or race condition. In the I/O Kit, this protection is provided by the IOWorkLoop class and its attendant event-source classes.

Work Loops

An IOWorkLoop object (or simply, a work loop) is primarily a gating mechanism that ensures single-threaded access to the data structures used by hardware. For some event contexts, a work loop is also a thread. In essence, a work loop is a mutually exclusive (mutex) lock associated with a thread. It does several things:

- Its gating mechanism synchronizes the actions among event sources.
- It provides a stackable environment for event handling.
- It spawns a dedicated thread for the completion of indirect interrupts delivered by the interrupt controller. This mechanism serializes interrupt handling for the work loop's driver, preventing simultaneous access to driver data by multiple interrupts.

To put the role of the work loop in perspective, it helps first to consider the event sources that it is designed for. In the I/O Kit there are five broad categories of asynchronous events:

- Interrupt events—indirect (secondary) interrupts originating from devices
- Timer events—events delivered periodically by timers, such as timeouts
- I/O commands—I/O requests issued by driver clients to their providers
- Power events—typically generated through calls down the driver stack
- Structural events—typically events involving the I/O Registry

The I/O Kit provides classes to handle these event sources: IOInterruptEventSource, IOTimerEventSource, and IOCommandGate. (You handle power and structural events using the mechanism provided by IOCommandGate objects.) Each of the event-source classes defines a mechanism specific to an event type for invoking a single function within the protected context of the work loop. If a thread carrying an event needs access to a driver's critical data, it must do so through an object of one of these classes.

Generally, client drivers set up their work loops, event sources, and event handlers in their `start` function. In order to avoid deadlocks and race conditions, all code that accesses the same data should share a single work loop, registering their event sources with it so that a single gating mechanism is used. Work loops can be safely shared among unrelated objects, of course, and often are shared by objects at different levels in a single driver stack. Work loops can also be dedicated for use by a particular driver and its clients. See "Shared and Dedicated Work Loops" (page 75) for more information.

The I/O Kit work-loop mechanism offers functionality roughly similar to that of the Vertical Retrace Manager, the Time Manager, and the Deferred Task Manager of Mac OS 9.

Work Loop Architecture

The I/O Kit's work-loop mechanism mitigates the performance penalty exacted by context switching, a by-product of the underlying event-handling model commonly used in some operating systems. To guarantee a single-threaded context for event handling, this model completes everything on one thread. Unfortunately, the transfer of work to the thread requires a switch in the context of the event-bearing thread. More precisely, when this thread goes from a running context to a non-running context, its current register state must be saved. When the secure thread completes its work, the state of the originating thread is restored and control branches back to the function originally referenced by the thread. This switching back and forth between thread contexts consumes cycles.

The work-loop model works quite differently for I/O commands and timer events. In these instances, the thread of the respective event source simply grabs the mutex lock held by the work loop. No other event from any source can be processed until the `Action` routine for the current event returns. Although the lock is mutually exclusive, it doesn't prevent reentrancy. Also, you can have multiple work loops in the same driver stack, and this increases the possibility of deadlock. However, work loops do avoid self-deadlocks because they are based on a recursive lock: They always check to see if they are the thread that currently own the lock.

The way the I/O Kit manages interrupt event sources does involve context switching. The completion routines for interrupts run on the work loop's thread and not on the thread delivering the interrupt. Context switching is required in this case because the interrupt controller must immediately dispatch direct (primary) interrupts to other threads to run the completion routines for those interrupts. See "Handling Interrupts" (page 78) for more information.

Two factors influence the order in which a work loop queries its event sources for work. The relative priority of threads is the main determinant, with timers having the highest priority. A client thread can modify its own priority and thereby expedite the delivery of I/O requests (it might not affect how soon they are processed, however, because I/O requests are usually queued in FIFO order). For interrupt event sources, which also have a relatively high priority, the order in which they are added to the work loop determines the order in which they are queried for work. See "Handling Interrupts" (page 78) for further details.

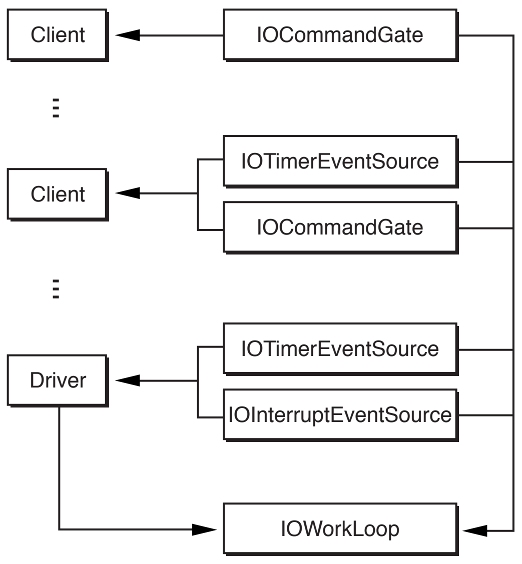
Regardless of event source and mechanism, a work loop is primarily used for one thing: to run the completion or `Action` routines specified by the event source. It guarantees that the routine handling an event is the only one running at any given time. This aspect of work loops raises a design point. When a thread is running code to handle an event, other events can be asynchronously delivered to their event sources, but they cannot be processed until the handler returns. Therefore event handlers should not attempt to complete large chunks of work or do anything that might *block* (that is, wait for some other process to complete), such as allocating memory or other resources. Instead they should, if possible, queue up the work or otherwise defer it for later processing.

Shared and Dedicated Work Loops

All I/O Kit services can easily share their provider's work loop. The base of the driver Registry, representing the logic board of the computer, always contains a work loop, so a driver is assured of having a work loop even if it doesn't create one itself. All a driver needs to do is call the IOService function `getWorkLoop` to access its provider's work loop.

In this way, an entire stack of driver objects, or a subset of such objects, can share one work loop. Figure 7-1 (page 75) shows how a work loop shared by multiple driver objects uses event sources to manage access to its gating mechanism.

Figure 7-1 Driver objects sharing a work loop



Most drivers won't create their own work loop. If hardware doesn't directly raise interrupts in your driver, or if interrupts rarely occur in your driver, then you don't need your own work loop. However, a driver that takes direct interrupts—in other words, that interacts directly with the interrupt controller—should create its own dedicated work loop. Examples of such drivers are PCI controller drivers (or any similar driver with a provider class of IOPCIDevice) and RAID controller drivers. Even these work loops may be shared by the driver's clients, however, so it's important to realize that in either case, the driver must not assume that it has exclusive use of the work loop. This means that a driver should rarely enable or disable all events on its work loop, since doing so may affect other I/O Kit services using the work loop.

If a driver handles interrupts or for some other reason needs its own work loop, it should override the IOService function `getWorkLoop` to create a dedicated work loop, used by just the driver and its clients. If `getWorkLoop` isn't overridden, a driver object gets the next work loop down in its stack.

Examples of Obtaining Work Loops

To obtain a work loop for your client driver, you should usually use your provider's work loop or, if necessary, create your own. To obtain your provider's work loop, all you have to do is call the IOService function `getWorkLoop` and retain the returned object. Immediately after getting your work loop you should create your event sources and add them to the work loop (making sure they are enabled).

To create a dedicated work loop for your driver, override the `getWorkLoop` function. Listing 7-1 (page 76) illustrates a thread-safe implementation of `getWorkLoop` that creates the work loop lazily and safely.

Listing 7-1 Creating a dedicated work loop

```
protected:
    IOWorkLoop *cntrlSync;/* Controllers Synchronising context */
// ...
IOWorkLoop * AppleDeviceDriver::getWorkLoop()
{
    // Do we have a work loop already?, if so return it NOW.
    if ((vm_address_t) cntrlSync >> 1)
        return cntrlSync;

    if (OSCompareAndSwap(0, 1, (UInt32 *) &cntrlSync)) {
        // Construct the workloop and set the cntrlSync variable
        // to whatever the result is and return
        cntrlSync = IOWorkLoop::workLoop();
    }
    else while ((IOWorkLoop *) cntrlSync == (IOWorkLoop *) 1)
        // Spin around the cntrlSync variable until the
        // initialization finishes.
        thread_block(0);

    return cntrlSync;
}
```

This code first checks if `cntrlSync` is a valid memory address; if it is, a work loop already exists, so the code returns it. Then it tests to see if some other thread is trying to create a work loop by atomically trying to compare and swap the controller synchronizer variable from 0 to 1 (1 cannot be a valid address for a work loop). If no swap occurred, then some other thread is initializing the work loop and so the function waits for the `cntrlSync` variable to stop being 1. If the swap occurred then no work loop exists and no other thread is in the process of creating one. In this case, the function creates and returns the work loop, which unblocks any other threads that might be waiting.

As you would when getting a shared work loop, invoke `getWorkLoop` in `start` to get your work-loop object (and then retain it). After creating and initializing a work loop, you must create and add your event sources to it. See the following section for more on event sources in the I/O Kit.

Event Sources

A work loop can have any number of event sources added to it. An event source is an object that corresponds to a type of event that a device driver can be expected to handle; there are currently event sources for hardware interrupts, timer events, and I/O commands. The I/O Kit defines a class for each of these event types: IOInterruptEventSource, IOTimerEventSource, and IOCommandGate, respectively. Each of these classes directly inherits from the abstract class IOEventSource.

An event-source object acts as a queue for events arriving from a particular event source and hands off those events to the work-loop context when it asks them for work. When you create an event-source object, you specify a callback function (also known as an "action" function) to be invoked to handle the event. Similar to the Cocoa environment's target/action mechanism, the I/O Kit stores as instance variables in an event source the target of the event (the driver object, usually) and the action to perform. The handler's signature must conform to an `Action` prototype declared in the header file of the event-source class. As required, the work loop asks each of its event sources in turn (by invoking their `checkForWork` function) for events to process. If an event source has a queued event, the work loop runs the handler code for that event in its own protected context. Note that when you register an event source with a work loop, the event source is provided with the work loop's signaling semaphore, which it uses to wake the work loop. (For more information on how the work loop sleeps and wakes, see the `threadMain` function in IOWorkLoop documentation.)

A client driver, in its activation phase (usually the `start` function), creates the event sources it needs and adds them to its work loop. The driver must also implement an event handler for each event source, ensuring that the function's signature conforms to the `Action` function prototype defined for the event-source class. For performance reasons, the event handler should avoid doing anything that might block (such as allocating memory) and defer processing of large amounts of data. See "Work Loops" (page 73) for further information on event priority and deferring work in event handlers.

The procedure for adding event sources to a work loop is similar for each type of event source. It involves four simple steps:

1. Obtain your work loop.
2. Create the event-source object.
3. Add the object to the work loop.
4. Enable the event source.

Disposing of an event source also has a common procedural pattern:

1. Disable the event source.
2. Remove it from the work loop.
3. Release the event source.

The following sections discuss the particulars of each event source and give examples specific to each kind.

Handling Interrupts

Interrupts are typically the most important type of event that drivers handle. They are the way that devices attached to a computer inform the operating system that an asynchronous action has occurred and that, consequently, they have some data. For example, when the user moves a mouse or plugs a Zip drive into a USB port, a hardware interrupt is generated and the affected driver is notified of this event. This section discusses interrupt handling in the I/O Kit, with particular attention to the role played by objects of IOInterruptEventSource and its subclasses.

Interrupt Handling in the I/O Kit

The I/O Kit's model for interrupt handling does not conform to the standard UNIX model. I/O Kit drivers nearly always work in the indirect-interrupt context instead of dealing with direct interrupts, as does the UNIX model. Indirect interrupts are less restrictive and permit the Mach scheduler to do its job. (Indirect interrupts are sometimes known as secondary interrupts and direct interrupts as primary interrupts.) The difference between the two types of interrupts has to do with the context in which the interrupt is dealt with.

Two types of events trigger an interrupt:

- Command-based events, such as incoming networking packets and reads of storage media
- Asynchronous events, such as keyboard presses

When an interrupt occurs, a specific interrupt line is set and, once the interrupted thread finishes the current instruction, control branches to the interrupt controller registered with the Platform Expert. When the interrupt controller receives the interrupt, its thread becomes that of the direct (primary) interrupt. There is typically only one direct interrupt in the system at any one time, and the direct-interrupt context has the highest priority in the system. The following list indicates the relative priorities of threads in the system:

1. Direct interrupt
2. Timers and page-out
3. Real time (multimedia)
4. Indirect interrupts (drivers)
5. Window Manager
6. User threads (including I/O requests)

Because of its extremely high priority, the direct-interrupt context has a design responsibility to hand off the interrupt to lower-priority threads as soon as possible. The interrupt controller must decode why the interrupt was taken, assign it to the appropriate driver object, and return.

In the direct-interrupt model, the target driver assumes the context carrying the direct interrupt. It must handle the interrupt in this highest-priority context. The problem with direct interrupts is that they can be neither lowered in priority nor preempted. All other interrupts are effectively disabled until the current interrupt is handled. Direct interrupts especially don't scale well in the Mac OS X multiprocessing environment.

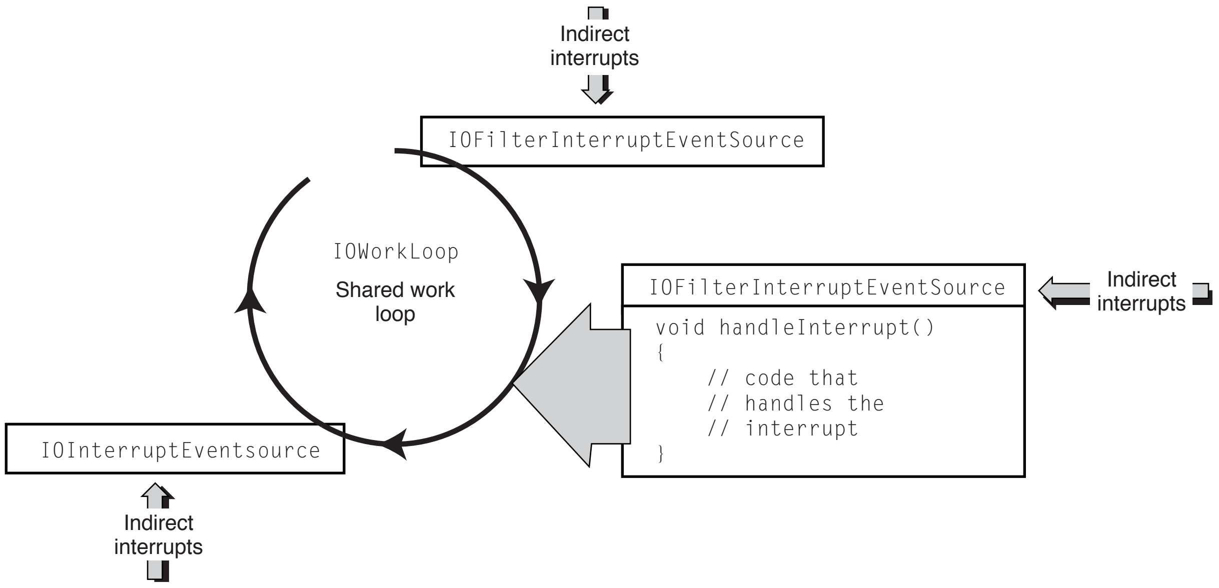
With indirect interrupts, the interrupt controller dispatches the interrupt it reads off the interrupt line to the appropriate interrupt event-source object of the target driver, effectively causing it to schedule on the driver's work-loop thread. The completion (or `Action`) routine defined by the event source is then run on the work-loop thread to handle the interrupt. The priority of the work-loop thread, although high compared to most client threads, is lower than the thread carrying the direct interrupt. Thus the completion routine running in the work-loop thread can be preempted by another direct interrupt.

The I/O Kit does not prohibit access to the direct-interrupt context, and in fact provides a separate programming interface for this purpose (see "Using Interrupt Handlers With No Work Loops" (page 82)). However, use of direct interrupts is strongly discouraged.

A work loop can have several IOInterruptEventSource objects attached to it. The order in which these objects are added to the work loop (through IOWorkLoop's `addEventSource` function) determines the general order in which interrupts from different sources are handled.

Figure 7-2 (page 79) illustrates some of these concepts. It shows events originating from different sources being delivered to the corresponding event-source objects "attached" to the work loop. As with any event-source object, each interrupt event source acts as a queue for events of that type; when there is an event in the queue, the object signals the work loop that it has work for it. The work loop (that is, the dedicated thread) awakes and queries each installed event source in turn. If an event source has work, the work loop runs the completion routine for the event (in this case, an interrupt) in its own protected thread. The previous thread—the client thread running the event-source code—is blocked until the routine finishes processing the event. Then the work loop moves to the next interrupt event source and, if there is work, runs the completion routine for that interrupt in its protected context. When there is no more work to do, the work loop sleeps.

Figure 7-2 A work loop and its event sources



Remember that the order in which you add interrupt event sources to a work loop determines the order of handling for specific interrupt events.

Setting Up an Interrupt Handler Attached to a Work Loop

A driver typically creates an interrupt event-source object—generally of the IOInterruptEventSource or IOFilterInterruptEventSource class—in its `start` function by calling the factory creation method for the class (for example, `interruptEventSource`). This method specifies the driver itself as a target and identifies an action member function (conforming to the `Action` type defined for the event-source class) to be invoked as the completion routine for the event source. The factory method also associates the driver with a provider that deals with the hardware interrupt facility (usually a nub such as an IOPCIDevice). The driver then registers the event source with the work loop through IOWorkLoop's `addEventSource` function.

Listing 7-2 (page 80) provides an example for setting up an interrupt event source.

Listing 7-2 Adding an interrupt event source to a work loop

```
myWorkLoop = (IOWorkLoop *)getWorkLoop();

interruptSource = IOInterruptEventSource::interruptEventSource(this,
    (IOInterruptEventAction)&MyDriver::interruptOccurred,
    provider);

if (!interruptSource) {
    IOLog("%s: Failed to create interrupt event source!\n", getName());
    // Handle error (typically by returning a failure result).
    }

if (myWorkLoop->addEventSource(interruptSource) != kIOReturnSuccess) {
    IOLog("%s: Failed to add interrupt event source to work loop!\n",
        getName());
    // Handle error (typically by returning a failure result).
}
```

In this example, if you do not specify a provider in the `interruptEventSource` call, the event source assumes that the client will call IOInterruptEventSource's `interruptOccurred` method explicitly. Invocation of this function causes the safe delivery of asynchronous events to the driver's IOInterruptEventSource.

Events originating from direct interrupts are handled within the work loop's thread, which should never block indefinitely. This specifically means that the completion routines that handle interrupts, and any function they invoke, must not allocate memory or create objects, as allocation can block for unbounded periods of time.

You destroy an interrupt event source in a driver's deactivation function (usually `stop`). Before you release the IOInterruptEventSource object, you should disable it and then remove it from the work loop. Listing 7-3 (page 80) gives an example of how to do this.

Listing 7-3 Disposing of an IOInterruptEventSource

```
if (interruptSource) {
    interruptSource->disable();
    myWorkLoop->removeEventSource(interruptSource);
    interruptSource->release();
    interruptSource = 0;
}
```

Filter Interrupt Event Sources

The I/O Kit supports shared interrupts, where drivers share a single interrupt line. For this purpose it defines the IOFilterInterruptEventSource class, a subclass of IOInterruptEventSource. Apple highly recommends that third-party device driver writers base their interrupt event sources on the IOFilterInterruptEventSource class instead of the IOInterruptEventSource class. The latter class does not ensure that the sharing of interrupt lines is safe.

The IOFilterInterruptEventSource class follows the same model as its superclass except that it defines, in addition to the `Action` completion routine, a special callback function. When an interrupt occurs the interrupt invokes this function for each driver sharing the interrupt line. In this function, the driver responds by indicating whether the interrupt is something that it should handle.

Listing 7-4 (page 81) shows how to set up and use an IOFilterInterruptEventSource.

Listing 7-4 Setting up an IOFilterInterruptEventSource

```
bool myDriver::start(IOService * provider)
{
    // stuff happens here

    IOWorkLoop * myWorkLoop = (IOWorkLoop *) getWorkLoop();
    if (!myWorkLoop)
        return false;

    // Create and register an interrupt event source. The provider will
    // take care of the low-level interrupt registration stuff.
    //
    interruptSrc =
        IOFilterInterruptEventSource::filterInterruptEventSource(this,
                    (IOInterruptEventAction) &myDriver::interruptOccurred,
                    (IOFilterInterruptAction) &myDriver::checkForInterrupt,
                    provider);
    if (myWorkLoop->addEventSource(interruptSrc) != kIOReturnSuccess) {
        IOLog("%s: Failed to add FIES to work loop.\n", getName());
    }
    // and more stuff here...
}


bool myDriver::checkForInterrupt(IOFilterInterruptEventSource * src)
{
    // check if this interrupt belongs to me

    return true; // go ahead and invoke completion routine
}


void myDriver::interruptOccurred(IOInterruptEventSource * src, int cnt)
{
    // handle the interrupt
}
```

If your filter routine (the `checkForInterrupt` routine in Listing 7-4 (page 81)) returns `true`, the I/O Kit will automatically start your interrupt handler routine on your work loop. The interrupt will remain disabled in hardware until your interrupt service routine (`interruptOccurred` in Listing 7-4 (page 81)) completes.

Note: In some cases, such as the implementation of pseudo-DMA, this behavior may not be desirable. In this case, you may choose to have your filter routine schedule the work on the work loop itself and then return `false`. If you do this, the interrupt will not be disabled in hardware and you could receive additional primary interrupts before your work loop–level service routine completes. Because this scheme has implications for synchronization between your filter routine and your interrupt service routine, you should avoid doing this unless your driver requires pseudo-DMA.

Using Interrupt Handlers With No Work Loops

The IOService class provides member functions for registering interrupt handlers that operate outside of the work-loop mechanism. These handlers can be invoked in a direct interrupt context and must call the interrupt management code of a provider such as an IOPCIDevice nub. Only one handler can be installed per interrupt source. It must be prepared to create and run its own threads and do its own locking.

Few drivers need to use interrupt handlers that are created and controlled in this way. One example where such an interrupt handler is justified is a multifunction card that needs to route direct interrupts to drivers. If you take this course, be careful. Very few system APIs are safe to call in the direct-interrupt context.

More information on this subject is forthcoming.

