I/O Kit Device Driver Design Guidelines

Drivers, Kernel, & Hardware: General Kernel Extensions

2009-08-14

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to I/O Kit Device Driver Design Guidelines

Important: This document was previously titled *Writing an I/O Kit Device Driver*.

To create and deploy an I/O Kit device driver requires a range of knowledge and skills, some of which seem only remotely connected to the business of writing driver code. For example, you need to package the driver for installation. You may need to localize text and images associated with the driver and display dialogs when user intervention is necessary. And unless the code you write is always perfect when first typed, you'll need to debug your driver.

This document describes various tasks that driver writers commonly perform. It is intended as a kind of "sequel" to *I/O Kit Fundamentals*. Whereas that document is primarily conceptual in nature—describing such things as the I/O Kit architecture and families, event handling, and memory and power management—this document takes a more practical approach. It is a collection of sundry topics related to writing, debugging, testing, and deploying I/O Kit device drivers.

Who Should Read This Document?

If you are developing a device driver to run in Mac OS X, you should read this document. Because this document assumes familiarity with basic I/O Kit concepts and terminology, it's a good idea to read *I/O Kit Fundamentals* first. It also helps to be familiar with object-oriented programming in general, and C++ programming specifically.

If you need to develop an application that accesses a device, you should read instead *Accessing Hardware From Applications* for more information on various ways to do that. If this sounds like a good solution for you, be aware that Objective-C does not provide interfaces for I/O Kit or BSD APIs. However, because these are C APIs, you can call them from a Cocoa application.

Organization of This Document

I/O Kit Device Driver Design Guidelines has the following chapters:

- "The libkern C++ Runtime" (page 15)

  Describes the libkern library's runtime typing system and the role of the OSMetaClass in it. It also describes techniques for object creation and destruction, dynamic casting, object introspection, and binary compatibility.

- "libkern Collection and Container Classes" (page 29)

  Describes what the libkern collection and container classes are and how to use them. It includes code samples showing how your driver can use these classes to configure itself during runtime.

- "The IOService API" (page 39)

  Provides an overview of the methods and types defined in IOService, the base class for all I/O Kit drivers. It includes descriptions of the methods for driver matching, sending and receiving notifications, client and provider messaging, power management, memory mapping, and interrupt handling. This chapter is an indispensable resource for those developing their own I/O Kit families or familyless drivers.

- "Making Hardware Accessible to Applications" (page 61)

  Starts by discussing issues related to the transfer of data between a driver and a user-space program and summarizing the various approaches for doing so. It then describes one of the approaches: Custom user clients. It gives an overview of user-client architecture and points out factors affecting the design of user clients. Finally, it describes how to implement both sides of a user client: The IOUserClient subclass in the kernel and the library in user space.

- "Kernel-User Notification" (page 103)

  Describes how you can use the Kernel–User Notification Center to present users with localized dialogs (blocking and non-blocking), launch user-space executables (including specific preference panes of System Preferences), and load sophisticated and localized user interfaces from bundles.

- "Displaying Localized Information About Drivers" (page 115)

  Summarizes the steps for internationalizing the bundles known as kernel extensions and describes how to access the localized resources in these bundles from user space.

- "Debugging Drivers" (page 121)

  A collection of tips and techniques for debugging I/O Kit device drivers. It discusses (among other things) debugging drivers during the matching and loading stages, setting up for two-machine debugging, using the kernel debugging macros, logging techniques, and debugging panics and system hangs.

- "Testing and Deploying Drivers" (page 151)

  Discusses strategies for driver testing and offers guidance on packaging and deploying device drivers.

- "Developing a Device Driver to Run on an Intel-Based Macintosh" (page 159)

  Provides tips for developing an in-kernel device driver to run in either a PowerPC-based or Intel-based Macintosh computer.

- "Document Revision History" (page 169)

  Lists changes to this document.

- "Glossary" (page 163)

  Defines key terms used in this document.

See Also

In addition to *I/O Kit Device Driver Design Guidelines*, Apple developer documentation includes several documents that cover the Mac OS X kernel, the I/O Kit in general, and driver development for specific devices. Some of these documents are listed below.

