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 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 an 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 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 See Should You Program in the Kernel? 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 OS X, particularly those of the kernel environment. As the driver model for 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 OS X.

Design Principles of the I/O Kit

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 OS X. The 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 OS X. Toward this end, the I/O Kit's designers settled on an object-oriented programming model that abstracted the kernel capabilities and hardware of an 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 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. 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. 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 an 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 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 an 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 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 OS X v10.3.x and OS X v10.4.x. For more information on the differences between GCC 3.3 (the default compiler in 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 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 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 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.
`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

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 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 *OS X Man Pages*.