Handling Timer Events

Device drivers occasionally need to set timers, usually to implement a timeout so the driver can determine if an I/O request doesn't complete within a reasonable period. The IOTimerEventSource class is designed for that purpose.

Important: The absolute accuracy of timeouts in the I/O Kit cannot be guaranteed. The Mach scheduler can always run a higher priority thread, which might delay the execution of the timer `Action` routine.

A driver creates an IOTimerEventSource with a callback `Action` function and a time at which to invoke that function, and then registers it with the work loop to run on. When the timeout passes, the event source is scheduled with the work loop. When the work loop queries it for work, the event source closes the work-loop gate (by taking the work loop's lock), invokes the callback function, and then releases the work-loop lock to open the gate. Listing 7-5 (page 82) show how to create and register a timer event source.

Listing 7-5 Creating and registering a timer event source

```
myWorkLoop = (IOWorkLoop *)getWorkLoop();

timerSource = IOTimerEventSource::timerEventSource(this,
    (IOTimerEventSource::Action)&MyDriver::timeoutOccurred);

if (!timerSource) {
    IOLog("%s: Failed to create timer event source!\n", getName());
    // Handle error (typically by returning a failure result).
    }
```

```
if (myWorkLoop->addEventSource(timerSource) != kIOReturnSuccess) {
    IOLog("%s: Failed to add timer event source to work loop!\n", getName());
    // Handle error (typically by returning a failure result).
}

timerSource->setTimeoutMS(MYDRIVER_TIMEOUT);
```

Often a driver wants to set the timer and issue an I/O request at the same time. If the I/O request completes before the timer event is triggered, the driver should cancel the timer immediately. If a timer event is triggered first, the driver typically reissues the time-out I/O request (at which time it resets the timer).

If you want the timer event to be recurrent, you should reset the timer to the desired interval in the `Action` handler. The IOTimerEventSource class does not have a mechanism for setting periodic timers. The class does provide a few functions for setting relative and absolute timer intervals at various granularities (nanoseconds, microseconds, and so on). The code fragment in Listing 7-5 (page 82) uses `setTimeoutMS` to set the timer with a specific time-out millisecond interval.

Events originating from timers are handled by the driver's `Action` routine. As with other event handlers, this routine should never block indefinitely. This specifically means that timer handlers, and any function they invoke, must not allocate memory or create objects, as allocation can block for unbounded periods of time.

To dispose of a timer event source, you should cancel the pending timer event before removing the event source from the work loop and releasing it. Listing 7-6 (page 83) illustrates how you might do this in your driver's deactivation function.

Listing 7-6 Disposing of a timer event source

```
if (timerSource) {
    timerSource->cancelTimeout();
    myWorkLoop->removeEventSource(timerSource);
    timerSource->release();
    timerSource = 0;
}
```

I/O Requests and Command Gates

Driver clients use IOCommandGate objects to issue I/O requests to a driver. A command gate controls access to the work-loop lock, and in this way it serializes access to the data involved in I/O requests. It does not require a thread context switch to ensure single-threaded access. An IOCommandGate event source simply takes the work-loop lock before it runs its `Action` routine; by doing so, it prevents other event sources on the same work loop from scheduling. This makes it an efficient mechanism for I/O transfers.

Note that nub classes usually define `Action` functions for their own clients to use, so that driver classes don't have to use command gates themselves.

Up Calls and Down Calls

Calls originated by clients through a command gate are known as down calls. These always originate in a thread other than the work loop's context, and so they may safely block without causing a deadlock (as long as they don't hold the work-loop gate). All allocation should occur on the down-call side of an I/O request before the command gate is closed.

Up calls, which are originated by an interrupt or timer event, occur within the work loop's context and should never block indefinitely. This specifically means that interrupt and timeout handlers, and any function they invoke, must not allocate memory or create objects, as allocation can block and, as a potential consequence, cause a paging deadlock.

It's possible for an up call to result in a client notification that immediately results in another I/O request through the command gate. A work loop can handle recursive closing of its gate by the same thread, so this situation never results in deadlock. However, because the new request is occurring on the context of an up call, that request cannot block; this concern belongs to the system client making the I/O request, though, so you need never worry about this as a driver developer.

Setting Up and Using Command Gates

Prior to closing a command gate, you should adequately prepare the I/O request. An I/O request involves three things: the command itself (which is family-specific), the memory involved in the transfer (defined as an IOMemoryDescriptor object), and the function to call to process the request within the context of the command gate. See "Managing Data" (page 87) for information on IOMemoryDescriptors and related objects.

Command gates should be closed for the briefest possible period, during which the least amount of work possible is performed. The longer a command gate holds the work-loop lock, the greater the likelihood of contention. As with all event sources, the command-gate function should not allocate memory or any other unbounded resource because of the danger of blocking. Instead, the client should preallocate the required resources before control is transferred to the work-loop context. For example, it could allocate a pool of resources in its `start` function.

You create an IOCommandGate object by calling the `commandGate` factory method, specifying as parameters the object "owner" of the event source (usually `this`) and a pointer to a function conforming to the `Action` prototype. You then register the command gate with the client's work loop using IOWorkLoop's `addEventSource` function. Listing 7-7 (page 84) gives an example of this procedure.

Listing 7-7 Creating and registering a command gate

```
workLoop = (IOWorkLoop *)getWorkLoop();
commandGate = IOCommandGate::commandGate(this,
                (IOCommandGate::Action)receiveMsg);
if (!commandGate ||
    (workLoop->addEventSource(commandGate) != kIOReturnSuccess) ) {
    kprintf("can't create or add commandGate\n");
    return false;
}
```

The IOCommandGate class provides two alternatives for initiating the execution of an I/O request within the command gate. One is the `runCommand` member function and the other is the `runAction` member function. These functions work similarly. When the client wishes to invoke the `Action` function, rather than invoking it directly, it invokes the command gate's `runCommand` or `runAction` function, passing in all required arguments. The command gate then grabs the work-loop lock (that is, it closes the command gate), invokes the `Action` function, and then opens the gate.

Where the two functions differ is in their flexibility. The `runCommand` function makes use of the same target/action mechanism used by the other event-source classes. In this mechanism, the created IOCommandGate object encapsulates (a pointer to) an `Action` function as well as the target (or "owner") object that implements this function. In this model, only one `Action` function can be invoked for an I/O request.

However, a driver often has to deal with multiple sources of I/O requests. If this is the case, you can use the `runAction` function to issue I/O requests in multiple command gates. This function lets you define the function to be called within the command-gate context; you must specify a pointer to this function as the first parameter.

Important: Do not call the `runAction` or `runCommand` function from interrupt context.

Listing 7-8 (page 85) illustrates one the use of the `runCommand` function to issue an I/O request.

Listing 7-8 Issuing an I/O request through the command gate

```
void ApplePMU::enqueueCommand ( PMUrequest * request )
{
    commandGate->runCommand(request);
}

void receiveMsg ( OSObject * theDriver, void * newRequest, void *, void *, void
 * )
{
    ApplePMU * PMUdriver = (ApplePMU *) theDriver;
    PMUrequest * theRequest = (PMUrequest*)newRequest;

    // Inserts the request in the queue:
    theRequest->prev = PMUdriver->queueTail;
    theRequest->next = NULL;
    if ( PMUdriver->queueTail != NULL ) {
        PMUdriver->queueTail->next = theRequest;
    }
    else {
        PMUdriver->queueHead = theRequest;
    }
    PMUdriver->queueTail =  theRequest;

    // If we can, we process the next request in the queue:
    if ( (PMUdriver->PGE_ISR_state == kPMUidle) && !PMUdriver->adb_reading) {
        PMUdriver->CheckRequestQueue();
    }
}
```

In this example, the `runCommand` function is used to indirectly invoke the command gate's `Action` function, `receiveMsg`. One important tactic that this example shows is how to defer processing I/O requests—when allocation of memory and other resources might be necessary—until no more events are queued at the command gate. The `receiveMsg` function queues up each incoming request and, if no more requests are pending, calls a `CheckRequestQueue` function to do the actual I/O work.

A typical procedure is to set a timeout (using an IOTimerEventSource object) at the same time you issue an I/O request. If the I/O request does not complete within a reasonable period, the timer is triggered, giving you the opportunity to correct any problem (if possible) and reissue the I/O request. If the I/O request is successful, remember to disable the timer. See "Handling Timer Events" (page 82) for details on using IOTimerEventSources.

You destroy an command-gate event source in a driver's deactivation function (usually `stop`). Before you release the IOCommandGate object, you should remove it from the work loop. Listing 7-9 (page 86) gives an example of how to do this.

Listing 7-9 Disposing of an IOCommandGate

```
if (commandGate) {
    myWorkLoop->removeEventSource(commandGate);
    commandGate->release();
    commandGate = 0;
}
```

Completion Chaining

Occasionally, the driver writer must deal with constraints imposed by hardware, such as a maximum byte size for requests that are at odds with what the driver's client expects. For example, the hardware may transfer data in 64-kilobyte chunks, but the driver's client expects to transfer data in 128-byte buffers.

For situations such as this, the driver writer can do completion chaining. In completion chaining, one I/O request is used to trigger another I/O request asynchronously. Completion chaining is a way to break up an original I/O request into a series of smaller requests in response to the constraints of hardware. For example, the first requests can be to read the data, the second to modify it appropriately, and the third to write the data in the expected size to the provider and return, as expected, to the originator of the request. Each leg of the compound request is asynchronous; it issues the next request upon completion of the current request.

It generally works like this. A driver's command gate initiates an I/O request. The request carries a pointer to the new completion routine, implemented in your driver, along with an opaque context structure known only to your driver. The original completion information from your driver's client is also saved away inside this context. When the request completes, your completion routine is called in place of the original one, with your context passed in. Your completion routine would submit the next leg of the request, if any; otherwise, it would invoke the original completion when the compound request completes. Completion chaining thus permits a linked list of I/O requests that are scheduled one after another, entirely asynchronously.

For an example of completion chaining, look at the implementation of the IOBlockStorageDriver class, especially the `deblockRequestCompletion` and `deblockRequest` methods.You can find the implementation of this class in the IOStorageFamily project in the Darwin Open Source project.

Managing Data

A driver's essential work is to shuffle data in and out of the system in response to client requests as well as events such as hardware-generated interrupts. The I/O Kit defines standard mechanisms for drivers to do this using a handful of classes. Drivers use the IOMemoryDescriptor and IOMemoryCursor classes to perform pre-I/O and post-I/O processing on data buffers, such as translating them between client formats and hardware-specific formats. This chapter discusses these classes and various issues related to device drivers and data management.

Drivers handle client requests and other events using the IOWorkLoop and IOEventSource classes to serialize access and thereby protect their critical data. Because of these mechanisms, drivers rarely have to worry about such issues as protecting critical data or disabling and enabling interrupts during the normal course of handling a request. See the chapter "Handling Events" (page 73) for information.

Handling I/O Transfers

An I/O transfer is little more than a movement of data between one or more buffers in system memory and a device. "System memory" in this context, however, refers to actual physical memory, not the virtual memory address space used by both user and kernel tasks in Mac OS X. Because I/O transfers at the level of device drivers are sensitive to the limitations of hardware, and hardware can "see" only physical memory, they require special treatment.

Input/output operations in Mac OS X occur within the context of a virtual memory system with preemptive scheduling of tasks and threads. In this context, a data buffer with a stable virtual address can be located anywhere in physical memory, and that physical memory location can change as virtual memory is paged in and out. It's possible for that data buffer to not be in physical memory at all at any given time. Even kernel memory, which isn't subject to relocation, is accessed by the CPU in a virtual address space.

For a write operation, where data in the system is being sent out, a data buffer must be paged in if necessary from the virtual memory store. For a read operation, where a buffer will be filled by data brought into the system, the existing contents of the data buffer are irrelevant, so no page-in is necessary; a new page is simply allocated in physical memory, with the previous contents of that page being overwritten (after being paged out, if necessary).

Yet a requirement for I/O transfers is that the data in system memory not be relocated during the duration of the transfer. To guarantee that a device can access the data in buffers, the buffers must be resident in physical memory and must be wired down so that they don't get paged out or relocated.

Then the physical addresses of the buffers must be made available to the device. After the device is finished with the buffers, they must be unwired so that they can once again be paged out by the virtual memory system.

To help you deal with these and other constraints of hardware, the I/O Kit puts several classes at your disposal.

Memory Descriptors and Memory Cursors

In a preemptive multitasking operating system with built-in virtual memory, I/O transfers require special preparation and post-completion processing:

- The space needed for an I/O transfer must reside in physical memory and must be wired down so it can't be paged out until the transfer completes.
- The virtual memory addresses used by software must be converted to physical addresses, and the buffer addresses and lengths must be collected into the scatter/gather lists that describe the data to be transferred.
- After the transfer completes, the memory must be unwired so it can be paged out.

In the I/O Kit, all of this work is performed by objects of the IOMemoryDescriptor and IOMemoryCursor classes. An I/O request typically includes an IOMemoryDescriptor object that describes the areas of memory involved in the transfer. Initially, the description takes the form of an array of structures, each consisting of a client task identifier (`task_t`), an offset into the client's virtual address space, and a length in bytes. A driver uses the memory descriptor to prepare the memory pages—paging them into physical memory, if necessary, and wiring them down—by invoking the descriptor's `prepare` function.

When the memory is prepared, a driver at a lower level of the stack—typically a driver that controls a DMA (Direct Memory Access) engine—then uses a memory-cursor object to get the memory descriptor's buffer segments and with them generate a scatter/gather list suitable for use with the hardware. It does this by invoking the `getPhysicalSegments` function of the memory cursor and doing any necessary processing on the segments it receives. When the I/O transfer is complete, the driver originating the I/O request invokes the memory descriptor's `complete` function to unwire the memory and update the virtual-memory state. When all this is done, it informs the client of the completed request.

Beginning in Mac OS X v10.2, IOBufferMemoryDescriptor (a subclass of IOMemoryDescriptor) allows a buffer to be allocated in any task for I/O or sharing through mapping. In previous versions of Mac OS X, an IOBufferMemoryDescriptor object could only represent a buffer allocated in the kernel's address space. In Mac OS X v10.2 and later, however, the changes to the IOBufferMemoryDescriptor API support a better way to handle I/O generated at the behest of a nonkernel client. Apple recommends that such I/O be sent to buffers the kernel allocates in the client's address space using the IOBufferMemoryDescriptor API. This gives control of the allocation method to the kernel, ensuring that the buffer is allocated in accordance with internal guidelines dictated by the virtual-memory system. The user task can still specify the kind of buffer to be allocated, such as pageable or sharable. And, the user task can still access the buffer using the `vm_address_t` and `vm_size_t` variables it receives from the user client. For software running in Mac OS X v10.2 and later, Apple recommends that:

- User tasks no longer use `malloc` or other user-level library functions to allocate I/O buffers in their own address space.

- User clients use IOBufferMemoryDescriptor objects to represent kernel-allocated buffers instead of the IOMemoryDescriptor objects that represented user task–allocated buffers.

Network drivers are the exception to the use of IOMemoryDescriptor objects. The Network family instead uses the `mbuf` structure defined by the BSD kernel networking stacks. BSD `mbuf` structures are already optimized for handling network packets, and translating between them and memory descriptors would merely introduce unnecessary overhead. The network family defines subclasses of IOMemoryCursor for extracting scatter/gather lists directly from `mbuf` structures, essentially making this aspect of I/O handling the same for network drivers as for any other kind of driver.

IOMemoryDescriptor and IOMemoryCursor are capable of handling the management and reuse of memory buffers for most drivers. Drivers with special requirements, such as that all buffer memory be contiguous or located within a particular area in physical memory, must perform the extra work necessary to meet these requirements, but still use memory descriptors and cursors for their interaction with the I/O Kit.

Memory in an I/O Request

Although the preceding section discusses how I/O Kit drivers transfer data between device and system memory by using objects of the IOMemoryDescriptor and IOMemoryCursor classes, it does so at a fairly general level. It's also instructive to consider an I/O request at a more detailed level: What happens from the moment a user process makes a call to write or read data to the instant the data is transferred to or from a device? What are the exact roles played by IOMemoryDescriptors, IOMemoryCursors, and other I/O Kit objects in this chain of events?

All read and write operations in user space—that is, made by applications or other non-kernel processes—are based ultimately on I/O vectors. An I/O vector is an array of structures, each of which gives the address and length of a contiguous chunk of memory of a particular process; this memory is expressed in the virtual address space of the process. An I/O vector is sometimes known as a scatter/gather list.

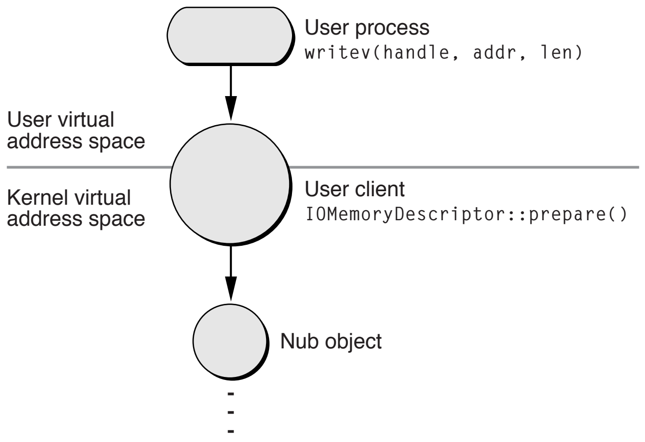
To start with a more specific example, a process makes a call to write some data to a device. It must pass in a minimum set of parameters: a handle to the target of the call (`io_service_t`), a command ("write"), the base address of the I/O vector array, and the number of array elements. These parameters get passed down to the kernel. But how is the kernel to make sense of them, particularly the base address of the array, which is typed as `void` *. The kernel lives in its own virtual address space, and the virtual address space of a user process, untranslated, means nothing to it.

Before going further, let's review the memory maps maintained by the operating system. In Mac OS X there are three different kinds of address space:

- The virtual address space of individual user processes (such as applications and daemons)
- The virtual address space of the kernel (which includes the I/O Kit)
- Physical address space (system memory)

When a user process issues an I/O call, the call (with its parameters) percolates down to the kernel. There the I/O Kit converts the handle parameter to an object derived from an appropriate IOUserClient subclass. The user-client object logically sits astride the boundary separating the kernel and user space. There is one user client per user process per device. From the perspective of the kernel, the user client is a client driver at the top of the stack that communicates with the nub object below it (see Figure 8-1 (page 90)).

Figure 8-1 The role of the user client in an I/O command

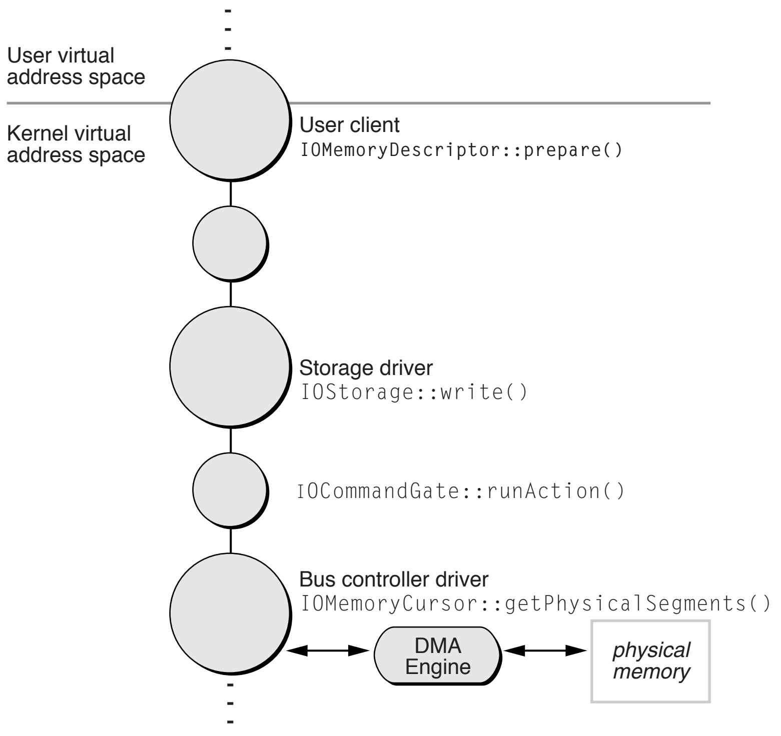

User process
writev(handle, addr, len)
User virtual address space
Kernel virtual address space
User client
IOMemoryDescriptor::prepare()
Nub object

The user client also translates between the address spaces, mapping the user-process buffers into the kernel's virtual address space. To do this, it creates an IOMemoryDescriptor object from the other parameters specified in the original call. The particular power of an IOMemoryDescriptor object is that it can describe a piece of memory throughout its use in an I/O operation and in each of the system's address spaces. It can be passed between various drivers up and down the driver stack, giving each driver an opportunity to refine or manipulate the command. A driver can also reuse a memory descriptor; in some circumstances, it is better—if continuous allocation is going to involve a performance hit—to create a pool of IOMemoryDescriptors ahead of time and recycle them.

After the user client creates (or reuses) the IOMemoryDescriptor object, it immediately invokes the memory descriptor's `prepare` member function. The `prepare` call first makes sure physical memory is available for the I/O transfer. Because this is a write operation, the virtual memory system may have to page in the data from its store. If it were a read operation, the virtual-memory (VM) pager might have to swap out some other pages from physical memory to make room for the I/O transfer. In either case, when the VM pager is involved in preparing physical memory, there are implications for bus controller drivers that program DMA engines or otherwise must deal directly with the requirements of devices (see "DMA and System Memory" (page 94)). After sufficient physical memory is secured for the I/O transfer, the `prepare` function wires the memory down so it cannot be paged out.

The `prepare` call must be made before the I/O command crosses formally into the I/O Kit, and it should also be made on the requesting client's thread and before taking any locks. Never invoke `prepare` within the command-gate context because `prepare` is a synchronous call and can *block* (that is, wait for some other process to complete) indefinitely. When `prepare` returns, the user client (typically) initiates the call to schedule the I/O request, passing the memory descriptor to a driver down the stack (through established interfaces, as described in "Relaying I/O Requests" (page 93)). The memory descriptor might be passed along to other drivers, each of which might manipulate its contents, until it eventually reaches an object that is close to the hardware itself—typically a bus controller driver or that driver's client nub. This object schedules the I/O request and takes the command (see "I/O Requests and Command Gates" (page 83) for information on command gates). Within the command gate it usually queues up the request for processing as soon as the hardware is free.