- *Kernel Programming Guide* describes at a high level the architecture and facilities of the Mac OS X core operating system, including Mach, BSD, the Virtual File System, networking, virtual memory, and kernel services. In addition, it discusses topics of interest to kernel programmers, such as performance, security, and coding conventions.

- *I/O Kit Fundamentals* describes the features, architecture, classes, and general mechanisms of the I/O Kit and includes discussions of driver matching and loading, event handling, memory management, and power management.
- *Kernel Extension Programming Topics* 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. It also includes information on other aspects of kernel extensions.
- Documentation that provides in-depth information on writing drivers for specific device families is available in Hardware & Drivers Reference Library.

In addition to these Apple publications, you can browse the BSD man pages for more information on BSD and POSIX APIs. You can view the documentation for BSD and POSIX functions and tools by typing `man`*function_name* in a Terminal window (for example, `man gdb`) or in HTML at *Mac OS X Man Pages*.

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` (user-space).

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

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

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

- The Darwin Open Source site (http://developer.apple.com/darwin/) contains information and resources for Darwin and other open-source projects maintained by Apple.
- 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.

The libkern C++ Runtime

When they designed the Mac OS X kernel, Apple engineers decided upon a restricted form of C++ because they felt the excluded features—exceptions, multiple inheritance, templates, and runtime type information (RTTI)—were either insufficient or not efficient enough for a high-performance, multithreaded kernel. But because some form of RTTI was required, Apple devised an enhanced runtime typing system. This system, which is implemented by the libkern library, provides the following features:

- Dynamic object allocation and object construction and destruction
- Object introspection and dynamic casting of objects
- Runtime object accounting (tracking the number of current instances per class)
- Safeguards for binary compatibility

The principal agent behind these features is libkern's OSMetaClass class. OSMetaClass is a peer class to OSObject—the class that all device drivers ultimately derive from—because both classes directly inherit from the same true root class, OSMetaClassBase. This chapter explores in some depth the APIs and services of the OSMetaClass and OSMetaClassBase classes and discusses how you can best take advantage of them in your code. The APIs of OSMetaClass and OSMetaClassBase are defined in `OSMetaClass.h`.

Creation of the Runtime System

The libkern library builds and updates its C++ runtime system whenever kernel extensions are loaded (or unloaded) from the kernel. Each class in the libkern or I/O Kit libraries is itself an object of type OSMetaClass. The OSMetaClass class specifies four instance variables to characterize class instances:

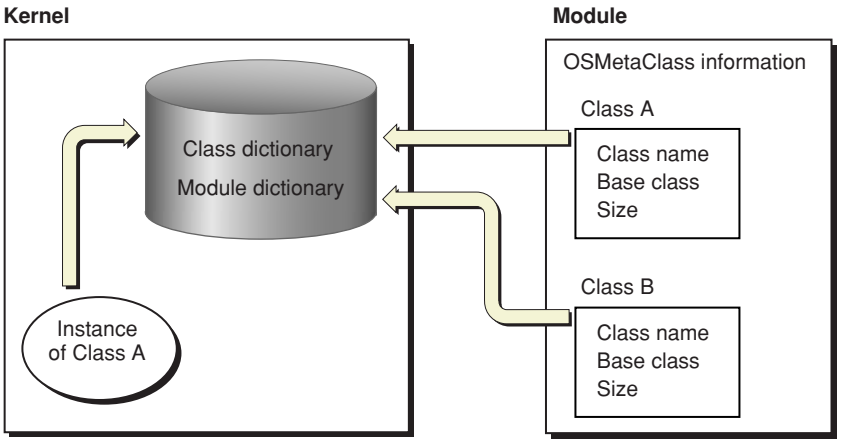
- Class name
- Base class (pointer to)
- Size of class
- Current number of instances

When the kernel loader loads a kernel extension (KEXT), it needs to register the first three bits of information with the runtime system. This system is actually a metaclass database consisting of two cross-indexed dictionaries. One dictionary—call it the class dictionary—is indexed by class name; the other dictionary, known as the module dictionary, is indexed by the name of the KEXT binary.