Figure 8-2 The principal I/O Kit objects in an I/O transfer


User virtual address space
Kernel virtual address space
User client
IOMemoryDescriptor::prepare()
Storage driver
IOStorage::write()
IOCommandGate::runAction()
Bus controller driver
IOMemoryCursor::getPhysicalSegments()
DMA Engine
physical memory

Ultimately receiving the request at the lower levels of the driver stack is a bus controller driver (such as for ATA or SCSI). This lower-level driver object must program the controller's DMA engine or otherwise create a scatter/gather list to move data directly into and out of the device. Using an IOMemoryCursor object, this driver generates physical addresses and lengths from the memory descriptor's buffer segments and does any other work necessary to create a scatter/gather list with the alignment, endian format, and size factors required by the hardware. To generate the segments, it must call the `getPhysicalSegments` function of the memory cursor. This whole procedure is run in the work-loop context. At the completion of the transfer, a hardware interrupt is typically generated to initiate an I/O transfer in the other direction.

When the I/O transfer is complete, the object that called `prepare` invokes the memory descriptor's `complete` function to unwire the memory and update the virtual-memory state. It's important to balance each `prepare` with a corresponding `complete`. When all this is done, the user client informs the originating process of the completed request.

Issues With 64-Bit System Architectures

Beginning with Mac OS X v10.3 and later, Apple introduced some changes to allow existing device drivers to work with the new 64-bit system architectures. The problem to be solved involved the communication between devices on the PCI bus, which can handle 32-bit addresses, and the 64-bit main memory.

Apple solved the problem with address translation which “maps” blocks of memory into the 32-bit address space of a PCI device. In this scheme, the PCI device still sees a 4-gigabyte space, but that space can be made up of noncontiguous blocks of memory. A part of the memory controller called the DART (device address resolution table) translates between the PCI address space and the much larger main memory address space. The DART handles this by keeping a table of translations to use when mapping between the physical addresses the processor sees and the addresses the PCI device sees (called I/O addresses).

The address-translation process is transparent if your driver adheres to documented, Apple-provided APIs. For example, when your driver calls the IOMemoryDescriptor method `prepare`, a mapping is automatically placed in the DART. Conversely, when your driver calls IOMemoryDescriptor’s `release` method, the mapping is removed. Although this has always been the recommended procedure, failure to do this in a driver running on Mac OS X v10.3 or later may result in random data corruption or panics. Be aware that the `release` method does not take the place of the `complete` method. As always, every invocation of `prepare` should be balanced with an invocation of `complete`.

If your driver experiences difficulty on a Mac OS X v10.3 system, you should first ensure that you are following these guidelines:

- Always call `IOMemoryDescriptor::prepare` to prepare the physical memory for the I/O transfer (this also places a mapping into the DART).
- Balance each `IOMemoryDescriptor::prepare` with an `IOMemoryDescriptor::complete` to unwire the memory.
- Always call `IOMemoryDescriptor::release` to remove the mapping from the DART.
- On hardware that includes a DART, pay attention to the DMA direction for reads and writes. On a 64-bit system, a driver that attempts to write to a memory region whose DMA direction is set up for reading will cause a kernel panic.

One side effect of these changes in the memory subsystem is that Mac OS X is likely to return physically contiguous page ranges in memory regions. In earlier versions of Mac OS X, the system returned multi-page memory regions in reverse order, beginning with the last page and moving towards the first page. Because of this, a multi-page memory region seldom contained a physically contiguous range of pages.

The greatly increased likelihood of seeing physically contiguous blocks of memory in memory regions might expose latent bugs in drivers that did not previously have to handle physically contiguous pages. Be sure to check for this possibility if your driver is behaving incorrectly or panicking.

Another result of the memory-subsystem changes concerns physical addresses a driver might obtain directly from the pmap layer. Because there is not a one-to-one correspondence between physical addresses and I/O addresses, physical addresses obtained from the pmap layer have no purpose outside the virtual memory system itself. Drivers that use pmap calls (such as `pmap_extract`) to get such addresses will fail to work on systems with a DART. To prevent the use of these calls, Mac OS X v10.3 will refuse to load a kernel extension that uses them, even on systems without a DART.

Relaying I/O Requests

Client requests are delivered to drivers through specific functions defined by the driver's family. A storage family driver, for example, handles read requests by implementing the function `read`. Similarly, a network family driver handles requests for transmitting network packets by implementing the function `outputPacket`.

An I/O request always includes a buffer containing data to write or providing space for data to be read. In the I/O Kit this buffer takes the form of an IOMemoryDescriptor object for all families except networking, which uses the `mbuf` structure defined by BSD for network packets. These two buffer mechanisms provide all drivers with optimized management of data buffers up and down the driver stacks, minimizing the copying of data and performing all the steps required to prepare and complete buffers for I/O operations.

An I/O Kit family defines I/O and other request interfaces. You typically don't have to worry about protection of resources in a reentrant context for your driver, unless it specifically forgoes the protection offered by the family or the I/O Kit generally.

More on Memory Descriptors

A memory descriptor is an object inheriting from the IOMemoryDescriptor class that describes how a stream of data, depending on direction, should either be laid into memory or extracted from memory. It represents a segment of memory holding the data involved in an I/O transfer and is specified as one or more physical or virtual address ranges (a range being a starting address and a length in bytes).

An IOMemoryDescriptor object permits objects at various levels of a driver stack to refer to the same piece of data as mapped into physical memory, the kernel's virtual address space, or the virtual address space of a user process. The memory descriptor provides functions that can translate between the various address spaces. In a sense, it encapsulates the various mappings throughout the life of some piece of data involved in an I/O transfer.

IOMemoryDescriptor is an abstract base class defining common methods for describing physical or virtual memory. Although it is an abstract class, the I/O Kit provides a concrete general-purpose implementation of IOMemoryDescriptor for objects that are directly instantiated from the class. The I/O Kit also provides two specialized public subclasses of IOMemoryDescriptor: IOMultiMemoryDescriptor and IOSubMemoryDescriptor. Table 8-1 (page 93) describes these classes.

Table 8-1 Subclasses of IOMemoryDescriptor

Class	Description
IOMultiMemoryDescriptor	Wraps multiple general-purpose memory descriptors into a single memory descriptor. This is commonly done to conform to a bus protocol.
IOSubMemoryDescriptor	Represents a memory area which comes from a specific subrange of some other IOMemoryDescriptor.

The IOMemoryDescriptor base class itself defines several methods that can be usefully invoked from objects of all subclasses. Some methods return the descriptor's physically contiguous memory segments (for use with an IOMemoryCursor object) and other methods map the memory into any address space with caching and placed-mapping options.

A related and generally useful class is IOMemoryMap. When you invoke IOMemoryDescriptor's `map` method to map a memory descriptor in a particular address space, an IOMemoryMap object is returned. Objects of the IOMemoryMap class represent a mapped range of memory as described by an IOMemoryDescriptor. The mapping may be in the kernel or a non-kernel task, or it may be in physical memory. The mapping can have various attributes, including processor cache mode.

More on Memory Cursors

An IOMemoryCursor lays out the buffer ranges in an IOMemoryDescriptor object in physical memory. By properly initializing a memory cursor and then invoking that object's `getPhysicalSegments` function on an IOMemoryDescriptor, a driver can build a scatter/gather list suitable for a particular device or DMA engine. The generation of the scatter/gather list can be made to satisfy the requirements of segment length, transfer length, endian format, and alignment imposed by the hardware.

A controller driver for a bus such as USB, ATA, or FireWire is typically the object that uses IOMemoryCursors. Such drivers should create an IOMemoryCursor and configure the memory cursor to the limitations of the driver's DMA hardware or (if PIO is being used instead) the limitations of the device itself. For instance, the memory cursor used for the FireWire SBP-2 protocol should be configured to a maximum physical segment size of 65535 and an unlimited transfer size.

You can configure an IOMemoryCursor in a variety of ways. The most obvious way is to supply the initialization parameters: maximium segment size, maximum transfer length, and a pointer to a segment function. This callback function, typed `SegmentFunction`, writes out a single physical segment to an element in a vector array defining the scatter/gather list. Your driver can also perform post-processing on the extracted segments, swapping bytes or otherwise manipulating the contents of the segments. Finally, you can create a subclass of IOMemoryCursor or use one of the subclasses provided by Apple. See "IOMemoryCursor Subclasses" (page 96) for more on this topic.

DMA and System Memory

Writers of bus controller drivers have two basic considerations when they effect I/O transfers. They are receiving data laid out in memory in a particular way and they must send that data to a destination that might expect a radically different memory layout. Depending on direction, the source and destination can be either a DMA engine (or a specific device) or the client memory represented by an IOMemoryDescriptor. Additionally, memory coming from the system might be conditioned by the Unified Buffer Cache (UBC) and this has implications for driver writers. This section discusses some of these factors.

Direct Memory Access (DMA) is a built-in capability of certain bus controllers for directly transferring data between a device attached to the bus and system memory—that is, the physical memory on the computer's motherboard. DMA enhances system performance by freeing the microprocessor from having to do the transfer of data itself. The microprocessor can work on other tasks while the DMA engine takes care of moving data in and out of the system.

Each bus on a Mac OS X system has its own DMA engine and they all are different. The PCI bus controller on Mac OS X uses bus master DMA, a type of DMA wherein the controller controls all I/O operations on behalf of the microprocessor. Other bus controllers (ATA, SCSI, or USB, for example) implement different DMA engines. Each engine can have its own alignment, endian format, and size restrictions.

An alternative to DMA is the Programmed Input/Output (PIO) interface, usually found on older or under-designed hardware. In PIO, all data transmitted between devices and system memory goes through the microprocessor. PIO is slower than DMA because it consumes more bus cycles to accomplish the same transfer of data.

Mac OS X supports both bus master DMA and PIO for moving data in and out of the system. In fact, some drivers could conceivably make use of both models for the same I/O transfer—for example, processing most bytes using DMA and then processing the last few bytes using PIO, or using PIO to handle error conditions.

The Unified Buffer Cache (UBC) is a kernel optimization that combines the file-system cache and the virtual-memory (VM) cache. The UBC eliminates the situation where the same page is duplicated in both caches. Instead, there is just one image in memory and pointers to it from both the file system and the VM system. The underlying structure of the UBC is the Universal Page List (UPL). The VM pager deals with memory as defined by UPLs; when an I/O request is based on paged-in memory, it is called a conforming request. A non-conforming request is one that isn't UPL-defined. A UPL segment of memory has certain characteristics; it:

- Is page sized (4 kilobytes)
- Is page aligned
- Has a maximum segment size of 128 kilobytes
- Is already mapped into the kernel's virtual address space

The I/O Kit has adapted its APIs to the UPL model because it's more efficient for I/O transfers; in addition, it makes it easier to batch I/O requests. An IOMemoryDescriptor object might be backed—entirely or partially—by UPL-defined memory. If the object is backed by a UPL, then there cannot be more than a prearranged number of physical segments. The bus controller driver that extracts the segments (using an IOMemoryCursor) must allocate sufficient resources to issue the I/O request associated with the memory descriptor.

Dealing With Hardware Constraints

Apple's policy on how drivers should deal with hardware constraints is generous toward clients of DMA controller drivers and places certain expectations on the drivers themselves.

- Clients of DMA controller drivers have no alignment restrictions placed on them. Generally, UPL-defined data (page-aligned and page-sized) should be optimal, but it is not required.
- If the memory is UPL-defined, then in order to avoid deadlocking the driver should not allocate memory. Controller drivers must be able to process a UPL without allocating any buffers, or they must preallocate sufficient resources prior to any particular UPL-based I/O transfer. Operations that don't meet the pager constraints (that is, that aren't UPL-based) can allocate buffers.

- A controller driver must do its best to honor any request that it receives. If necessary, it should even create a buffer (or even use a static buffer), or preallocate resources and copy the data. (Remember that allocation during an I/O request can cause a deadlock.) If the driver cannot execute the I/O request, it should return an appropriate error result code.

In summary, driver writers must be prepared to handle a request for any alignment, size, or other restriction. They should first attempt to process the request as conforming to UPL specifications; if the assumption of UPL proves true, they should never allocate memory because doing so could lead to deadlock. If the request is non-conforming, the driver should (and can) do whatever it has to do to satisfy the request, including allocating resources.

IOMemoryCursor Subclasses

Apple provides several subclasses of IOMemoryCursor for different situations. If your DMA engine requires a certain endian data format for its physical segments, your driver can use the subclasses that deal with big-endian and little-endian data formats (and thus will not have to perform this translation when it builds the scatter/gather lists for the DMA engine). Another subclass enables your driver to lay out data in the byte orientation expected by the system's processor. Table 8-2 (page 96) describes these subclasses.

Table 8-2 Apple-provided subclasses of IOMemoryCursor

Subclass	Description
IONaturalMemoryCursor	Extracts and lays out a scatter/gather list of physical segments in the natural byte order for the given CPU.
IOBigMemoryCursor	Extracts and lays out a scatter/gather list of physical segments encoded in big-endian byte format. Use this memory cursor when the DMA engine requires a big-endian address and length for each segment.
IOLittleMemoryCursor	Extracts and lays out a scatter/gather list of physical segments encoded in little-endian byte format. Use this memory cursor when the DMA engine requires a little-endian address and length for each segment.

Of course, you can create your own subclass of the virtual IOMemoryCursor or of one of its subclasses to have your memory cursor accomplish exactly what you need it to do. But in many cases, you may not have to create a subclass to get the behavior you're looking for. Using one of the provided memory-cursor classes, you can implement your own `outputSegment` callback function (which must conform to the `SegmentFunction` prototype). This function is called by the memory cursor to write out a physical segment in the scatter/gather list being prepared for the DMA engine. In your implementation of this function, you can satisfy any special layouts required by the hardware, such as alignment boundaries.

Managing Power and Device Removal

Over the life of a device driver, from the time it is loaded to the time it is unloaded, the range of events that the driver is expected to handle is *mostly* well defined. Whether those events are I/O requests, hardware interrupts, or some other category of event, they all deal ultimately with moving data between device and system. However, a driver can receive events that might not have any direct bearing on its principal responsibility. To ensure the orderly behavior of the operating system, it must be prepared to deal with these events.

Two major types of events that fall into this "unscheduled" category are power events and events generated from devices being "hot-swapped"—specifically devices removed from a running Mac OS X system. This chapter discusses how device drivers should anticipate and respond to such events.

Handling Unscheduled Asynchronous Events

Power events and hot-swapping—particularly events precipitated when a user removes a device from a running Mac OS X system—share some characteristics. Primarily, they are asynchronous events that don't fall in the normal range of events a driver is expected to handle. These asynchronous activities globally affect hardware and generally happen as the result of something outside the driver's control.

A driver can be performing any operation when one of these "unscheduled" asynchronous events hits it. When this happens, there are two important things for the driver to do: manage the state associated with its device and do the proper locking to prevent data corruption. The I/O Kit introduces functions and classes to help you manage these tasks, especially managing driver state.

As a general approach, a driver should be careful about its execution context when it handles the event. A power-management event and a device-removal event both should be handled in a work-loop context. To handle these events, the driver needs to close the command gate and either handle the event immediately or queue it up for handling right after the gate is opened. Whereas the work loop for the power management should run in a separate thread, the handler for the device-removal event should execute in the work-loop context of the driver's provider.

In other words, you handle a power-management event and a device-removal event just as you would any asynchronous I/O request. You should take a lock such as a command gate and handle the event within that context. But remember, as with I/O requests, it's bad programming practice to make calls within that context that might block the caller indefinitely. Never block waiting for hardware to finish. Allocate resources before taking the command gate, if possible, and only if it is necessary to manage the event.

Power Management

The power-management functionality of the I/O Kit aims to minimize the power consumed by a computer system, behavior that is especially important for portable computers where battery life is a crucial feature. Power management also imposes an orderly sequence of actions, such as saving and restoring state, when a system (or a parts of it) are powered on and off.

Concepts of Power Management

Several concepts underpin power management in the I/O Kit. They involve such issues as the roles of and relationships between objects and the factors and logic that go into decisions on changing power states. An understanding of these concepts is essential to implementing power management in your driver objects.

Devices and Power States

The fundamental entity in power management is the device. From the perspective of this technology, a device is a unit of hardware whose power consumption can be measured and controlled independently of system power. (A device that the system cannot directly control can be indirectly controlled through the power domain to which the device belongs; see "Power Domains" (page 98) for further information.) A device can also have some state that needs to be saved and restored across changes in power.

A device has at least two power states associated with it—off and on. A device that is on uses maximum power and has complete functionality; when it is off, it uses no power and has no capability. Some devices can also have intermediate power states. A device can be in a reduced-power state in which it is still usable (albeit at a lower level of performance or functionality). A device can also be in an intermediate state in which it is not usable but still retains some configuration or state.

The power-management functionality of the I/O Kit associates several attributes with each power state of a device:

- How much power the devices uses while in that state
- The capability of the device while in that state
- How much power is required to change the device to the next highest state (currently unused)
- How much time is required to change the device to that state (currently unused)

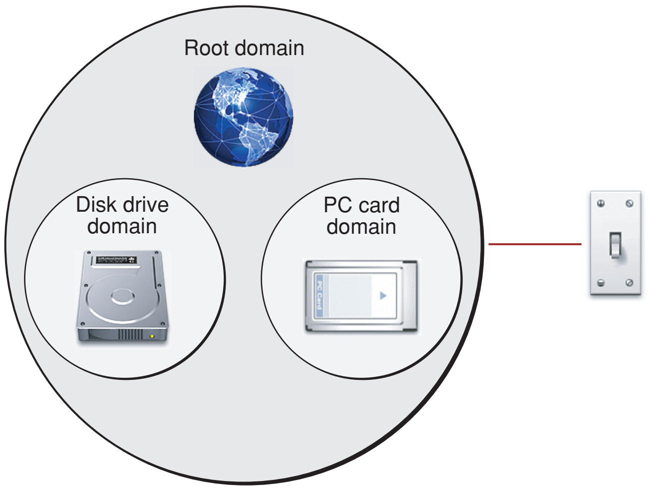
Power Domains

A power domain is a switchable source of power in the system. It provides power for one or more devices that are considered members of the power domain. When a power domain is off, all members of that domain are turned off. As with devices, a power domain has at least two power states—off and on—and might also have intermediate power states.

Power domains are hierarchical. Any power domain can "contain" another power domain. The all-enclosing domain is called the root power domain, which represents the main power of the system. Every Mac OS X system has at least the root power domain. The root power domain also corresponds to the Power plane (IOPower) of the I/O Registry. (You can enter `ioreg -p IOPower` at the command line to see the current power plane.) See Figure 9-1 (page 99) for a depiction of power domains in a system.

The PC card system is an example of a power domain that is contained by another power domain. It provides power to its member devices (the PC card slots) and is, in turn, a member of the root power domain.

Figure 9-1 Power domains and devices



Logic and Factors of Power Management

A major factor in power management is device idleness. Power management—considered as the sum of all I/O Kit objects involved in this activity—works by monitoring devices and deciding whether they are idle. If power management deems a device to be idle, it proceeds based on the type of device:

- If the device's power can be directly controlled, power management lowers it to the next lower power state. If, after this action, the device is in its lowest power state, power management checks the other devices in the same power domain to see if they are also in their lowest power state. If they are, it lowers the power state of the domain.
- If the device's power cannot be directly controlled, power management reduces the power consumed by the device by reducing power to the domain the device is a member of. If this device and all other devices in this domain are in their lowest power state after this, power management lowers the power state of the domain.

In both instances, after lowering the power state of the current domain, power management determines if the power state of the parent domain can be lowered.

When a device in a reduced-power state needs to be functional again, power management raises the power to the device to a usable level (generally, the "on" state). This step might require power management to power up the power domain the device is a member of as well as any enclosing domains.

After all power domains except the root domain have been powered down, power management puts the system to sleep. When a user wakes the system, the system resumes running with all power domains turned on.

The period of idleness after which power management changes a device's power state is determined by an object called a policy maker (see "Policy Makers and Power Controllers" (page 100)). The object can be influenced by several factors when it has to decide when this period has expired, including power-supply conditions (such as a low battery) and something called aggressiveness.

Aggressiveness is an indication of how aggressive power management should be in conserving power. For example, a person using a portable computer on a long flight would probably prefer aggressive power management and would be willing to suffer longer latency in exchange for longer battery life. On the other hand, the same person using the same computer in a meeting would probably prefer performance over aggressive power management. Aggressiveness in such cases obviously needs to be a configurable option.

There are also several levels of system-wide aggressiveness. One level is set when AC power is present in the system, and another is set when the computer is running on battery power; another level comes into play when the battery is low on power. Users can change the first two system-wide aggressiveness factors but not the third. When power-supply conditions trigger a system-wide change in aggressiveness, that new factor is propagated to all power-management objects.

Policy Makers and Power Controllers

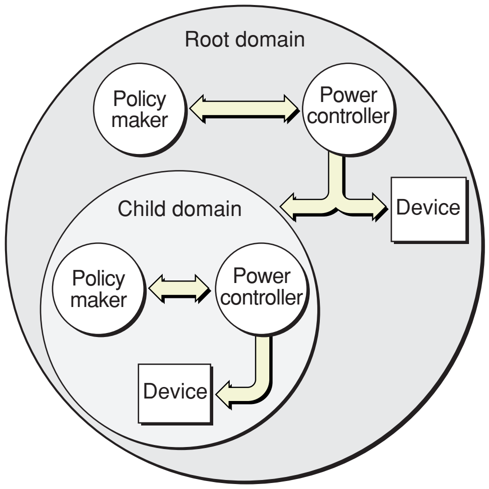
Each entity in power management—device, power domain, and thermal zone—is represented by an object that inherits from the IOService class. When linked together, these objects form the power-domain hierarchy for a system, which is also equivalent to the IOPower plane in the I/O Registry. IOService is the common base class for power-management objects because it provides methods enabling power-management objects to communicate with each other.

The IOService class also defines methods meant to be overridden by two types of power-management objects: policy makers and power controllers. A power controller knows about the various power states of a device and can switch the device between them. A policy maker, on the other hand, decides *when* to change the power state of a device and instructs the power controller for the device to carry out this change.

A policy maker is typically an instance of an I/O Kit family class but can be any object at an appropriate level in a driver stack. Using aggressiveness as well as other factors, the policy maker decides when power to the device should be lowered. The policy maker communicates its decision to a power-controller object, which carries it out. The power-controller object not only sets the power state of a device at the direction of the policy maker, but communicates power-related information about the device to the policy maker to assist it in its decisions. The policy maker also decides to raise power in a device when some object in the system informs it that the device is needed; it communicates this decision to the power controller.

Figure 9-2 (page 101) illustrates the roles of power controllers and policy makers.

Figure 9-2 Power controllers and policy makers



Power domains also have policy-maker and power-controller objects associated with them. The policy maker decides when to change the power state of the domain and has the power controller carry out its decision. However, there are a couple of differences between policy makers for devices and policy makers for power domains. The first is that a power-domain policy maker does not alter the power consumption of a device; instead, it alters the character of the power supplied to the members of the domain. The character of power, in this regard, can involve factors other than voltage, such as adjusting the frequency of the clock, or turning it off or on. The second difference is that a policy maker for a power domain doesn't base its decisions on idleness. It decides to raise and lower power in the domain based on requests from its members. For example, when the power domain is in a state of lowered power and one of its members requests more power, the policy maker tells the power controller to raise power throughout the domain. When all members of the domain have requested low power, the policy maker has the power of the domain lowered.

Both devices and power domains are affected by the power domains they are members of. Changes in power induced by a power domain can bypass policy makers. By altering the electrical power to a device, a power domain can alter its power state. Devices that cannot be directly controlled are especially susceptible to changes in their power domain. When the power domain is brought up, the device is turned on, and when the power domain is brought down, the device is effectively turned off.

For guidelines on creating policy makers and power controllers, see "Implementing Policy-Maker Objects" (page 104) and "Implementing Power-Controller Objects" (page 106).

Notification of Power-State Changes

Power management brackets a change to the power of a device or a power domain with a pair of notifications to all interested objects to allow them to prepare for the change. Interested objects are IOService driver objects that want to be informed of power-state changes for a variety of reasons. A policy maker's power controller is automatically included in the list of interested objects.

The notifications are delivered through invocations of `powerStateWillChangeTo` and `powerStateDidChangeTo` methods, which interested objects must implement in order to receive the notifications. The notified objects here have an opportunity to prepare for the change.

Usually, interested objects want the notification so that they can receive or restore state and program devices upon system power-up. But they can also just want to be made aware when a particular device is usable. The bracketed notifications are also a way to ensure that objects refrain from accessing parts of a device that are sensitive to power changes during a transition.

Notifications of power-state changes work almost identically for devices and power domains. A big difference, however, is the set of interested driver objects. For a power domain, the principal interested driver object is the driver that switches the power domain between states. Also notified are the children of the domain, which are the driver objects controlling power to devices or to other power domains. Notifications to child power domains are propagated, in turn, to the members of those domains. Each parent domain must get acknowledgement from all of its children before a power-state change is either begun or completed. Another difference in power-state notification for power domains is the value returned from the notification methods. If the value is positive, the policy maker does not interpret it as time, only as an indication that the interested object will call `acknowlegePowerChange` in the near future.

User processes can also request notifications of system and device power events. See "Notifying User Processes" (page 111) for details. Also, see "Notifying I/O Kit Objects" (page 110) for the procedure for notifying I/O Kit objects of power-state changes.

Overview of Power Management APIs

The I/O Kit concentrates most of the programmatic interfaces for power management in the IOService class, which is defined in `IOService.h`. A subset of IOService methods enables power management, including most methods called or overridden by policy makers and power controllers. For discussions of these methods, see "Implementing Policy-Maker Objects" (page 104) and "Implementing Power-Controller Objects" (page 106).

Not all power-management APIs are defined in `IOService.h`. The IOService class itself includes two power-management objects as instance variables. One object encapsulates the private power-management data structures of the IOService object, and the other encapsulates the protected data structures. The classes for both objects—IOPMpriv and IOPMprot—are defined in `IOServicePM.h`. If an IOService object is not a policy maker (that is, it does not call `PMinit`) its power-management instance variables are not initialized with the appropriate objects.

The IOPMpriv class defines instance variables for various timers, locks, lists, and other entities required for power management. It also includes ordinal values representing the power state currently desired by the policy maker (for device or domain), the controller driver, and any controllable device. The IOPMprot class defines instance variables specifying the array of known power states (for a device or domain), the current power state, flags characterizing the union of all power states in the array, aggressiveness values, and pointers to the policy maker's parent domain and power controller, as well as its work loop, command queue, and command gate. See the reference documentation or `IOServicePM.h` for descriptions of these and other instance variables of IOPMpriv and IOPMprot.

The I/O Kit defines another set of power-management APIs in the header files in `Kernel.framework/Headers/IOKit/pwr_mgmt`. Most of the APIs here are for communication with the root power domain, and third-party implementers of policy makers and power controllers generally do not need to use them. But some header files in `pwr_mgmt` do define important classes and data structures used throughout power management.

The IOPowerConnection class (`IOPowerConnection.h`) adds some instance variables and accessor methods to the IOService class that help to coordinate parent and child power domains. Several IOService methods that are related to power domains take IOPowerConnection objects in their parameters.

The `IOPM.h` header file in `pwr_mgmt` defines various constants and structures that are used throughout the power-management APIs. One of the more ubiquitous types is `IOPMPowerFlags`, which is used to characterize the power supplied by a device or power domain in a particular power state. Table 9-1 (page 103) describes the `IOPMPowerFlags` flags that can be set.

Table 9-1 IOPMPowerFlags settings

Flag	Description
`kIOPMNotAttainable`	The device or domain cannot put itself into this state.
`kIOPMPowerOn`	The power domain has applied power to its members.
`kIOPMClockNormal`	The power domain's clock is running at its normal speed.
`kIOPMClockRunning`	The power domain's clock is running.
`kIOPMWakeupEnabled`	The power domain is capable of passing on a device's wake-up signal in this state.
`kIOPMAuxPowerOn`	The PCI auxiliary power supply is on.
`kIOPMDeviceUsable`	The clients of the device can use it in this state.
`kIOPMMaxPeformance`	The device is capable of its highest performance in this state.
`kIOPMContextRetained`	The device can retain any context in this state.
`kIOPMConfigRetained`	The device can retain any configuration in this state.
`kIOPMNotPowerManaged`	The device is not being managed by a power controller.
`kIODoze`	The system is dozing.

A policy maker does not directly know anything about the power states of devices or power domains. The power controller for these devices or domains must inform the policy maker of valid power states by invoking `registerPowerDriver`, passing in an array of `IOPMPowerState` structures. Thus the `IOPMPowerState` structure is a central power-management data type. Table 9-2 (page 103) describes its fields.

Table 9-2 IOPMPowerState fields

Field	Description
`version`	Version number of this structure.
`capabilityFlags`	Describes the capability of the device in this state (as `IOPMPowerFlags`); used by interested objects.

Field	Description
`outputPowerCharacter`	Describes the power supplied in this state (as `IOPMPowerFlags`); used by power-domain children.
`inputPowerRequirement`	Describes the input power required in this state (as `IOPMPowerFlags`); used by the power-domain parent.
`staticPower`	Average power consumption (in milliwatts) of a device in this state or of the controller of a power domain in this state.
`unbudgetedPower`	Additional power consumption (in milliwatts) from a separate power supply, such as a battery.
`powerToAttain`	The power consumed by a device or power domain (in milliwatts) in entering this state from the next lowest state.
`timeToAttain`	The time (in microseconds) required for a device or power domain to enter this state from the next lower state; in other words, the time required to program the hardware.
`settleUpTime`	The time (in microseconds) required to allow power to settle after entering this state from the next lower state.
`timeToLower`	The time (in microseconds) required for a device or power domain to enter this the next lower state from this state; in other words, the time required to program the hardware.
`settleDownTime`	The time (in microseconds) required to allow power to settle after entering the next lower state from this state.
`powerDomainBudget`	The power (in milliwatts) that a power domain in this state is electronically able to deliver to its children.

To summarize, `IOPMPowerFlags` is an important data type in several fields of the `IOPMPowerState` structure, which itself is part of the array of power states that is a central, protected instance variable (IOPMprot) of a policy-maker object.

Implementing Policy-Maker Objects

Recall the role of the policy maker in power management. The policy maker decides when the power state of a device or power domain should change to the next level. It then instructs the power controller for the device or domain to effect the change. In simple outline, this is what the policy maker does.

So what object in a driver stack should be the policy maker for a device or power domain? It can be any object, although many I/O Kit families implement policy-maker code in a superclass, typically the superclass for a nub. It should be an object that "knows" about the type of device being controlled, including its usage patterns. It can be the same object that acts as the power controller.

If you've decided to implement a policy maker, you must ensure that code for three major tasks is in place:

- Initialize the power-management system.

- Implement an idleness-determination policy.
- Communicate with the power controller.

In implementing a policy maker, you might need to override IOService power-management methods. In most cases, you should ensure that you invoke the superclass implementation in your overrides.

Keep in mind that the policy maker, once it has initialized power management, can access the protected instance variables defined in IOPMprot. These include the power-state array provided by the power controller, pointers to the power-domain parent and the power controller, and variables identifying the current power state, aggressiveness factor, and other values.

Initializing Power Management Instance Variables

In your driver object's `start` method, invoke the superclass's implementation of the `PMinit` method. This call initializes the power-management instance variables encapsulated by instances of IOPMpriv and IOPMprot. You need to initialize these variables further with the array of power states passed in by your power controller when that object registers with your policy maker through the `registerPowerDriver` method.

When a policy maker is properly initialized, it is properly integrated into the power-management hierarchy (with a parent power domain and any child domains) and will have its power-controller object set. A parent domain and a power controller must be established before a policy maker can figure out what power state a device is in. With the power flags describing the current state of its power domain, it can find out what state its device is in.

After calling `PMinit`, the policy maker should call also `joinPMTree` on its provider. This method causes the policy-maker object to be attached into the power management hierarchy. The default implementation of the method function calls the platform expert, which knows how to do the attaching.

Be sure to bracket the invocation of `PMinit` with an invocation of `PMstop` in your `stop` method, or whenever you need to relinquish your object's role as policy maker. The `PMstop` method releases the objects and instance variables initialized in `PMinit`.

Implementing Idleness Determination

The power-management feature of the I/O Kit provides a few APIs to help policy makers determine device idleness. Foremost among these is the `activityTickle` method, which helps the policy maker to know when the device is being used and, more importantly, when it isn't being used and so can be powered down. Objects in the driver stack that know a device is being used call `activityTickle` with a parameter of `kIOPMSubclassPolicy` to inform the policy maker. (Of course, if the policy maker is in the data path of the device, it knows automatically when the device is being used.)

If the policy maker is determining idleness totally on its own, the parameter that driver objects use with `activityTickle` should be `kIOPMSubclassPolicy` . The policy maker should implement `activityTickle` to intercept the call and should implement its own mechanism to gauge the interval that has elapsed without device activity—that is, idleness.

A policy maker can also determine idleness in cooperation with the IOService superclass. In this approach, the policy maker first calls `setIdleTimerPeriod` on its superclass, passing in the number of seconds for the timer interval. If the policy maker is relying on other objects to inform it of device usage, it should implement `activityTickle` as in the approach described above. When it becomes aware of device activity, either from another object or on its own, the policy maker calls activityTickle on its superclass, passing in a parameter of kIOPMSuperclassPolicy1. When the idle timer expires, the IOService superclass checks to see if there has been activity since the last timer expiration—that is, if activityTickle(kIOPMSuperclassPolicy1) has been called. If there has been device activity, it restarts the timer. If the timer expires without activity having occurred, IOService calls setPowerState on the power-controller object to power down the device to the next lowest state.

After the device has been powered down to a lower state, the next activityTickle invocation causes power management to raise the power in the device to the level required for the activity. If IOService is being used for this purpose (kIOPMSuperclassPolicy1), this happens automatically; IOService returns true if the device is currently in the correct state; otherwise it returns false and automatically makes the device usable. If the policy maker is implementing idleness determination on its own, it should call changePowerStateToPriv on its superclass. (The superclass implementation takes care of notifying interested objects as well as calling the power controller.)

A policy maker typically determines when a device is idle only after factoring in aggressiveness. As described earlier in "Logic and Factors of Power Management" (page 99), aggressiveness can be specific to a device or power domain, or it can be system-wide. Often a policy maker uses the level of aggressiveness—unless the aggressiveness does not apply to its device or domain—to change the interval of the idleness timer. The aggressive factor is automatically broadcast to all policy makers in the power plane.

However, since new aggressiveness values can be propagated at any time down the power-domain hierarchy, the policy maker should also override the setAggressiveness method to intercept new values and reset the idleness timer with them. If the policy maker overrides setAggressiveness, it should be sure to invoke the superclass implementation.

Communicating With the Power Controller

Once they have determined device idleness, policy-maker objects must instruct their power controllers to lower the power state of a device or power domain. If they are explicitly setting the idle timer (setIdleTimerPeriod) and calling activityTickle with a parameter of kIOPMSuperclassPolicy1, the IOService superclass calls setPowerState on the power controller for them.

The IOService implementation of setPowerState broadcasts notifications to all interested observers before (powerStateWillChangeTo) and after (powerStateDidChangeTo) invoking the power-controller implementation of the method.

The power controller itself can become aware of the need to change the power state of its device or power domain. In these cases, the power controller calls changePowerStateTo on the policy maker when it wants the device switched to this state. This gives the policy maker opportunity to ensure that the power domain is correct and to do anything else that is necessary before invoking the superclass implementation of setPowerState.

Implementing Power-Controller Objects

It is generally easier to identify the object that is the power controller for a device than that device's policy maker. In most cases, the driver that controls I/O for the device is the object that volunteers to control the device's power state. It is, after all, the object with the hardware-switching code. However, a power controller can be implemented in an I/O Kit family subclass, some other driver class, or in the same class that implements the policy maker.

Locating the policy maker for your power controller might present more of a challenge. The policy maker for devices of a certain type is usually implemented in an I/O Kit family, but not every family has a policy maker. Because Darwin is open source, you can look through the source code of a family to see what class, if any, is calling `PMinit`. If there is no policy maker for a family, your driver class can implement your policy maker *and* your power controller.

In its `start` method, the power-controller object should construct the array of power states (each a structure of type `IOPMPowerState`) and call `registerPowerDriver` on its policy maker, passing in a pointer to the array and the number of power states. If multiple power controllers volunteer to control a device or domain by calling `registerPowerDriver`, the one with the richest array of power states is typically the one chosen.

Listing 9-1 (page 107) shows how a power controller might register with a policy maker. It also shows the power controller allocating two threads on which to run the power state-change code (in a command gate, which will be demonstrated later).

Listing 9-1 Building the power-state array and registering with the policy maker

```
enum {
    k82558OffState  = 0,
    k82558OnState   = 1,
    kNum82558States = 2
};

static IOPMPowerState ourPowerStates[kNum82558States] = {
    {1,0,0,0,0,0,0,0,0,0,0,0},
    {1,IOPMDeviceUsable,IOPMPowerOn,IOPMPowerOn,0,0,0,0,0,0,0,0}
    };
IOReturn ExampleIntel82558::registerWithPolicyMaker(IOService * policyMaker)
{
    currentPowerState = k82558OnState;
    ourPolicyMaker = policyMaker;

   /*****
    * Allocate threads for asynchronous power-up and power-down.
    * Allocation failure is not detected here, but before use
    * in the setPowerState() method.
    */
    powerDownThread = thread_call_allocate((thread_call_func_t)startPowerDown,
        (thread_call_param_t)this);
    powerUpThread = thread_call_allocate((thread_call_func_t)startPowerUp,
        (thread_call_param_t)this);

    return policyMaker->registerPowerDriver(this,
                                            ourPowerStates,
                                            kNum82558States);
}
```

Note that a single power controller can control the power of multiple devices or power domains, each of which has its own policy maker. Also, a policy maker automatically includes its power controller among the list of interested objects it broadcasts notifications to. See “Notifying I/O Kit Objects” (page 110) for information on IOService’s notification methods.

Changing the Power State

The power controller overrides the `setPowerState` method and changes the power state of its device or power domain as indicated by the ordinal value passed in. This value is an index to the power-state array for the device or domain. The power controller alters the power state of a device by writing to certain device registers. If the power controller can satisfy the request right away, it switches the power to the new state and returns `IOPMAckImplied`. If it needs more time to satisfy the request, it returns a number indicating how many microseconds it needs. In the latter case, the power controller should call `acknowledgeSetPowerState` on the policy maker when it has completed the switch to the new power state.

Listing 9-2 (page 108) illustrates how a power controller might change the power state of a device. In this example, taken from the Intel 82558 Ethernet Controller example project, the code that changes the power state is run in a separate thread and in a protected command-gate context.

Listing 9-2 Setting the power state on a separate thread in a command-gate context

```
IOReturn ExampleIntel82558::setPowerState(unsigned long powerStateOrdinal,
    IOService * whatDevice)
{
    IOReturn result = IOPMAckImplied;

    if (currentPowerState == powerStateOrdinal) {
        return result;
    }

    switch (powerStateOrdinal) {
        case k82558OffState:
            /*****
            * The driver is being told to turn off the device for some reason.
            * It saves whatever state and context it needs to and then turns
            * off the device.
            *
            * It takes some time to turn off the 82558, so the driver spawns
            * a thread to power down the device and returns immediately to
            * the policy-maker giving an upper bound on the time it will need.
            */
            if ( powerDownThread ) {
                thread_call_enter(powerDownThread);
                result = kFiveSeconds;
            }
            break;

        case k82558OnState:
           /*****
            * The driver is being told to turn on the device.  It does so
            * and then restores any state or context that it has previously
saved.
            *
            * It takes some time to turn on the 82558, so the driver spawns
            * a thread to power up the device and returns immediately to
            * the policy-maker giving an upper bound on the time it will need.
            */
            if ( powerUpThread ) {
                thread_call_enter(powerUpThread);
                result = kFiveSeconds;
            }
            break;
    }

    return result;
}

void startPowerDown (thread_call_param_t arg, thread_call_param_t)
{
    void * argument2;
    void * argument3;
    void * argument4;

    ((ExampleIntel82558 *)arg)->getCommandGate()->runAction(
(IOCommandGate::Action)
                                         &startPowerDownGated,
                                         (void *)arg,
                                         (void *)argument2,
                                         (void *)argument3,
                                         (void *)argument4);

    return;
}


void startPowerDownGated (ExampleIntel82558 * self, void * arg2,
    void * arg3, void * arg4)
{
    self->doPowerDown();

    return;
}


void ExampleIntel82558::doPowerDown(void)
{
   /*****
    * At this point the driver would save any state or context that it
    * will restore later when power comes on again.  Then, if it can
    * do so, it switches the device off.
    */
    currentPowerState = k82558OffState;

   /*****
    * Since the driver returned a non-acknowledgement when called at
    * setPowerState(), it sends an acknowledgement to the policy-maker here.
    */
    ourPolicyMaker->acknowledgeSetPowerState();

    return;
}
```

Important: The above code fragment excludes the code that powers up the Intel 82558 Ethernet controller because this code is almost identical to the code that powers the controller down. In addition, the code fragment excludes some of the comments in the source file it was taken from, `ExampleIntel82558.cpp`. Refer to this file in the Intel 82558 Ethernet controller driver example project for the full set of comments.

Informing the Policy Maker

A power controller should never change the power state of a device or power domain without authorization from a policy maker. There might be power-budget restrictions it doesn't know about, and it cannot inform interested objects of changes because it doesn't know what those objects are. If a power controller becomes aware of a need to change the power state of its device or domain, it should inform its policy maker.

For example, if any object in the driver stack, including a user client, wants to raise a dormant device to an active power state, it can call IOService's `makeUsable` method. If the power controller intercepts this message, it should not effect the change immediately. In cases like this, it should call `changeStateTo` on its policy maker.

The policy maker occasionally queries its power controller about the power states specific to the device in relation to its power domain. There are three such methods invoked in the power controller:

- `initialPowerStateForDomainState`. The policy maker calls this method when it is initializing to find out which power state the device is in, given the current state of the power domain.
- `maxCapabilityForDomainState`. The policy maker calls this method to find out the highest power state possible for a given domain state.
- `powerStateForDomainState`. The policy maker calls this method to find out what power state the device would be in for a given power-domain state.

The default implementation for each of these methods, provided by IOService, scans the power-state array of its device or domain looking for the highest state whose `inputPowerRequirement` field (in `IOPMPowerState`) exactly matches the passed-in parameter. If you want a different behavior, you can have the power controller override one or more of these methods.

Receiving Notification of Power Events

The I/O Kit offers programmatic interfaces enabling driver objects in the kernel and processes outside the kernel to receive notifications before and after a device or power domain (especially the root power domain) undergoes a change in power state. This section discusses how to use those APIs, both for driver objects and user processes.

Notifying I/O Kit Objects

As "Notification of Power-State Changes" (page 101) explains, I/O Kit power management brackets each change to the power state of a device or power domain with a pair of notifications. The notifications are delivered through invocations of `powerStateWillChangeTo` and `powerStateDidChangeTo` methods, which must be implemented by interested objects. If you have a driver object that you want to receive these notifications, complete the following steps:

1. Register the driver as an interested object by calling the IOService method `registerInterestedDriver` on the policy-maker object for the device or domain.

2. Implement the `powerStateWillChangeTo` method.

   When the policy maker is about to instruct a power-controller object to alter the power state of a device or power domain, the first notification is broadcast via the `powerStateWillChangeTo` method. If the object is already prepared, it should return zero; if it needs more time to prepare, it should return an upper limit on the time required (in microseconds). It can examine the `IOPMPowerFlags` bitfield (see Table 9-1 (page 103)) passed in to make its determination; this bitfield is derived from the `capabilityFlags` field of the power-state array documented in Table 9-2 (page 103).