- The class dictionary consists of class-name keys paired with OSMetaClass objects. It is essential to the dynamic creation of libkern objects, including driver instances.
- The module dictionary consists of the KEXT-binary names paired with an array of the OSMetaClass objects in the binary. It is an essential part of the way KEXTs are safely unloaded from the kernel

Figure 1-1 (page 16) illustrates how the libkern runtime writes class information to these dictionaries when KEXTs are loaded into the kernel.

Figure 1-1 Registering metaclass information from a KEXT binary



There are three distinct phases to metaclass registration that the loader performs when loading a kernel extension:

1. **Before Loading** The loader calls the OSMetaClass member function `preModLoad`. This function grabs a lock to guarantee single-threading during the loading of multiple kernel extensions. It also generates global information about classes in KEXT binaries in preparation for registration.

2. **Loading** The loader calls each of the static constructors created by OSMetaClass macros as part of a class definition (see"Object Creation and Destruction" (page 17)). As a result, the OSMetaClass constructor for each class is invoked; the arguments for this constructor are three of the OSMetaClass data members: Class name, pointer to the base class, and class size. The loader updates the class and module dictionaries, using the KEXT-binary name and class names as keys and inserting the just-created OSMetaClass objects into those dictionaries. It also links up all base-class inheritance pointers.

3. **After Loading** After all static constructors are called, the loader calls the OSMetaClass member function `postModLoad`. This function releases the lock and returns the result code from the loading. If this code indicates an error, such as a badly formed constructor, the load attempt is aborted.

Note: Although the OSMetaClass member functions for registering metaclass information are given public scope, you should not call them in your code, particularly `preModLoad` and `postModLoad`.

Whenever kernel code creates an instance of an OSObject-derived class, the libkern runtime typing system increments the instance counter of the associated OSMetaClass object; it decrements the counter whenever an instance is freed. The runtime system uses this running tally of instances to prevent the unloading of kernel extensions having "live" objects in the system. (Of course, if code improperly retains and releases objects, this could lead to the retention of KEXTs that should be unloaded—and, consequently, memory leaks. And leaked object references lead to annoying and costly delays in the development cycle.)

When the instance count of all classes in a KEXT binary reaches zero, the kernel unloader waits a minute (to ensure that the binary won't be used again soon) before unloading the binary from the kernel. Just before unloading, the unloader calls the static destructor of each class in the binary, which removes all references to that class from the runtime system.

Object Creation and Destruction

Because exceptions are excluded from the kernel's restricted form of C++, you cannot implement "normal" C++ constructors and destructors without jeopardy. Constructors and destructors are typed to return no value (such as an error code). Normally, if they encounter a problem, they raise an exception. But because exceptions aren't supported in the kernel's C++ runtime, there is no way for you to know when an allocation or deallocation error has occurred.

This situation prompted a design feature of the libkern's C++ runtime system that uses OSMetaClass macros to specify the structure of a class—that is, the metaclass data structures and functional interfaces—for the runtime typing system. The macros also define the primary constructor and a destructor for a class. These macro-created constructors are guaranteed not to fail because they do not themselves perform any allocations. Instead, the runtime system defers the actual allocation of objects until their initialization (usually in the `init` member function). Because the `init` function is typed to return a `bool`, it makes it possible to return an error upon any failure.

Using the OSMetaClass Constructor Macros

When you create a C++ class based on OSObject, your code must call a matching pair of macros based upon the OSMetaClass class. The calls must be among the first statements in both the definition and implementation of the class. These macros are critical to your class because they enter metaclass information about it into the libkern runtime typing facility and define the static constructor and destructor for your class.