   If a driver returns a positive number, it should invoke the `acknowledgePowerChange` method of the policy maker once it is prepared. If it does not do this, and the time requested for preparation elapses, the policy maker carries on as if the driver acknowledged the change. This behavior prevents power-state changes from stalling because of failing drivers.

3. Implement the `powerStateDidChangeTo` method.

   After the change to power occurs and the power has settled to its new level, power management broadcasts the second notification to all interested objects via the `powerStateDidChangeTo` method. If a device or domain is going to a reduced power state, interested objects generally don't need to do much with this notification. However, if the device or domain is going to a greater power state, interested objects would use this notification to prepare for the change (restoring state or programming a device, for example). The opposite, of course, is true for objects preparing for reduced power states.

   The interested object can examine the `IOPMPowerFlags` bitfield (see Table 9-1 (page 103)) passed in to make its determination; this bitfield is derived from the `capabilityFlags` field of the power-state array documented in Table 9-2 (page 103). As with `powerStateWillChangeTo`, the notified object should return zero if it has prepared for the change. If it needs time to prepare, it should return the maximum time required (in microseconds); when the object is finally ready for the change, it should call the `acknowledgePowerChange` method of its policy maker.

4. Deregister the interested object by calling the IOService method `deRegisterInterestedDriver`.

   Deregister the object when your object is no longer interested in notifications of power-state changes. This often occurs in the `stop` method.

Notifying User Processes

The I/O Kit framework for device interfaces contains several functions enabling user processes to receive and respond to notifications of power events. Using this API, an application or other user process can be notified when the system is about to go to sleep or power down or up. It can also get power-state notifications relating to specific devices. In most cases, it can allow the power change or cancel it.

To get a notification of a system or device power event, do the following:

1. Implement a callback function that handles the notification. This function should conform to the prototype `IOServiceInterestCallback` defined in `IOKitLib.h`.

In the function, intercept the power-event messages you're looking for and act appropriately; these messages are defined in IOMessage.h. Your code should end by calling IOAllowPowerChange to permit the new power state or IOCancelPowerChange to disallow it; these functions are defined in pwr_mgmt/IOPMLib.h.

2. Elsewhere in your code (probably in an initializer), call one of the IOPMLib.h registration functions, depending on whether you want notification of system power events or notification of power-state changes affecting a particular device.

   - IORegisterForSystemPower for system-wide power events
   - IORegisterApp for power events affecting a particular device

   Both functions require you to pass in (among other parameters) a pointer to the callback function. They directly return an io_connect_t representation of the policy maker for the root domain or the device; indirectly, they return a reference to a port to receive notifications on (IONotificationPortRef).

3. Call the function IONotificationPortGetRunLoopSource, passing in the IONotificationPortRef, to get a reference to a run-loop source (CFRunLoopSourceRef). The IONotificationPortGetRunLoopSource function is defined in IOKitLib.h.

4. Add this run-loop source to the current run loop using the Core Foundation function CFRunLoopAddSource.

5. When your process has no more need for further notifications, deregister the process by calling the appropriate function, IODeregisterForSystemPower or IODeregisterApp.

Listing 9-3 (page 112) shows how you might have your application notified of system sleep events.

Listing 9-3 A user process getting notified of system sleep events

```
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>

#include <mach/mach_port.h>
#include <mach/mach_interface.h>
#include <mach/mach_init.h>

#include <IOKit/pwr_mgt/IOPMLib.h>
#include <IOKit/IOMessage.h>

io_connect_t    root_port;


void callback(void * x,io_service_t y,natural_t messageType,void *
messageArgument)
{
    printf("messageType %08lx, arg %08lx\n",(long unsigned int)messageType,
(long unsigned int)messageArgument);

    switch ( messageType ) {
    case kIOMessageSystemWillSleep:
        // Handle demand sleep (such as sleep caused by running out of
        // batteries, closing the lid of a laptop, or selecting
        // sleep from the Apple menu.
        IOAllowPowerChange(root_port,(long)messageArgument);
        break;
    case kIOMessageCanSystemSleep:
        // In this case, the computer has been idle for several minutes
        // and will sleep soon so you must either allow or cancel
        // this notification. Important: if you don't respond, there will
        // be a 30-second timeout before the computer sleeps.

        // IOCancelPowerChange(root_port,(long)messageArgument);
        IOAllowPowerChange(root_port,(long)messageArgument);
        break;
    case kIOMessageSystemHasPoweredOn:
        // Handle wakeup.
        break;
    }

}


int
main(int argc, char **argv)
{
    IONotificationPortRef   notify;
    io_object_t             anIterator;

    root_port = IORegisterForSystemPower (0,&notify,callback,&anIterator);
    if ( root_port == NULL ) {
            printf("IORegisterForSystemPower failed\n");
            return 1;
    }


    CFRunLoopAddSource(CFRunLoopGetCurrent(),
                        IONotificationPortGetRunLoopSource(notify),
                        kCFRunLoopDefaultMode);

    printf("waiting...\n\n");

    CFRunLoopRun();

    return 0;
}
```

Keep in mind that when your application gets a notification of a power-state change, it *should* respond. If an application does not respond, the system times out after 30 seconds and interprets the non-response to either the `kIOMessageSystemWillSleep` or `kIOMessageCanSystemSleep` message as a call to `IOAllowPowerChange`. Your application should not ignore the message and rely on the default interpretation of a non-response, however: The delay a non-response causes can considerably degrade the user experience. If any application vetoes the power change (by calling `IOCancelPowerChange`), the system will not sleep.

Of course, certain situations demand a power-state change, such as when the user shuts down the system. In these cases, power management tells applications (via notification) that they must prepare for the change. Calling `IOCancelPowerChange` has no effect, and if applications fail to respond positively, power management enacts the power-state change anyway.

In addition to information about system-wide power-state changes, applications can also get information about both external and internal power sources using the I/O Kit's `IOPowerSources` API (located in `/System/Library/Frameworks/IOKit.framework/Headers/ps/IOPowerSources.h`). Using the methods defined in the header file, an application can get a list of attached power sources, receive notification when its power sources change, and determine how much power is left in an internal or external battery.

Responding to Device Removal

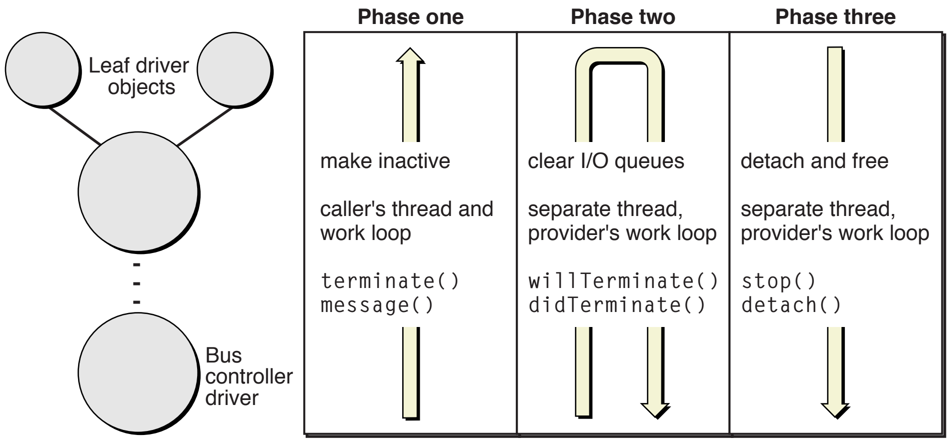
Mac OS X is an operating system that includes hot-swapping as a feature. Users can plug in and remove external devices (for example, mass-storage drives, CD-RW drives, modems, and scanners) and the system immediately does what is necessary to make the device usable or, in the case of removal, to register the absence of the device. No system restart or shutdown is necessary.

When a user plugs a device into the system, the I/O Kit responds to the event using the normal process for discovering and loading drivers. A low-level driver scans its bus, notices a new device, and kicks off the matching process to find a suitable driver. The I/O Kit then loads the driver into the kernel and the device is usable.

When a user removes a device, however, the situation is different. A driver stack must be torn down rather than built up. Before the drivers in the stack can be released, they must, in a coordinated manner, stop accepting new requests and clear out all queued and in-progress work; this requires a special API and procedure.

The I/O Kit performs an orderly tear-down of a driver stack upon device removal in three phases. The first phase makes the driver objects in the stack inactive so they receive no new I/O requests. The second phase clears out pending and in-progress I/O requests from driver queues. Finally, in the third phase, the I/O Kit invokes the appropriate driver life-cycle methods on drivers to clean up allocated resources and detach the objects from the I/O Registry before freeing them. Figure 9-3 (page 115) summarizes what happens during the three phases, including the calling direction within the driver stack.

Figure 9-3 Phases of device removal



Making Drivers Inactive

Just as a bus controller driver scans its bus to detect a newly inserted device, it also detects devices that have just been removed. When this happens, it calls `terminate` on its client nub; the `terminate` method has the default behavior of making the called object inactive immediately. The `terminate` method is also recursively invoked on clients; it is called on each object in the stack above the bus controller until all objects in the stack are made inactive.

As a consequence of being made inactive, each object also sends its clients (or, in rare cases, providers) a `kIOServicesIsTerminated` message via the `message` method. When the `terminate` call returns to the original caller (the bus controller driver), all objects in the stack are inactive, but the stack is still attached to the I/O Registry.

The I/O Kit assumes that objects that have multiple providers (drivers of RAID devices, for instance) do not want to be torn down and thus does not call `terminate` on them. If these objects do want to receive the `terminate` message, they should implement the `requestTerminate` method to return `true`.

The `terminate` call is asynchronous to avoid deadlocks and, in this first phase, takes place in the thread and work-loop context of the caller, the bus controller driver.

Clearing I/O Queues

The I/O Kit itself coordinates the second phase of the device-removal procedure. It starts with the newly inactive client of the bus controller driver and, as in the first phase, moves up the driver stack until it reaches the leaf objects. It calls the `willTerminate` method on each object it encounters. Drivers should implement the `willTerminate` method to clear out any queues of I/O requests they have. To do this, they should return an appropriate error to the requester for each request in a queue.

After `willTerminate` has been called on each object, I/O Kit then reverses direction, going from the leaf object (or objects) to the root object of the driver stack, calling `didTerminate` on each object. Certain objects at the top of the stack—particularly user clients—may decide to keep a count of I/O requests they have issued and haven't received a response for ("in-flight" I/O requests). (To ensure the validity of this count, the object should increment and decrement the count in a work-loop context.)

By this tracking mechanism, they can determine if any I/O request hasn't been completed. If this is the case, they can implement `didTerminate` to return a deferral response of `true`, thereby deferring the termination until the final I/O request completes. At this point, they can signal that termination should proceed by invoking `didTerminate` on themselves and returning a deferral response of `false`.

If a driver attaches to a client nub and has it open, the I/O Kit assumes a deferred response (`true`) to `didTerminate`. The termination continues to be deferred until the client driver closes its provider.

At the end of this second phase, there shouldn't be any I/O requests queued or "in flight." The I/O Kit completes this phase of the device-removal procedure on its own separate thread and makes all calls to clients on the work-loop context of the provider.

Detaching and Releasing Objects

In the third phase of the procedure for device removal, the I/O Kit calls the driver life-cycle methods `stop` and `detach` (in that order) in each driver object in the stack to be torn down, starting from the leaf object (or objects). The driver should implement its `stop` function to close, release, or free any resources it opened or created in its `start` function, and to leave the hardware in the state the driver originally found it. The driver can implement `detach` to remove itself from its nub through its entry in the I/O Registry; however, this behavior is the default, so a driver usually does not need to override `detach`. The `detach` method usually leads immediately to the freeing of the driver object because the I/O Registry typically has the final `retain` on the object.

The I/O Kit completes this phase of the device-removal procedure on its own separate thread and makes all calls to clients on the work-loop context of the provider.

I/O Kit Family Reference

This appendix describes each of the I/O Kit families in detail, paying particular attention to client/provider relationships. For most families, it provides a class hierarchy chart. It also tells you if a family exports a device interface, thereby allowing applications to access devices represented by the family. You should seriously consider taking the device-interface approach before attempting to write a kernel-resident driver. For information on using device interfaces, see the document *Accessing Hardware From Applications*.

Some categories of devices are not currently supported by an I/O Kit family. If your device falls into an unsupported category, you might be able to write a "family-less" driver, use an SDK other than the I/O Kit, or create a new family. See "Devices Without I/O Kit Families" (page 142) for details.

You may find it helpful to examine the source code for an I/O Kit family or a specific device driver. To do this, visit Darwin Releases, select the appropriate version of Mac OS X, and click Source to view the available source projects.

ADB

The ADB family provides support for, and access to, devices attached to the Apple Desktop Bus (ADB). It provides an abstraction for ADB bus controller drivers (IOADBController) and another for drivers of ADB devices (IOADBDevice).

Headers at:

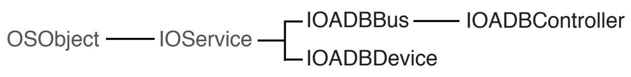
- Kernel resident: `Kernel.framework/Headers/IOKit/adb/`
- Device interface: `IOKit.framework/Headers/adb`

References and specifications:

- Chapter 5 — "ADB Manager" of *Inside Macintosh: Devices* (http://developer.apple.com/documentation/Hardware/DeviceManagers/adb/adb.html)
- Technical note HW01 (http://developer.apple.com/technotes/hw/hw_01.html)
- *Guide to the Macintosh Family Hardware*, second edition, Apple Computer.

Class hierarchy:

ADB family



Device Interface:

- Exports an interface for reading and writing registers on ADB devices. The interface is defined in `IOADBLib.h`. Only polled mode operations are supported through this library. Interrupt operations are only supported for kernel-resident clients.

Table A-1 Clients and providers of the ADB family

	Client of the nub	Provider for the nub
Action	Drives devices that plug into an ADB port.	Drives an ADB bus controller
Example	A driver for an ADB mouse is a *client* of the ADB family but is a *member* of the HID family (in the IOHIPointing class).	
Classes	An instance of IOADBDevice matches your driver and loads it into the kernel. Your driver communicates with its family through an instance of IOADBDevice.	ADB bus controller drivers should inherit from the IOADBController class as defined in the header file `IOADBController.h`
Notes	Common client families include the HID family (IOHIPointing and, IOHIKeyboard classes) and the Graphics family (IODisplay class).	All current ADB bus hardware produced by Apple is well supported by drivers that are included with Mac OS X. Third-party developers should not need to write drivers for the ADB family, except for ADB-USB adaptors.

ATA and ATAPI

The ATA and ATAPI family provides support for ATA controllers, and access to ATA and ATAPI devices on the ATA bus.

Important: The ATA and ATAPI family is still under development. The information in this section is subject to change.

Headers in:

- Kernel resident: `Kernel.framework/Headers/IOKit/ata`

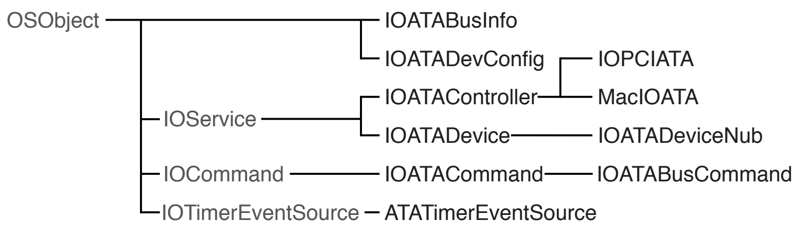
References and specifications:

- American National Standards Institute (ANSI)—http://www.ansi.org

- National Committee for Information Technology Standards (NCITS)—http://www.ncits.org
- Technical Committee 13 (T13)—http://www.t13.org

Class hierarchy:

ATA family



Device interface:

- Although this family does not itself export device interfaces, the "SCSI Architecture Model" (page 132) family does provide device-interface support for ATAPI devices.

Table A-2 Clients and providers of the ATA and ATAPI family

	Client of the nub	Provider for the nub
Action	Drives a device that is connected to an ATA bus.	Drives an ATA bus controller.
Example	An ATA hard drive or an ATAPI DVD-ROM drive. The Storage family is the most common family for clients.	
Classes	Clients of this family match against an instance of IOATADevice, one of which is created and published by the controller driver for every ATA or ATAPI device that is detected on the bus. They use the services provided by that object to communicate with the physical device on the bus.	
Notes	Clients of the ATA family must issue ATA/ATAPI commands encapsulated by an IOATACommand object. This command object encapsulates all the information necessary to encode a single ATA/ATAPI command as well as the results of the command's execution.	Third-party developers should never need to create a member of the ATA/ATAPI family.

Audio

The Audio family provides support to enable access to devices that record or play back audio signals. It provides a flexible abstraction for audio devices that permits an unlimited number of channels as well as arbitrary sample rates, bit depths, and sample formats. The Audio family utilizes a high-resolution time base that is used as the basis for timing information for the entire audio and MIDI system in Mac OS X. (The Audio family itself does not provide any MIDI services; these are provided by the Core MIDI framework.)

The Audio Hardware Abstraction Layer (Audio HAL) provides all audio services to applications in Mac OS X. The Audio HAL is accessed through the Core Audio framework and has its programmatic interface defined in `AudioHardware.h` in that framework. The Audio family provides the link between an audio driver and the Audio HAL. Because the Audio HAL is a client of the Audio family, all audio device functionality is available to clients of the Audio HAL.

Headers in:

- Kernel resident: `Kernel.framework/Headers/IOKit/audio/`

Device interface:

- Although this family does not directly export device interfaces, the Audio family does provide a device interface that is used by the Audio Hardware Abstraction Layer (Audio HAL) to access all of the abstractions provided by the Audio family (see description above).

Table A-3 Clients and providers of the Audio family

	Client of the nub	Provider for the nub
Action	A kernel-resident client is not necessary. Use the Core Audio framework to access Audio HAL.	Either records or plays back audio signals.
Example		PCI audio cards, external USB or FireWire audio devices and any other device that produces or consumes audio.
Classes		An audio driver must contain subclasses of both IOAudioDevice and IOAudioEngine.
Notes		An audio driver is a client of other families that provide access to the hardware that the driver supports. For example, a driver for a PCI audio card will be a client of the PCI family.

FireWire

The FireWire family provides support for, and access to, devices attached to the FireWire bus (FireWire is an Apple trademark applied to the IEEE 1394 standard, also sometimes known as i.LINK™).

The FireWire family has strong affinities with the SBP2 family. A driver that uses the SBP-2 transport protocol is the most common client of the FireWire family.

Headers in:

- Kernel-resident: `Kernel.framework/Headers/IOKit/firewire/`

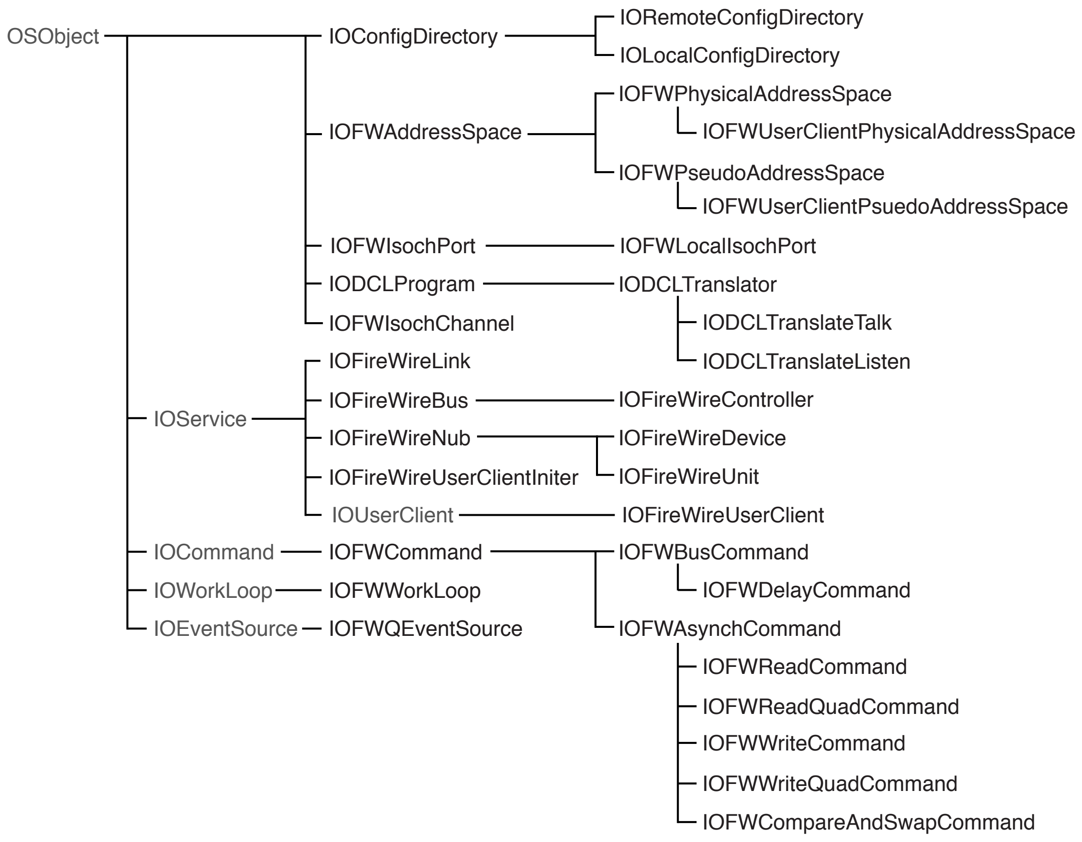
- Device interface: `IOKit.framework/Headers/firewire`

References and specifications:

- 1394 Trade Association—http://www.1394ta.org
- IEEE—http://www.ieee.org/portal/site. Copies of the IEEE 1394 specification can be purchased here.
- Apple Developer Connection—http://developer.apple.com/hardwaredrivers/firewire/index.html

Class hierarchy

Firewire family



Device interface:

- Provides a device interface exporting an interface for sending and receiving packets on the FireWire bus and for adding entries into the computer's own FireWire config ROM.

Table A-4 Clients and providers of the FireWire family

	Client of the nub	Provider for the nub
Action	*Unit drivers:* Drives communication with a unit of a device that plugs into the FireWire bus. *Protocol drivers:* Adds support for a protocol enabling peer-to-peer communication or emulation over FireWire.	Drives a FireWire bus controller.
Example	*Unit drivers:* A driver for a FireWire speakers. *Protocol drivers:* A driver that provides TCP/IP over FireWire.	
Classes	*Unit drivers:* An instance of IOFireWireUnit, representing a unit found in a device's configuration ROM, is matched against the driver and it is loaded into the kernel. The driver communicates with the FireWire family through this instance. *Protocol drivers:* An instance of IOFireWireController matches a protocol driver and loads it into the kernel. The driver communicates with the FireWire family through this instance.	Driver classes should inherit from the IOFireWireController class as defined in the header file `IOFireWireController.h`.
Notes	The most common client family is the SBP2 family.	Mac OS X ships drivers for all Open Host Controller Interface (OHCI) bus controllers. Third-party developers generally should not need to write bus-controller drivers for the FireWire family.

In some cases, you can write a driver for a FireWire device instead of for a unit. An example might be a driver for a device with a minimal config ROM (that is, with just a vendor ID). However, use of the minimal config ROM is strongly discouraged by Apple. Also, if your driver matches against a FireWire unit, it is often possible to do some things with the device.

Graphics

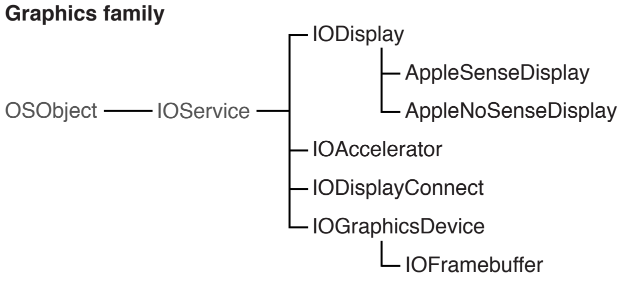
The Graphics family provides support for frame buffers and display devices (monitors).

Headers in:

- Kernel resident: `Kernel.framework/Headers/IOKit/graphics/`
- Device interface: `IOKit.framework/Headers/graphics`

Class hierarchy:

Graphics family



Device interface:

- The Graphics family exports several device interfaces since most access of graphics devices is from user space. However, the Quartz layer arbitrates access to frame buffers from user space through the windowing system or the CGDirectDisplay API. Other layers, such as Carbon Draw Sprockets, provide application access to graphics.
- Graphics acceleration is supplied by modules loaded into user address space. A CFPlugIn interface, defined in `IOGraphicsInterface.h`, implements two-dimensional acceleration. Similarly, OpenGL defines a loadable-bundle interface for three-dimensional rendering. Because there is no standard way to implement this functionality, hardware-specific code can exist in both user-space code and in a kernel-loaded driver.

Table A-5 Clients and providers of the Graphics family

	Client of the nub	Provider for the nub
Action		Implements support for a frame buffer.
Example		
Classes		Frame-buffer drivers must be subclasses of the IOFramebuffer class. The IONDRVFramebuffer class supports native Power PC Mac OS graphics drivers (known as “`ndrv`"s); this support is automatic, provided the drivers are written correctly to the specification.
Notes	Support for kernel-resident clients is limited. Because the Quartz layer owns the display, the kernel generally does not render graphics directly.	Apple provides generic support for displays and so displays should not generally require third-party drivers.

A Note on NDRV Compatibility

NDRV graphics drivers should function in OS X if they are correctly written. If they are not correctly written, the many differences in Mac OS X’s runtime environment could cause them to fail, be ignored, or even cause a crash. If you are writing an NDRV driver, follow these rules:

- Access the card hardware using virtual addressing. Do not assume the card is mapped into its physically assigned address. In Mac OS 9, NDRV cards are mapped one-to-one, but in Mac OS X, this is not guaranteed. Obtain the virtual addresses for your card's hardware via the `AAPL,address` property as documented in "Designed Cards and Drivers for PCI Power Macintosh."
- Link only on the native driver libraries, which are NameRegistryLib, DriverServicesLib, and VideoServicesLib. If your card links on InterfaceLib or any other application-level library, it probably won't work on Mac OS X.
- Do not access low memory; doing so causes a crash (kernel panic) in Mac OS X.
- Name registry calls are not supported from interrupt level in Mac OS 9 or Mac OS X. They return errors in Mac OS X.
- Secondary interrupts are not supported in Mac OS X. There is no need to fake vertical blank interrupts if your card does not support them—simply do not create a VBL service. Mac OS 9 continues to require a VBL service to be installed to move the cursor on your device.
- Stack size is limited to 16K on Mac OS X; any NDRV invocation should consume no more than 4K of stack.

If you want to make runtime conditional changes to your NDRV code, the property `AAPL,iokit-ndrv` is set in the PCI device properties before OS X uses your driver.

Mac OS X supports 32 bits-per-pixel, alpha-blended cursors in hardware. If your device supports an alpha-blended direct color cursor, it should call `VSLPrepareCursorForHardwareCursor` with these fields set in the `HardwareCursorDescriptor` record:

```
bitDepth = 32maskBitDepth = 0numColors = 0colorEncodings = NULL
```

The `hardwareCursorData` buffer in the `HardwareCursorInfo` should point to a buffer of 32 bits per pixel, ARGB data. The data is not premultiplied by the alpha channel.

HID

The Human Interface Device (HID) class is one of several device classes described by the USB (Universal Serial Bus) architecture. The HID class consists primarily of devices humans use to control a computer system's operations. Examples of such HID class devices include:

- Keyboards and pointing devices such as mice, trackballs, and joysticks
- Front-panel controls such as knobs, switches, sliders, and buttons
- Controls that might be found on games or simulation devices such as data gloves, throttles, and steering wheels

Currently, Mac OS X provides the HID Manager to allow applications to access joysticks, audio devices, and non-Apple displays. You can also use the HID Manager to get information from another type of HID class device, a UPS (uninterruptible power supply) device. UPS devices share the same report descriptor structure as other HID class devices and provide information such as voltage, current, and frequency. The Mac OS X HID Manager consists of three layers:

- The HID Manager client API that provides definitions and functions your application can use to work with HID class devices

- the HID family that provides the in-kernel HID infrastructure such as the base classes, the kernel-user space memory mapping and queueing code, and the HID parser
- the HID drivers provided by Apple

Headers in:

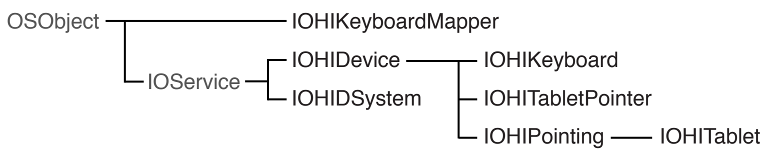
- Kernel resident: `Kernel.framework/Headers/IOKit/hid/` and `Kernel.framework/Headers/IOKit/hidsystem/`
- Device interface: `IOKit.framework/Headers/hid/` and `IOKit.framework/Headers/hidsystem/`

References and specifications:

- HID Information section of USB.org—Developers website (http://www.usb.org/developers/hidpage)
- *HID Class Device Interface Guide*

Class hierarchy:

Human Interface Device family



Device interface:

- The HID family exports a device interface through the HID Manager client API. The HID Manager includes `IOHIDLib.h` and `IOHIDKeys.h` (located in `/System/Library/Frameworks/IOKit.framework/Headers/hid`) which define the property keys that describe a device, the element keys that describe a device's elements, and the device interface functions and data structures you use to communicate with a device. After you've created a device interface for a selected HID class device, you can use the device interface functions to open and close the device, get the most recent value of an element, or set an element value.

Table A-6 Clients and providers of the HID family

	Client of the nub	Provider for the nub
Action		Drives an input device such as a multi-button mouse, trackball, or joystick.
Example		
Classes		
Notes		Support for most simple input devices is provided by the generic driver.

Network

The Network family provides support for network controllers. The Network family consists of two logical layers:

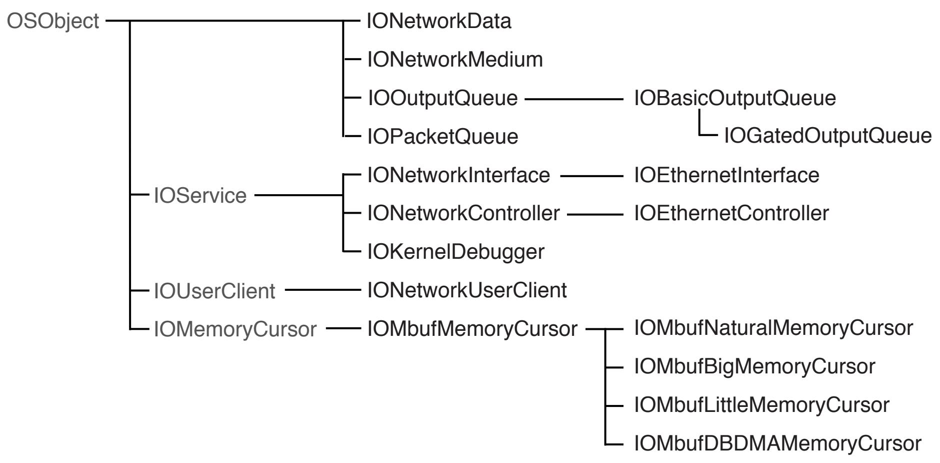
- Controller layer—this layer represents the network controller.
- Interface layer—this layer represents the network interface published by the network controller.

Headers in:

- Kernel resident: `Kernel.framework/Headers/IOKit/network/`
- Device interface: `IOKit.framework/Headers/network/`

Class hierarchy:

Network family



Device interface:

- The device interface for this family is usually the BSD network stack. Applications use the socket interface provided by the network stack to access indirectly the services provided by the Network family.

Power management:

- The policy maker is implemented in the IONetworkController class. IONetworkInterface (and subclasses) and IOKernelDebugger are "interested drivers" that take appropriate action when the power controller is about to change the power state and after the power state is changed.

Table A-7 Clients and providers of the Network family

	Client of the nub	Provider for the nub
Action		Drives a network controller.

	Client of the nub	Provider for the nub
Example		Controllers on Ethernet, Token Ring, and FDDI adapters.
Classes		Driver must be an instance of a subclass of a controller class that implements generic network controller functionality, such as IONetworkController or of a controller class that builds upon IONetworkController to specialize for Ethernet controller support (IOEthernetController). See discussion on Network family classes below for more information.
Notes	Drivers are typically not clients of the Network family. The primary system client of this family is the DLIL (Data Link Interface Layer) module in the BSD network stack.	

Member drivers must also create IONetworkInterface objects that are registered with the DLIL; such registration associates the driver with a network interface (for example, `en0`) in the system. You can create a Network Kernel Extension (NKE) and insert it at various locations above the IOKit/DLIL boundary to intercept the packets, commands, or other event traffic between an IONetworkInterface object and the upper layers.

Another client of this family is the KDP (Kernel Debugger Protocol) module in the kernel. A driver can create an IOKernelDebugger object that vends debugging services and allows kernel debugging through the network hardware managed by the driver. Only drivers that drive a built-in network controller are required to provide this support.

Other classes of the Network family include:

IOOutputQueue—assists in queuing outbound packets.
IOPacketQueue—represents a generic `mbuf` packet FIFO queue.
IOMbufMemoryCursor—translates `mbuf` packets into a scatter-gather list of physical addresses and length pairs.
IONetworkData—represents a container for a single group of statistics counters.
IONetworkMedium—represents a single network medium supported by the network device.

PC Card

The PC Card family supports 32-bit PC cards (CardBus), 16-bit PC cards (I/O and memory), and Zoom Video cards. This support encompasses controllers that are compatible with ExCA (Intel 82365) and Yenta register sets. Apple's PC Card family's Card Services are, for the most part, compliant with the 1997 PC Card™ standard.

CardBus cards are essentially PCI devices in a different form factor. If you are writing a driver for a CardBus card, you can choose to subclass from either IOPCIDevice or IOCardBusDevice. See the reference section "`PCI and AGP`" (page 128) for more information about the PCI family.

Other classes provided by the family include

IOPCCard16Device—represents a 16-bit PC card device.

IOPCCard16Enabler—allows you to override the card configuration process for 16-bit cards. It is used mainly for cards with broken or missing CIS entries.

IOZoomVideoDevice—represents a Zoom Video device.

IOPCCardBridge—represents a PC Card bridge; this class is a subclass of IOPCI2PCIBridge which is a subclass of IOPCIBridge.

Headers in:

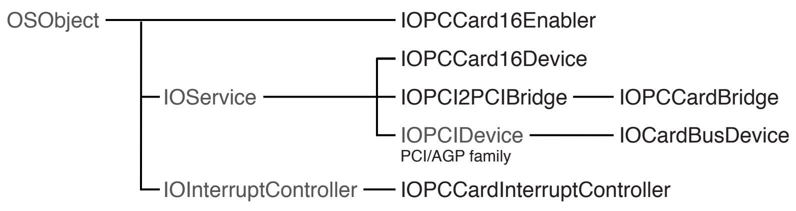
- Kernel resident: `Kernel.framework/Headers/IOKit/pccard/`

References and specifications:

- Documentation for Card Services can be found in the `doc` directory of the PC Card family source (available via Darwin CVS). You can also find in the same location a sample 16-bit driver.
- Apple Developer Connection—http://developer.apple.com/hardwaredrivers/pci/

Class hierarchy:

PC Card family



Device interface:

- None.

Power management:

- Classes IOPCCardBridge, IOPCCard16Device, IOPCCard16Enabler, and IOCardBusDevice implement policy makers for power management.

PCI and AGP

The PCI and AGP family provides support for, and access to, devices attached to PCI and AGP buses and PCI bridges.

Headers in:

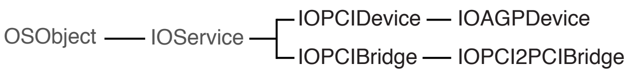
- `Kernel.framework/Headers/IOKit/pci/`

References and specifications:

- This family supports all major features of the PCI Localbus 2.1 specification.
- PCI Special Interest Group—http://www.pcisig.com
- Apple Developer Connection—http://developer.apple.com/hardwaredrivers/pci/

Class hierarchy:

PCI family



Device interface:

- None. Applications are not permitted direct access to PCI bus hardware for security reasons. Instead, applications should interact with higher-level services, such as those provided by device interfaces of the USB or other families.

Matching properties:

- The PCI/AGP family permits matching based on Open Firmware or on PCI registers. See the description of the IOPCIDevice class in the reference documentation for details.

Check the Darwin Open Source project for example PCI drivers.

Table A-8 Clients and providers of the PCI and AGP family

	Client of the nub	Provider for the nub
Action	Drives a device that plugs into a PCI bus.	Drives a PCI bus controller or a PCI bridge.
Example	A driver for a PCI SCSI controller card is a *client* of the PCI family but is a *member* of the SCSI Parallel family.	
Classes	The driver communicates with its family via an instance of IOPCIDevice or IOAGPDevice. An instance of one of these classes matches your driver and loads it into the kernel.	PCI family member drivers should inherit from the IOPCIBridge class.
Notes	The most common client families are the USB, Network, SCSI, Graphics, and Audio families.	Mac OS X supports most PCI bus hardware with a set of generic drivers. In general, third-party developers do not need to write drivers for the PCI and AGP family unless they are building a PCI expansion chassis or developing drivers for a PCI bridge with special characteristics not addressed by the generic drivers.

SBP-2

The SBP2 family (Serial Bus Protocol 2) provides support for, and access to, devices attached to the FireWire bus that use the SBP-2 transport protocol. The SBP2 family is a client of the FireWire family. SBP-2 devices require FireWire to run.

Headers in:

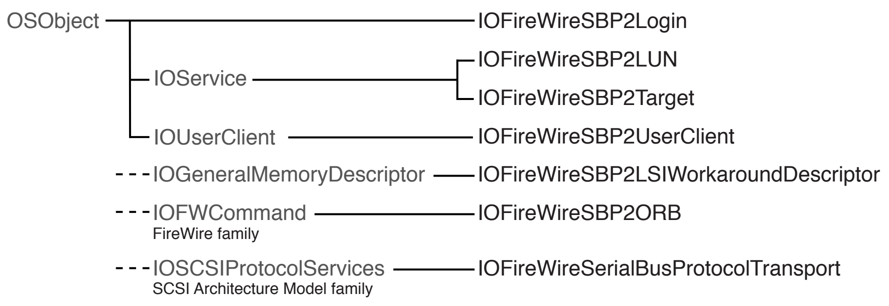
- Kernel resident: `Kernel.framework/Headers/IOKit/sbp2/`
- Device interface: `IOKit.framework/Headers/sbp2`

References and specifications:

- T10 Technical Committee —http://www.t10.org
- SBP-2 standard: ftp://ftp.t10.org/t10/drafts/sbp2/

Class hierarchy:

SBP2 family



Device interface:

- The SBP2 family provides a device interface that exports an interface for sending SBP-2 ORBs to a device.

Table A-9 Clients and providers of the SBP2 family

	Client of the nub	Provider for the nub
Action	Drives a device that uses the SBP-2 transport protocol.	Provides SBP-2 transport services
Example	A driver for a typical FireWire hard disk drive is a *client* of the SBP2 family but is a *member* of the mass storage family.	

	Client of the nub	Provider for the nub
Classes	A client driver communicates with the SBP2 family through an instance of IOFireWireSBP2LUN. An instance of this class is created for each LUN (Logical Unit Number) found in a configuration ROM unit directory; this instance matches your driver and loads it into the kernel.	SBP2 family-member drivers should inherit from the IOFireWireSBP2Target class.
Notes	The most common client family is the Storage family.	

SCSI Parallel

The SCSI Parallel family provides support for, and access to, devices attached to a parallel SCSI bus. This family supports all major features of the SCSI Parallel Interface-5 (SPI-5) specification for parallel buses.

Headers in:

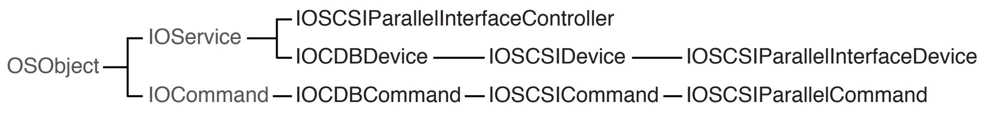
- Kernel resident: `Kernel.framework/Headers/IOKit/scsi/spi`. The header file `IOSCSIParallelInterfaceController.h` supports the development of a driver for a SCSI card.

References and specifications:

- T10 Technical Committee —http://www.t10.org
- SCSI Technical Library of Information—http://www.scsilibrary.com/

Class hierarchy:

SCSI Parallel family



Device interface:

- Device interface support for parallel SCSI devices is provided by the SCSI Architecure Model family. See "SCSI Architecture Model" (page 132).

Table A-10 Clients and providers of the SCSI Parallel family

	Client of the nub	Provider for the nub
Action	Drives a device that plugs into a SCSI bus.	Drives a SCSI host adapter or controller chip.

	Client of the nub	Provider for the nub
Example	A driver for a SCSI disk drive is a *client* of the SCSI Parallel family but a *member* of the Storage family.	
Classes	A client driver communicates with the SCSI Parallel family through an instance of IOSCSIDevice. Your driver would match on this instance and be loaded into the kernel. Another class of interest is IOSCSICommand.	Because the SCSI Parallel family presently supports parallel buses, member drivers use the IOSCSIParallelInterfaceController class.
Notes	Common client families include the Transport Drivers for storage devices.	Mac OS X includes generic drivers for most built-in SCSI hardware. In general, third-party developers do not need to write drivers that are members of the SCSI Parallel family unless they are developing drivers for an expansion card.

SCSI Architecture Model

The SCSI Architecture Model family provides common client support for SCSI, USB (Storage), FireWire SBP-2 and ATAPI devices. Many of the classes of this family belong to the Transport Driver layer, which is summarized in Table A-11 (page 132).

Table A-11 SCSI Architecture Model family—Transport Driver layer

Type	Classes	Comments
Device Services Linkage	IOBlockStorageServices IOReducedBlockServices IOCompactDiscServices IODVDServices	Linkage objects that understand the APIs from the Device Services layer (Storage family) and export APIs that the Logical Unit drivers (in the Transport Driver layer) understand.
Logical Unit drivers	IOSCSIPeripheralDeviceType00 IOSCSIPeripheralDeviceType05 IOSCSIPeripheralDeviceType07 IOSCSIPeripheralDeviceType0E	Drive mass storage devices that have file systems or are bootable.
Peripheral Device Type Nub	IOSCSIPerpheralDeviceNub	Not really an I/O Kit nub, but an object that queries the device and determines which Logical Unit driver is needed.
SCSI Protocol drivers	IOUSBMassStorageClass IOATAPIProtocolTransport IOFireWireSerialBusProtocol-Transport IOSCSIParallelInterface-ProtocolTransport	Classes for bus-specific drivers. Although these classes belong to other families, they are part of the SCSI Architecture Model layering.

The Logical Unit drivers and the Peripheral Device Type Nub are in the SCSI Application layer, which inherits (ultimately) from IOSCSIPrimaryCommandsDevice. The SCSI Protocol drivers are in the SCSI Protocol layer, which inherits from IOSCSIProtocolServices.