For concrete (that is, non-abstract) classes, 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 com_MyCompany_driver_MyDriver : public IOService
{
    OSDeclareDefaultStructors(com_MyCompany_driver_MyDriver);
    /* ... */
};
```

Your class implementation must include the companion "define" macro, `OSDefineMetaClassAndStructors`. This macro defines the constructor and destructor, implements the OSMetaClass allocation member function (`alloc`) for the class, and supplies the metaclass information for the runtime typing system. `OSDefineMetaClassAndStructors` takes as arguments the name of your driver's class and the name of its base class. It uses these to generate code that allows your driver class to be loaded and instantiated while the kernel is running. It typically occurs as one of the first statements of the class implementation. For example, `MyDriver.cpp` might begin like this:

```
#include "MyDriver.h"

// This convention makes it easy to invoke base class member functions.
#define super    IOService
```

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

If the class you are defining is an abstract class intended only to be inherited from, use the `OSDeclareAbstractStructors` and `OSDefineMetaClassAndAbstractStructors` macros instead. These macros do the same things as their non-abstract counterparts except that they make the primary constructor private and define the `alloc` member function to return zero.

The `OSDefineMetaClassAndStructors` and `OSDefineMetaClassAndAbstractStructors`macros are based on two other OSMetaClass macros: `OSDefineMetaClassAndStructorsWithInit` and `OSDefineMetaClassAndAbstractStructorsWithInit`, respectively. These latter two macros are deprecated and should not be used directly.

Allocating Objects Dynamically

The OSMetaClass plays an important role in the libkern C++ runtime by allocating objects based upon class type. Derived classes of OSMetaClass can do this dynamically by implementing the `alloc`function; the class type is supplied by the OSMetaClass derived class itself. As mentioned in the previous section, "Object Scope and Constructor Invocation" (page 19), the constructors created by the OSMetaClass macros implement `alloc` automatically for your class.

The container classes of libkern and the families of the I/O Kit provide various helper member functions for creating (allocating) objects, and these are what you use most of the time. But you can also directly allocate an instance of any libkern or I/O Kit class using two kinds of OSMetaClass calls:

- By calling one of the OSMetaClass `allocClassWithName`functions, supplying an identification of class type (as an OSSymbol, OSString, or C-string)
- By calling the macro `OSTypeAlloc` (defined in the OSMetaClassBase class)

Both the `allocClassWithName` and the `OSTypeAlloc` macro are similar in that they take some indication of type as the sole argument. However, there is an important difference. Because of a preprocessor artifact, the macro takes a type argument that is a compile-time symbol and not a string; thus the macro is more efficient than the dynamic allocation performed by the `allocClassWithName` member functions, but it is not truly dynamic. The allocation member functions defer binding until runtime, whereas the macro will generate a link-time error if the kernel extension doesn't properly specify the dependencies of the type argument. In addition, the macro, unlike the functions, casts the result to the appropriate type for you.

The `OSTypeAlloc` macro is intended to replace the C++ `new` operator when you are creating objects derived from OSObject. The reason behind this change is binary compatibility: The `new` operator is fragile if you are creating an object that doesn't exist in the same kernel extension as your code. By passing the size of a class as an argument to `malloc`, C++ compiles the size value into the calling binary. But if there are any dependencies on that size and the class later becomes larger, the binary might break in sundry subtle ways, such as by writing over succeeding allocation blocks. The `OSTypeAlloc` macro, on the other hand, allows the class doing the allocation to determine its own size.

Freshly allocated objects created with the allocation macro or functions have a retain count of one as their sole data member and are otherwise uninitialized. After allocation, you should immediately invoke the object's initialization member function (typically `init` or some variant of `init`). The initialization code thus has the opportunity to test whether the object is valid and to return an appropriate error code if it isn't.

Global Initializers

Global constructors—sometimes called global initializers—are a useful feature of the C++ language. For drivers, they permit the safe initialization and allocation of resources that must be available before the driver starts transferring data. To properly understand what global constructors are and why they are useful, let's first review *when* C++ objects with various kinds of scope are constructed and destroyed.

Object Scope and Constructor Invocation

C++ gives you four different ways to create an object. In each case, the object has a specific scope because the invocations of its constructor and destructor occur at specific times. Where the construction of the object occurs as a result of declaration (automatic local, static local, and global), only the default constructor is invoked.

- **Explicitly created objects** Such objects are dynamically created through the `new` operator and are destroyed through the `delete` operator. The life time of the object is the period between `new` and `delete`, because that is when the constructor and destructor are called. OSMetaClass bases its mechanism for dynamic allocation of libkern objects on the `new` operator. All objects that inherit from OSObject can only be explicitly created objects. OSMetaClassBase's `retain` and `release` calls implement a more sophisticated reference-counting mechanism—based on `new` and `delete`—for object persistence.
- **Automatic local objects** An automatic local object is created each time control pass through its declaration within a member function. Its destructor is called each time control passes back above the object's declaration or out of the enclosing block. The following example illustrates the scope of automatic local objects (`IntArray` is a C++ class):