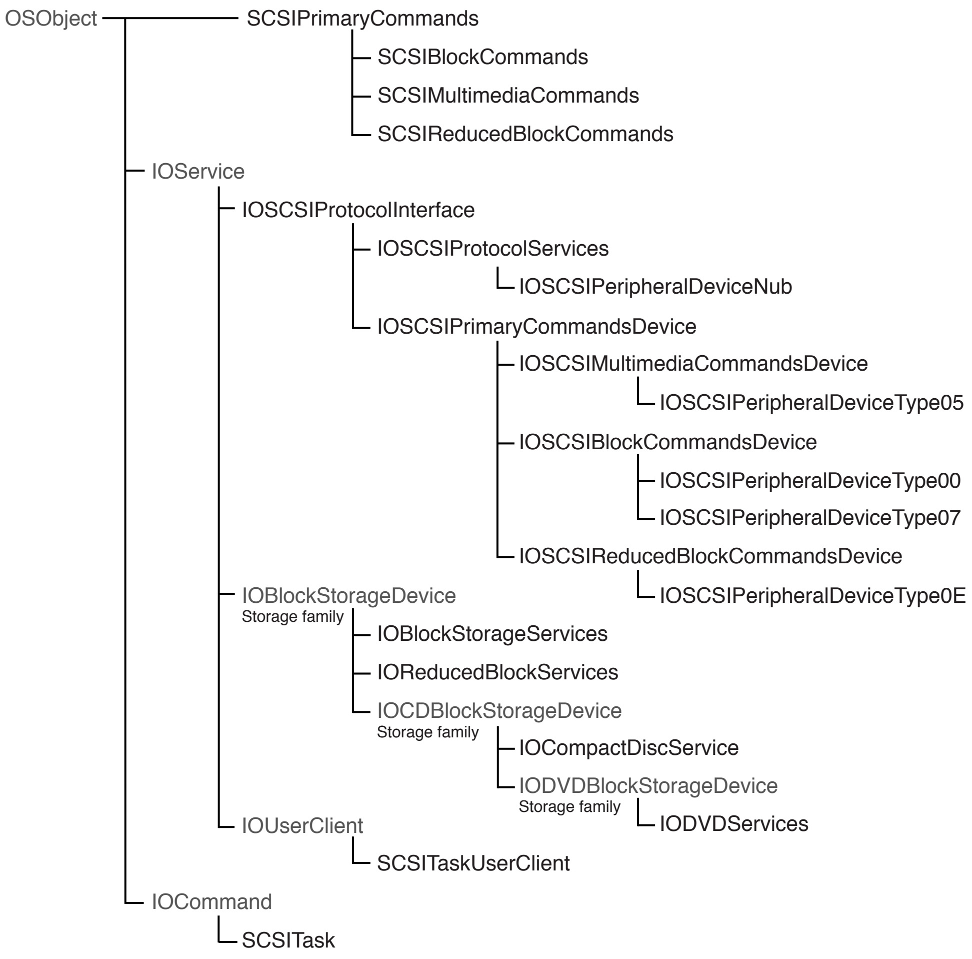
The SCSI Architecture Model family supports multimedia command set devices, such as CD-RW drives.

Headers in:

- Kernel-resident: `Kernel.framework/Headers/IOKit/scsi-commands/`
- Device interface: `IOKit.framework/Headers/scsi-commands`

Class hierarchy:

SCSI Architecture Model family



Device interface:

- The library for the SCSI Architecture Model family is called SCSITaskLib. It includes three interfaces: MMCDeviceInterface, SCSITaskDeviceInterface, and SCSITaskInterface. The user-client class is SCSITaskUserClient. You should use the SCSI parallel device interface (that is, the old

IOCDBInterface) instead for parallel devices which are *not* one of the following peripheral device types: 00h, 01h, 05h, 07h, or 0Eh. As an example, you would use the SCSI parallel device interface for a SCSI parallel scanner or printer. Note that at some future date all SCSI device interfaces will use the SCSITaskUserClient model.

Power management:

- The classes IOSCSIMultimediaCommandsDevice, IOSCSIBlockCommandsDevice, and IOSCSIReducedBlockCommandsDevice implement a policy maker.

Serial

The Serial family provides support for serial byte character streams.

Headers in:

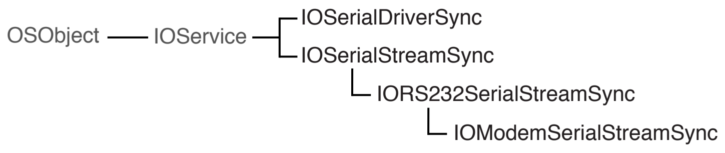
- Kernel resident: `Kernel.framework/Headers/IOKit/serial/`
- Device interface: `IOKit.framework/Headers/serial/`

References and specifications:

- See `termios(4)`. Also see the related header file `Kernel.framework/Headers/sys/termios.h`

Class hierarchy:

Serial family



Device interface:

- Applications can access the Serial family through the BSD device nodes, the most common client of this family. An application can read and write data using the BSD device nodes in `/dev`. Data is also routed through to PPP via these device nodes. You can find keys and other properties for use in device access in `IOKit.framework/Headers/serial/IOSerialKeys.h`.

Table A-12 Clients and providers of the Serial family

	Client of the nub	Provider for the nub
Action	*Requires* a single-banded data streaming service with elementary flow control.	*Provides* a single-banded streaming service; in other words, it cannot be packet-based. although it may be bi-directional. The driver may also implement flow control. The driver must describe the services it is capable of providing.

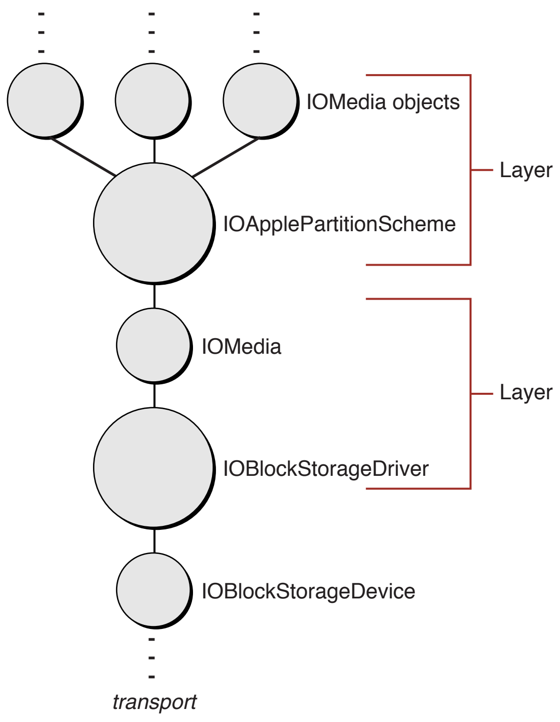
	Client of the nub	Provider for the nub
Classes		Serial port writers should subclass IOSerialDriverSync for their drivers, then publish as nubs objects of the IOModemSerialStreamSync or IORS232SerialStreamSync classes, or (if neither of these suffices) a concrete subclass of IOSerialStreamSync. The I/O Kit uses these objects to create the appropriate user-client interface for user-space access via BSD.
Notes	Developers should use the BSD device file mechanism documented in *Accessing Hardware From Applications*.	

Storage

The Storage family provides high-level support for random-access mass storage devices. It is separate from the underlying technology that transports the data to and from the represented storage space. The interface to the underlying transport technology is declared in the abstract class IOBlockStorageDevice. Storage driver objects communicate all mass-storage requests across this interface, without having to have knowledge of, or involvement with, the commands and mechanisms used to communicate with the device or bus.

The scope of the Storage Family encompasses the IOBlockStorageDevice interface, at one end (provider direction), and the BSD interface at the other end (client direction), with various driver and media layers in the between. Figure A-1 (page 136) illustrates this stack.

Figure A-1 Storage family driver stack



Each layer consists of a set of two objects—a driver object and the child media object (or objects) it publishes. The IOStorage class is the common base class for both driver and media objects. It is an abstract class that declares the basic open, close, read, and write interfaces that subclasses are to implement. It establishes the protocol with which media objects can talk to driver objects without needing to be subclassed for each driver. The read and write interfaces provide byte-level access to the storage space.

The IOBlockStorageDriver class is the common base class for generic block storage drivers. It matches and communicates via an IOBlockStorageDevice interface, and connects to the remainder of the storage framework via the IOStorage protocol. It extends the IOStorage protocol by implementing the appropriate open and close semantics, deblocking for unaligned transfers, polling for ejectable media, implementing locking and ejection policies, creating and tearing down media object, and gathering and reporting statistics. The Storage family supports other basic types of drives, such as CD drives and DVD drives, through subclasses of the IOBlockStorageDriver. You rarely, if ever, need to subclass the generic block storage driver to handle device idiosyncrasies; rather, you should change the underlying transport drivers to correct any non-conforming behavior.

The IOMedia class is an abstraction of a random-access disk device. It is equivalent to the “device object” or “interface object” in other I/O Kit families.

- It represents the presence of a device, or a piece thereof.
- It presents an programmatic interface to that device.
- It is backed by a separate driver that implements the required functionality.

IOMedia provides a consistent interface for both real and virtual disk devices, for subdivisions of disks such as partitions, for supersets of disks such as RAID volumes, and so on. It extends the IOStorage class by implementing the appropriate open, close, read, write, and matching semantics for media objects. Its properties reflect the properties of real disk devices, including natural block size and writability.

The other driver and media layers in the Storage Family are known as filter schemes. These optional layers separate media objects, providing some kind of data manipulation or offset manipulation between media objects. These layers may stack arbitrarily on top one another, as they are both a client and a provider of media objects. Most third-parties will develop drivers for the filter scheme layer, if not for the underlying transport technology. Refer to "IOMedia Filter Schemes" (page 138) for more information on developing filter scheme drivers.

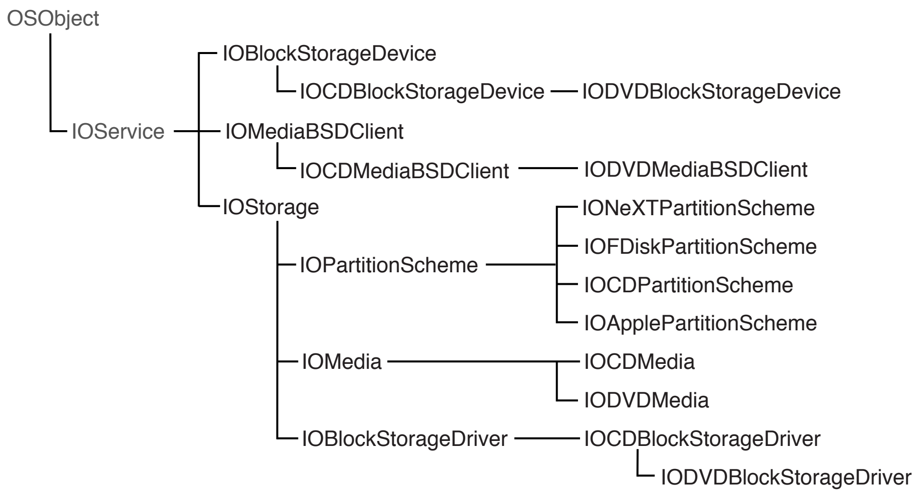
For more information on developing drivers for the underlying transport technology, refer to SCSI Architecture Model Family documentation. The SCSI Architecture Model Family is the common underlying transport technology for ATAPI, FireWire, SCSI, and USB. It provides a consistent CDB-based access model for application writers and driver writers, and a simple infrastructure for correcting device idiosyncrasies.

Headers in:

- Kernel resident: `Kernel.framework/Headers/IOKit/storage/`
- Device interface: `IOKit.framework/Headers/storage`

Class hierarchy:

Storage family



Device interface:

- See "Accessing IOMedia From Applications" (page 139)

IOMedia Filter Schemes

A filter scheme is a driver for IOMedia objects. It acts as both as a client and a provider of media objects. The filter-scheme driver receives mass storage requests through its abstract `read` and `write` member-function interfaces, in which it can perform the data manipulation or offset manipulation before passing the request on to its provider IOMedia object (or objects). There are several different kinds of media filter schemes:

- One-to-one—A block-level compression or encryption scheme, for example, would match against one IOMedia object and produce one child IOMedia object representing the uncompressed or unencrypted content.
- One-to-many—A partition scheme, for example, would match against one IOMedia object and produce multiple child IOMedia objects representing the content of each distinct partition (see "Partition Schemes" (page 138) for more information).
- Many-to-one—A RAID scheme, for example, would match against multiple IOMedia objects and produce one child IOMedia object representing the RAIDed content.
- Many-to-many—A many-to-many scheme would match against multiple IOMedia objects and produce multiple child IOMedia objects.

A filter-scheme driver inherits from the IOStorage class and necessarily participates in the IOStorage match category (IOMatchCategory's value in personality).

Partition Schemes

Mac OS X includes two standard partition schemes:

- IOApplePartitionScheme, the standard Apple partition scheme driver
- IOFDiskPartitionScheme, the standard PC partition scheme driver

A partition-scheme driver inherits from the IOPartitionScheme class (which inherits from IOStorage), matches against a single IOMedia parent, and produces one or more IOMedia child objects for each partition it must represent. It participates in the IOStorage match category, as with all filter scheme drivers.

A partition scheme need not be subclassed in order to make use of developer-defined content within a partition. A partition's contents is represented by a distinct IOMedia object, published as a child of the partition-scheme driver. Each child media object has properties that further identify information known about the partition, such the content hint, size, and natural block size. The content hint field is a string formed similarly to the well-known "Apple_Driver" and "Apple_HFS" strings, or by definition, in the form "MyCompany_MyContent". It permits partitions with developer-defined contents to be identified uniquely (when the partition is created), and permits filter-scheme drivers to match against such content automatically without ever probing a disk.

IOMedia Properties

Table A-13 (page 139) lists the standard set of properties for all IOMedia objects. These properties can be used as matching properties in I/O Kit's search and notification APIs, as well as for filter scheme driver matching purposes.

Table A-13 Storage family (IOMedia) properties

Key	Type	Description
`kIOMediaEjectableKey`	Boolean	Is the media ejectable?
`kIOMediaLeafKey`	Boolean	Is the media a leaf in the media tree? This is false whenever a client filter-scheme driver has matched against the media object.
`kIOMediaPreferred-BlockSizeKey`	Number	The media's natural block size in bytes.
`kIOMediaSizeKey`	Number	The media's entire size in bytes.
`kIOMediaWholeKey`	Boolean	Is the media at the root of the media tree? This is true for the physical media representation, a RAID media representation, and similar representations.
`kIOMediaWritableKey`	Boolean	Is the media writable?
`kIOMediaContentKey`	String	The media's content description, as forced upon by the client filter-scheme driver. (This content description is copied automatically from the client filter-scheme driver's `kIOMediaContentMaskKey` property.) The string's format follows the "MyCompany_MyContent" convention, and defaults to the Content Hint string should no client filter scheme have matched against the object. Used for informational purposes in user disk utilities.
`kIOMediaContent-HintKey`	String	The media's content description, as hinted at the time of the object's creation. The string's format follows the "MyCompany_MyContent" convention. Used for matching purposes in filter scheme drivers.
`kIOBSDNameKey`	String	The media's BSD device node name. The name is dynamically assigned at the time of the object's creation. Used for read and write access to the media's contents (see "Accessing IOMedia From Applications" (page 139)).

A media object also has a unique I/O Kit path, which can be obtained via standard I/O Kit APIs.

Accessing IOMedia From Applications

The standard user-space mechanism for accessing data on a piece of media is the BSD device interface. The BSD device interface is abstracted at the file-system layer through distinct read and write APIs, while general user application access is provided via the `read` and `write` system calls (see the corresponding man pages for more documentation). The properties and structure of a physical disk are represented in the I/O Kit Registry object hierarchy and in each media object's properties, including the BSD device interface name assigned to the media object.

A specific media object can be found via the standard I/O Kit search and notification APIs. The dictionary used to describe the IOMedia might refer to specific property values, to a specific service path or device tree path, or to a specific subclass of media. A CD, for example, appears as an IOCDMedia subclass, with properties appropriate to a CD, such as `kIOCDMediaTOCKey`. Such properties can be combined to describe a media object uniquely in the system, or generalized to identify a certain kind of media with multiple possible matches (returned in an iterator).

The Carbon APIs and Cocoa APIs provide mechanisms for sending notifications of file system mount and unmount events, as well as for file access within the file system. An application can obtain the associated BSD device interface name for a given file system through `GetVolParms` (`vMDeviceID`) in Carbon and through `getmntinfo` in BSD. The associated IOMedia object can be obtained for a given BSD device interface name through I/O Kit APIs using the `kIOBSDNameKey` property.

USB

The USB family provides support for, and access to, devices attached to a Universal Serial Bus (USB).

Two basic types of drivers are clients of this family: kernel-mode drivers and user-mode drivers. Kernel-mode drivers are required when the clients of the driver also reside in the kernel (such as HID devices, mass storage devices, or networking devices). User-mode drivers are preferred when only one process has access to the device (for example printers and scanners).

Headers in:

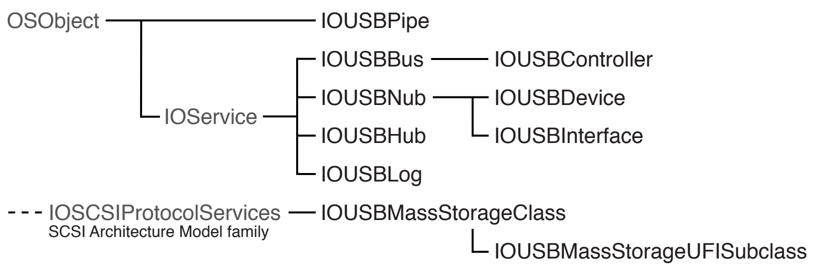
- Kernel resident: `Kernel.framework/Headers/IOKit/usb/`
- Device interface: `IOKit.framework/Headers/usb`

References and specifications:

- USB.org—http://www.usb.org; see especially USB Common Class Specification, revision 1.0, available for download at http://www.usb.org/developers/devclass_docs/usbccs10.pdf
- Apple Developer Connection—http://developer.apple.com/hardwaredrivers/usb/index.html

Class hierarchy:

USB family



Device interface:

- User mode clients use an API which is part of the I/O Kit framework; this API is defined in `IOKit/usb/IOUSBLib.h`. Clients use user mode abstractions of the IOUSBDevice and IOUSBInterface classes found in the kernel in order to communicate with the USB device or USB interface.

Kernel-resident drivers:

- Kernel drivers for physical USB devices can be written for either USB devices or for USB interfaces. A physical USB device consists of a device descriptor that can describe any number of interfaces. When writing a kernel-resident driver, you need to decide if the driver is to control the whole USB device or if it is to control only an interface of a USB device.

The USB family for kernel-resident drivers consists of three main classes:

- **IOUSBDevice**: The IOUSBDevice class is an abstraction of a physical USB device. There is one IOUSBDevice class instantiated for every USB device connected to the bus. The provider for an IOUSBDevice object is an IOUSBController object (which is an abstraction of a USB controller).
- **IOUSBInterface**: The IOUSBInterface class is an abstraction of one of the interfaces of a USB device. There is one of these classes instantiated for every interface in a device. When it is created, the IOUSBInterface creates IOUSBPipe objects for each endpoint described in the interface descriptor of the interface. The provider for an IOUSBInterface object is an IOUSBDevice object.
- **IOUSBPipe**: The IOUSBPipe class contains the methods that are used for communicating with a USB device or a USB interface. There is one IOUSBPipe object created for the default control endpoint and an additional one for each endpoint described in the interface descriptor. The provider for an IOUSBPipe object is an IOUSBInterface object.

Kernel-resident USB drivers are clients of the family which provides the transport services and are members of the family from which they get their class inheritance. For example, a driver for a USB keyboard is a client of the IOUSBInterface object that is, the IOUSBInterface object is the provider for the driver) but the keyboard driver is a member of the IOHIDFamily. The USB family provides the mechanism for getting at the key presses in the keyboard. The keyboard driver supports methods from the HID family for sending those key presses to the event system.

As mentioned above, you can write a driver to match against a USB device or a USB interface. The IOUSBDevice or IOUSBInterface classes are the providers for the drivers. The drivers themselves can be members of a separate family (such as the IOHIDFamily or IOAudioFamily) or can be members of the IOService family.

Table A-14 Clients and providers of the USB family

	Client of the nub	Provider for the nub
Action	Drives a device that plugs into a USB port.	Drives a USB bus controller.
Example	A keyboard driver is an interface driver; its provider is the driver for the USB *device*. The keyboard driver inherits from the IOHIKeyboard class—it's a member of the HID family.	

	Client of the nub	Provider for the nub
Classes	IOUSBDevice drivers can only communicate with the USB device through the default control pipe. IOUSBInterface drivers have IOUSBPipe objects created for all the endpoints that are described in the interface descriptor for the current configuration. The driver uses these objects to communicate with the device.	USB family-member drivers should inherit from the IOUSBController class.
Notes	Common client families include the HID family (IOHIPointing and IOHIKeyboard classes) and the Transport Drivers for Storage family devices. Most kernel-mode clients of the USB family are interface drivers, and only occasionally device drivers.	Mac OS X includes generic drivers that support all Open Host Controller Interface (OHCI) bus controllers. In general, third-party developers do not need to write drivers for the USB family.

Driver matching: The USB family uses the USB Common Class Specification, revision 1.0 to match devices and interfaces to drivers (for a link to this specification, see the section above titled "References and specifications"). The driver should use the keys defined in this specification to create a matching dictionary that defines a particular driver personality. There are two tables of keys in the specification. The first table contains keys for finding devices and the second table contains keys for finding interfaces. Both tables present the keys in order of specificity: the first key in each table defines the most specific search and the last key defines the broadest search. Each key consists of the combination of elements listed in the left column of the table.

For a successful match, you must add the elements of exactly one key to your personality. If your personality contains a combination of elements not defined by any key, the matching process will not find your driver. For example, if you're attempting to match a device and you add values representing that device's vendor, product, and protocol to your matching dictionary, the matching process is unsuccessful even if a device with those same values in its device descriptor is currently represented by an IOUSBDevice nub in the I/O Registry. This is because there is no key that combines the elements of vendor number, product number, and protocol.

Devices Without I/O Kit Families

Some categories of devices do not have family support from I/O Kit. In general, there are three reasons why a particular device may not be supported by an I/O Kit family.

- Support for certain devices is provided by other frameworks. The I/O Kit is not the most appropriate place for the abstractions that represent these devices. Examples of such devices include printers, scanners, digital cameras, and other imaging devices. If you are developing a driver for this category of device, you should use the appropriate imaging SDK.
- For some devices, it is not possible to provide a set of useful, common abstractions. Such devices might include USB security dongles, data acquisition cards, and other vendor-specific devices. These devices do not share a sufficiently large number of characteristics to make creation of I/O Kit families worthwhile. For example, although security dongles all connect via USB, there is no easily defined set of abstractions common to all such devices. An I/O Kit family would not provide

substantial assistance to developers. It should not be assumed, however, that a family is required to write a new driver. In many cases, the `IOService` class provides everything a driver requires to write a “family-less” driver.

- For many devices, it is possible to define a set of useful abstractions; however, Apple has not chosen to create a family for one or more reasons. These devices may be part of a technology that is not a common Macintosh market. Or, Apple’s engineers may not have sufficient in-house expertise with certain devices to create the best family definition. In these cases, an opportunity exists for third-party developers to extend the I/O Kit model by developing families of their own. In addition, families developed under the Apple Public Source License can be sent back to Apple for possible inclusion in future releases of Mac OS X.

Imaging Devices

There is no I/O Kit family for imaging devices (cameras, printers, and scanners). Instead, support for particular imaging device characteristics is handled by user-space code (see “Controlling Devices From Outside the Kernel” (page 36) for further discussion). Developers who need to support imaging devices should employ the appropriate imaging SDK.

Digital Video

To add digital video capabilities to your software, use the QuickTime APIs.

Sequential Access Devices (Tape Drives)

There is, at present, no I/O Kit family specifically designed for sequential access devices, such as tape drives. However, third-party developers can use the SAM device interface to create plug-in components for such devices.

Telephony Devices

There is, at present, no I/O Kit family for telephony devices. Apple is evaluating plans for a Telephony family for the future.

Vendor-Specific Devices

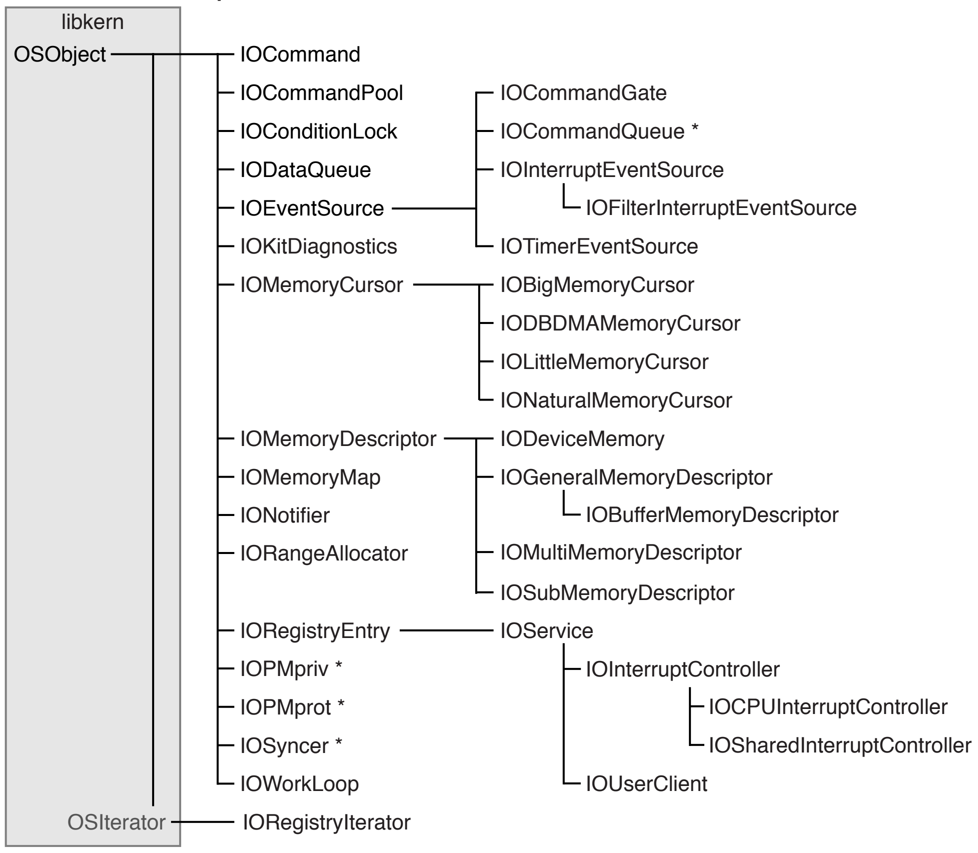
For some devices, it is not possible to provide a set of useful, common abstractions. Because families define the set of abstractions shared by all devices within the family, it is not feasible to create a family for these devices.

In most cases, however, a family is not necessary in order to write a driver for these devices. Developers should start by inheriting functionality from `IOService`, then use the `GetProperty` and `SetProperty` calls to communicate with their driver. In many cases, this should suffice. In some cases, however, such as data acquisition cards requiring high bandwidth, the developer should create their own user client (for a device-interface plug-in). Such objects can provide shared memory and procedure-call interfaces to a user-space library (see `IOUserClient.h`). You can find several good examples in IOKitExamples on the Darwin Open Source site.

Base and Helper Class Hierarchy

The following chart presents the class hierarchy of all I/O Kit classes that are not members of a specific family. See the preceding appendix, "I/O Kit Family Reference" (page 117), for the class hierarchy charts of most families.

I/O Kit Base and Helper classes



* Classes that are special-purpose or to be obsoleted; do not sublcass.

Document Revision History

This table describes the changes to *I/O Kit Fundamentals*.

Date	Notes
2006-10-03	Made minor corrections.
2006-05-23	Added caveat regarding in-function static constructors.
2006-02-07	Added information on the use of namespaces and a caveat about nested classes.
2005-12-06	Made minor corrections.
2005-11-09	Fixed minor typos.
2005-04-08	Fixed links, typos. Added note that Objective-C does not supply I/O Kit interfaces.
2004-10-05	Removed information about deprecated SCSI family; added information about new SCSI Parallel family.
2004-08-31	Added information about handling interrupts when implementing pseudo-DMA.
2004-05-27	Changed outdated links.
2004-04-22	Updated documentation references, added information on Xcode.
2004-02-13	Fixed minor errors.
2003-10-10	Added information about the use of IOBufferMemoryDescriptor objects to represent kernel-allocated buffers in user-space tasks.
2003-09-18	Updated for Mac OS X v10.3. Added information about changes in memory subsystem to support 64-bit architectures.

Bibliography

System Internals

Advanced Topics in UNIX: Processes, Files, & Systems, Ronald J. Leach, Wiley, 1996, ISBN 1-57176-159-4

The Design and Implementation of the 4.4BSD UNIX Operating System, Marshall Kirk McKusick, et al, Addison-Wesley, 1996, ISBN 0-201-54979-4

Panic!: UNIX System Crash Dump Analysis Chris Drake, Kimberly Brown Prentice Hall, 1995, ISBN 0-13-149386-8

UNIX Internals: The New Frontiers, Uresh Vahalia, Prentice-Hall, 1995, ISBN 0-13-101908-2

Websites - Online Resources

Apple Computer's developer website (http://developer.apple.com/index.html) is a general repository for developer documentation. Additionally, the following sites provide more domain-specifc information.

Apple's Public Source projects and Darwin

http://developer.apple.com/darwin/projects/darwin/

CVS (Concurrent Versions System)

http://www.opensource.apple.com/tools/cvs/cederquist/

GDB, GNUPro Toolkit 99r1 Documentation

http://www.redhat.com/docs/manuals/gnupro/

The Internet Engineering Task Force (IETF)

http://www.ietf.org/

jam

http://www.perforce.com/jam/jam.html

The USENIX Association

http://www.usenix.org/

USENIX Proceedings

http://www.usenix.org/publications/library/

Glossary

base class In C++, the class from which another class (a subclass) inherits. It can also be used to specify a class from which all classes in a hierarchy ultimately derive (also known as a root class).

BSD Berkeley Software Distribution. Formerly known as the Berkeley version of UNIX, BSD is now simply called the BSD operating system. The BSD portion of Darwin is based on 4.4BSD Lite 2 and FreeBSD, a flavor of 4.4BSD.

bundle A directory in the file system that typically stores executable code and the software resources related to that code. (A bundle can store only resources.) Applications, plug-ins, frameworks, and kernel extensions are types of bundles. Except for frameworks, bundles are file packages, presented by the Finder as a single file instead of a folder. See also "kernel extension" (page 153)

bus A transmission path on which signals can be dropped off or picked up by devices attached to it. Only devices addressed by the signals pay attention to them; the others discard the signals. Buses both exist within the CPU and connect it to physical memory and peripheral devices. Examples of I/O buses on Darwin are PCI, SCSI, USB, and FireWire.

bus master A program, usually in a separate I/O controller, that directs traffic on the computer bus or input/output paths. The bus master actually controls the bus paths on which the address and control signals flow. DMA is a simple form of bus mastering where the bus master controls I/O transfers between a device and system memory and then signals to the CPU when it has done so. See also "DMA" (page 151).

client A driver object that consumes services of some kind supplied by its provider. In a driver stack, the client in a provider/client relationship is farther away from the Platform Expert. See also "provider" (page 155).

command gate A mechanism that controls access to the lock of a work loop, thereby serializing access to the data involved in I/O requests. A command gate does not require a thread context switch to ensure single-threaded access. IOCommandGate event-source objects represent command gates in the I/O Kit.

Darwin Another name for the Mac OS X core operating system, or kernel environment. The Darwin kernel environment is equivalent to the Mac OS X kernel plus the BSD libraries and commands essential to the BSD Commands environment. Darwin is Open Source technology.

DMA (Direct Memory Access) A capability of some bus architectures that enables a bus controller to transfer data directly between a device (such as a disk drive) and a device with physically addressable memory, such as that on a computer's motherboard. The microprocessor is freed from involvement with the data transfer, thus speeding up overall computer operation. See also "bus master" (page 151).

device Computer hardware, typically excluding the CPU and system memory, which can be controlled and can send and receive data. Examples of devices include monitors, disk drives, buses, and keyboards.

device driver A component of an operating system that deals with getting data to and from a device, as well as the control of that device. A driver written with the I/O Kit is an object that implements the appropriate I/O Kit abstractions for controlling hardware.

device file In BSD, a device file is a special file located in `/dev` that represents a block or character device such as a terminal, disk drive, or printer. If a program knows the name of a device file, it can use POSIX functions to access and control the associated device. The program can obtain the device name (which is not persistent across reboots or device removal) from the I/O Kit.

device interface In the I/O Kit, a mechanism that uses a plug-in architecture to allow a program in user space to communicate with a nub in the kernel that is appropriate to the type of device the program wishes to control. Through the nub the program gains access to I/O Kit services and to the device itself. From the perspective of the kernel, the device interface appears as a driver object called a user client.

device matching In the I/O Kit, a process by which an application finds an appropriate device interface to load. The application calls a special I/O Kit function that uses a "matching dictionary" to search the I/O Registry. The function returns one or more matching driver objects that the application can then use to load an appropriate device interface. Also referred to as device discovery.

driver See "device driver" (page 151).

driver matching In the I/O Kit, a process in which a nub, after discovering a specific hardware device, searches for the driver or drivers most suited to drive that device. Matching requires that a driver have one or more personalities that specify whether it is a candidate for a particular device. Driver matching is a subtractive process involving three phases: class matching, passive matching, and active matching. See also "personality" (page 154).

driver stack In an I/O connection, the series of driver objects (drivers and nubs) in client/provider relationships with each other. A driver stack often refers to the entire collection of software between a device and its client application (or applications).

event source An I/O object that corresponds to a type of event that a device driver can be expected to handle; there are currently event sources for hardware interrupts, timer events, and I/O commands. The I/O Kit defines a class for each of these event types, respectively IOInterruptEventSource, IOTimerEventSource, and IOCommandGate.

family A collection of software abstractions that are common to all devices of a particular category. Families provide functionality and services to drivers. Examples of families include protocol families (such as SCSI, USB, and Firewire), storage families (disk drives), network families, and families that describe human interface devices (mouse and keyboard).

fault In the virtual-memory system, faults are the mechanism for initiating page-in activity. They are interrupts that occur when code tries to access data at a virtual address that is not mapped to physical memory. See also "page" (page 154); "virtual memory" (page 156).

framework A type of bundle that packages a dynamic shared library with the resources that the library requires, including header files and reference documentation.Note that the Kernel framework (which contains the I/O Kit headers) contains no dynamic shared library. All library-type linking for the Kernel framework is done using the `mach_kernel` file itself and kernel extensions. This linking is actually static (with vtable patch-ups) in implementation

information property list A property list that contains essential configuration information for bundles such as kernel extensions. A file named `Info.plist` (or a platform-specific variant of that filename) contains the information property list and is packaged inside the bundle.

interrupt An asynchronous event that suspends the currently scheduled process and temporarily diverts the flow of control through an interrupt handler routine. Interrupts can be caused by both hardware (I/O, timer, machine check) and software (supervisor, system call, or trap instruction).

interrupt handler A routine executed when an interrupt occurs. Interrupt handlers typically deal with low-level events in the hardware of a computer system, such as a character arriving at a serial port or a tick of a real-time clock.

I/O Catalog A dynamic database that maintains entries for all available drivers on a Darwin system. Driver matching searches the I/O Catalog to produce an initial list of candidate drivers.

I/O Kit A kernel-resident, object-oriented environment in Darwin that provides a model of system hardware. Each type of service or device is represented by one or more C++ classes in a family; each available service or device is represented by an instance (object) of that class.

I/O Kit framework The framework that includes IOKitLib and makes the I/O Registry, user client plug-ins, and other I/O Kit services available from user space.It lets applications and other user processes access common I/O Kit object types and services. See also "framework" (page 152).

I/O Registry A dynamic database that describes a collection of driver objects, each of which represents an I/O Kit entity. As hardware is added to or removed from the system, the I/O Registry changes to accommodate the addition or removal.

kernel The complete Mac OS X core operating-system environment, which includes Mach, BSD, the I/O Kit, drivers, file systems, and networking components. The kernel resides in its own protected memory partition. The kernel includes all code executed in the kernel task, which consists of the file `mach_kernel` (at file-system root) and all loaded kernel extensions. Also called the kernel environment.

kernel extension (KEXT) A dynamically loaded bundle that extends the functionality of the kernel. A KEXT can contain zero or one kernel modules as well as other (sub) KEXTs, each of which can contain zero or one kernel modules. The I/O Kit, file system, and networking components of Darwin can be extended by KEXTs. See also "kernel module" (page 153).

kernel module (KMOD) A binary in Mach-O format that is packaged in a kernel extension. A KMOD is the minimum unit of code that can be loaded into the kernel. See also "kernel extension" (page 153).

lock A data structure used to synchronize access to a shared resource. The most common use for a lock is in multithreaded programs where multiple threads need access to global data. Only one thread can hold the lock at a time; by convention, this thread is the only one that can modify the data during this period. See also "mutex" (page 154).

Mach A central component of the kernel that provides such basic services and abstractions as threads, tasks, ports, interprocess communication (IPC), scheduling, physical and virtual address space management, virtual memory, and timers.

map To translate a range of memory in one address space (physical or virtual) to a range in another address space. The virtual-memory manager accomplishes this by adjusting its VM tables for the kernel and user processes.

matching See "device matching" (page 152) and "driver matching" (page 152).

memory cursor An object that lays out the buffer ranges in a memory descriptor in physical memory, generating a scatter/gather list suitable for a particular device or DMA engine. The object is derived from the IOMemoryCursor class. See also "DMA" (page 151); "memory descriptor" (page 153).

memory descriptor An object that describes how a stream of data, depending on direction, should either be laid into memory or extracted from memory. It represents a segment of memory holding the data involved in an I/O transfer and is specified as one or more physical or virtual address ranges. The object is derived from the IOMemoryDescriptor class. See also "DMA" (page 151); "memory cursor" (page 153).

memory protection A system of memory management in which programs are prevented from being able to modify or corrupt the memory partition of another program. Although Mac OS X has memory protection, Mac OS 8 and 9 do not.

mutex A mutual-exclusion locking object that allows multiple threads to synchronize access to shared resources. A mutex has two states: locked and unlocked. Once a mutex has been locked by a thread, other threads attempting to lock it will block. When the locking thread unlocks (releases) the mutex, one of the blocked threads (if any) acquires (locks) it and uses the resource. The thread that locks the mutex must be the one that unlocks it. The work-loop lock (which is used by a command gate) is based on a mutex. See also "lock" (page 153); "work loop" (page 156).

notification A programmatic mechanism for alerting interested recipients (sometimes called observers) that an event has occurred.

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, disk, graphics adaptor, or any number of similar entities. A nub supports dynamic configuration by providing a bridge between two drivers (and, by extension, between two families). See also "device" (page 151); "driver" (page 152).

page (1) The smallest unit (in bytes) of information that the virtual memory system can transfer between physical memory and backing store. In Darwin, a page is currently 4 kilobytes. (2) As a verb, page refers to the transfer of pages between physical memory and backing store. Refer to `Kernel.framework/Headers/mach/machine/vm_params.h` for specifics. See also "fault" (page 152); "virtual memory" (page 156).

personality A set of properties specifying the kinds of devices a driver can support. This information is stored in an XML matching dictionary defined in the information property list (`Info.plist`) file in the driver's KEXT bundle. A single driver may present one or more personalities for matching; each personality specifies a class to instantiate. Such instances are passed a reference to the personality dictionary at initialization.

physical memory Electronic circuitry contained in random-access memory (RAM) chips, used to temporarily hold information at execution time. Addresses in a process's virtual memory are mapped to addresses in physical memory. See also "virtual memory" (page 156).

PIO (Programmed Input/Output) A way to move data between a device and system memory in which each byte is transferred under control of the host processor. See also "DMA" (page 151).

plane A subset of driver (or service) objects in the I/O Registry that have a certain type of provider/client relationship connecting them. The most general plane is the Service plane, which displays the objects in the same hierarchy in which they are attached during Registry construction. There are also the Audio, Power, Device Tree, FireWire, and USB planes.

Platform Expert A driver object for a particular motherboard that knows the type of platform the system is running on. The Platform Expert serves as the root of the I/O Registry tree.

plug-in A 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 CFPlugIn architecture, for Mac OS X applications. A kernel extension is a type of kernel plug-in.

policy maker A power-management object that decides when to change the power state of a device and instructs the power controller for the device to carry out this change. A policy maker uses factors such as device idleness and aggressiveness to make its decision. See also "power controller" (page 155).

port A heavily overloaded term which in Darwin has two particular meanings: (1) In Mach, a secure unidirectional channel for communication between tasks running on a single system; (2) In IP transport protocols, an integer identifier used to select a receiver for an incoming packet or to specify the sender of an outgoing packet.

POSIX The Portable Operating System Interface. An operating-system interface standardization effort supported by ISO/IEC, IEEE, and The Open Group.

power controller A power-management object that knows about the various power states of a device and can switch the device between them. A power controller changes the power state of a device only upon instruction from its policy maker. See also "policy maker" (page 154).

power domain A switchable source of power in a system, providing power for one or more devices that are considered members of the domain. Power domains are nested hierarchically, with the root power domain controlling power to the entire system.

preemptive multitasking A type of multitasking in which the operating system can interrupt a currently running program in order to run another program, as needed.

probe A phase of active matching in which a candidate driver communicates with a device and verifies whether it can drive it. The driver's `probe` member function is invoked to kick off this phase. The driver returns a probe score that reflects its ability to drive the device. See also "driver matching" (page 152).

process A BSD abstraction for a running program. A process' resources include a virtual address space, threads, and file descriptors. In Mac OS X, a process is based on one Mach task and one or more Mach threads.

provider A driver object that provides services of some kind to its client. In a driver stack, the provider in a provider/client relationship is closer to the Platform Expert. See also "client" (page 151).

release Decrementing the reference count of an object. When an object's reference count reaches zero, it is freed. When your code no longer needs to reference a retained object, it should release it. Some APIs automatically execute a release on the caller's behalf, particularly in cases where the object in question is being "handed off." Retains and releases must be carefully balanced; too many releases can cause panics and other unexpected failures due to accesses of freed memory. See also "retain" (page 155).

retain Incrementing the reference count of an object. An object with a positive reference count is not freed. (A newly created object has a reference count of one.) Drivers can ensure the persistence of an object beyond the present scope by retaining it. Many APIs automatically execute a retain on the caller's behalf, particularly APIs used to create or gain access to objects. Retains and releases must be carefully balanced; too many retains will result in wired memory leak. See also "release" (page 155).

scheduler That part of Mach that determines when each program (or program thread) runs, including assignment of start times. The priority of a program's thread can affect its scheduling. See also "task" (page 155); "thread" (page 155).

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. A service type is identified by a matching dictionary that describes properties of the service. A nub or driver can provide services to other I/O Kit objects.

socket In BSD-derived systems such as Darwin, a socket refers to different entities in user and kernel operations. For a user process, a socket is a file descriptor that has been allocated using `socket(2)`. For the kernel, a socket is the data structure that is allocated when the kernel's implementation of the `socket(2)` call is made.

task A Mach abstraction consisting of a virtual address space and a port name space. A task itself performs no computation; rather, it is the context in which threads run. See also "process" (page 155); "thread" (page 155).

thread In Mach, the unit of CPU utilization. A thread consists of a program counter, a set of registers, and a stack pointer. See also "task" (page 155).

timer A kernel resource that triggers an event at a specified interval. The event can occur only once or can be recurring. Timers are one of the event sources for work loops.

user client An interface provided by an I/O Kit family, that enables a user process (which can't call a kernel-resident driver or other service directly) to access hardware. In the kernel, this interface appears as a driver object called a user client; in user space, it is called a device interface and is implemented as a Core Foundation Plug-in Services (CFPlugin) object. See also "device interface" (page 152).

user space Virtual memory outside the protected partition in which the kernel resides. Applications, plug-ins, and other types of modules typically run in user space.

virtual address A memory address that is usable by software. Each task has its own range of virtual addresses, which begins at address zero. The Mach operating system makes the CPU hardware map these addresses onto physical memory only when necessary, using disk memory at other times.

virtual memory The use of a disk partition or a file on disk to provide the same facilities usually provided by RAM. The virtual-memory manager in Mac OS X provides 32-bit (minimum) protected address space for each task and facilitates efficient sharing of that address space.

wired memory A range of memory that the virtual-memory system will not page out or move. The memory involved in an I/O transfer must be wired down to prevent the physical relocation of data being accessed by hardware. In the I/O Kit memory is wired when the memory descriptor describing the memory prepares the memory for I/O (which happens when its `prepare` method is invoked).

work loop A gating mechanism that ensures single-threaded access to the data structures and hardware registers used by a driver. Specifically, it is a mutex lock associated with a thread. A work loop typically has several event sources attached to it; they use the work loop to ensure a protected, gated context for processing events. See also "event source" (page 152).

Index