```
void func() {
    IntArray a1;                // a1 constructed here 1 time
    int l = 2;
    for (i=0; i < 3; i++) {
        IntArray a2;            // a2 constructed here 3 times
        if (i == l) {
            IntArray a3;        // a3 constructed here 1 time
            // ...                 (when "if" is true)
        }                       // a3 destroyed at exit of "if" block
    }                           // a2 destroyed here 3 times
}                               // a1 destroyed here 1 time
```

- **Static local objects** These objects are similar to automatic local objects, but with a few important differences. Static local objects are declared within a member function with a `static` keyword. Like an automatic local object, a static object's constructor is called when control passes through the declaration, but only the first time. It won't be constructed if control never passes through its declaration. A static local object's destructor is called only when the executable exits normally (such as when `main` returns or `exit` is called or a kernel extension is unloaded).

  Important: Because of implementation constraints, static local objects do not work as intended in the Darwin kernel. Therefore your code should not rely on them.

- **Global objects** A global object is declared outside of any function. It exists throughout the runtime life of an executable. The constructor of a global object is invoked before an executable's entry point; the destructor is called after the program exits. Adding the keyword `static` to the declaration limits the scope of the object to the translation unit (typically an object file) in which it is defined.

Note: The descriptions and example above borrow liberally from *C++: The Core Language*, Gregory Satir and Doug Brown, O'Reilly & Associates, Cambridge 1995.

For I/O Kit drivers, the scope of a global object entails a guarantee that two things will happen:

- The constructor is called before the invocation of its KEXT-binary's functional entry point. For drivers, global constructors are called at load time; for other kernel extensions, global constructors are called before `MODULE_START`.
- The code running in the constructor is single-threaded (per KEXT binary).

Because of these guarantees, a global constructor—or, more descriptively, a global initializer—is an ideal programming locale for implementing locks (taking advantage of the single-threadedness) and for initializing global resources.

There are a couple of caveats to note about C++ objects with global scope. First, such objects cannot derive from OSObject; as noted earlier, libkern permits only the explicit creation of objects. Second, if your code has multiple global initializers in the same translation unit, they are invoked in the order of their definition. However, if you have multiple global initializers in different KEXT binaries, the order of their invocation between binaries is undefined. Because of this, it is good programming practice to avoid dependencies among global initializers.

An Example of a Global Initializer

As the previous section makes clear, global initializers are an ideal means for initializing global data structures and for setting up resources such as locks, typically to protect those data structures. Let's look at how a global initializer might be used in driver writing.

The I/O Kit Serial family uses a global initializer to initialize global data structures. Listing 1-1 (page 20) shows the definition of the (private) IOSerialBSDClientGlobals class. Following the definition is the declaration of a `static` variable of the class; this definition generates a single instance of the class and tells the compiler that this instance and its data are global in scope.

Listing 1-1 Definition of a class to be declared global

```
class IOSerialBSDClientGlobals {
private:

    unsigned int fMajor;
    unsigned int fLastMinor;
    IOSerialBSDClient **fClients;
    OSDictionary *fNames;

public:
    IOSerialBSDClientGlobals();
    ~IOSerialBSDClientGlobals();

    inline bool isValid();
    inline IOSerialBSDClient *getClient(dev_t dev);

    dev_t assign_dev_t();
    bool registerTTY(dev_t dev, IOSerialBSDClient *tty);
    const OSSymbol *getUniqueTTYSuffix
        (const OSSymbol *inName, const OSSymbol *suffix, dev_t dev);
```

```
    void releaseUniqueTTYSuffix(const OSSymbol *inName, const OSSymbol *suffix);
};

static IOSerialBSDClientGlobals sBSDGlobals;
```

The declaration of the `sBSDGlobals` variable kicks off the constructor for the IOSerialBSDClientGlobals class (and other global initializers) at load time. The Serial family implements this constructor as shown in Listing 1-2 (page 21).

Listing 1-2 Implementation of a global constructor

```
#define OSSYM(str) OSSymbol::withCStringNoCopy(str)
IOSerialBSDClientGlobals::IOSerialBSDClientGlobals()
{
    gIOSerialBSDServiceValue = OSSYM(kIOSerialBSDServiceValue);
    gIOSerialBSDTypeKey      = OSSYM(kIOSerialBSDTypeKey);
    gIOSerialBSDAllTypes     = OSSYM(kIOSerialBSDAllTypes);
    gIOSerialBSDModemType    = OSSYM(kIOSerialBSDModemType);
    gIOSerialBSDRS232Type    = OSSYM(kIOSerialBSDRS232Type);
    gIOTTYDeviceKey          = OSSYM(kIOTTYDeviceKey);
    gIOTTYBaseNameKey        = OSSYM(kIOTTYBaseNameKey);
    gIOTTYSuffixKey          = OSSYM(kIOTTYSuffixKey);
    gIOCalloutDeviceKey      = OSSYM(kIOCalloutDeviceKey);
    gIODialinDeviceKey       = OSSYM(kIODialinDeviceKey);
    gIOTTYWaitForIdleKey     = OSSYM(kIOTTYWaitForIdleKey);

    fMajor = (unsigned int) -1;
    fNames = OSDictionary::withCapacity(4);
    fLastMinor = 4;
    fClients = (IOSerialBSDClient **)
                IOMalloc(fLastMinor * sizeof(fClients[0]));
    if (fClients && fNames) {
        bzero(fClients, fLastMinor * sizeof(fClients[0]));
        fMajor = cdevsw_add(-1, &IOSerialBSDClient::devsw);
    }

    if (!isValid())
        IOLog("IOSerialBSDClient didn't initialize");
}
#undef OSSYM
```

This code creates OSSymbol objects for various attributes of the Serial family and allocates other global resources. One thing to note about this example code is the call to `isValid`; this member function simply verifies whether the global initialization succeeded. Because exceptions are excluded from the kernel's restricted form of C++, you need to make some validation check like this either in the constructor itself or in the `start` member function. If the global initializer did not do its work, your driver should gracefully exit.

The destructor for the IOSerialBSDClientGlobals (see Listing 1-3 (page 21)) class frees the global resources created by the constructor. A global destructor is called after the invocation of the `MODULE_STOP` member function or, in the case of drivers, at unload time.

Listing 1-3 Implementation of a global destructor

```
IOSerialBSDClientGlobals::~IOSerialBSDClientGlobals()
{
    SAFE_RELEASE(gIOSerialBSDServiceValue);
    SAFE_RELEASE(gIOSerialBSDTypeKey);
```

```
    SAFE_RELEASE(gIOSerialBSDAllTypes);
    SAFE_RELEASE(gIOSerialBSDModemType);
    SAFE_RELEASE(gIOSerialBSDRS232Type);
    SAFE_RELEASE(gIOTTYDeviceKey);
    SAFE_RELEASE(gIOTTYBaseNameKey);
    SAFE_RELEASE(gIOTTYSuffixKey);
    SAFE_RELEASE(gIOCalloutDeviceKey);
    SAFE_RELEASE(gIODialinDeviceKey);
    SAFE_RELEASE(gIOTTYWaitForIdleKey);
    SAFE_RELEASE(fNames);
    if (fMajor != (unsigned int) -1)
        cdevsw_remove(fMajor, &IOSerialBSDClient::devsw);
    if (fClients)
        IOFree(fClients, fLastMinor * sizeof(fClients[0]));
}
```

As shown in this example, the destructor should always check if initializations did occur before attempting to free resources.

Object Introspection and Dynamic Casting

Object introspection is a major benefit that the libkern runtime typing facility brings to kernel programming. By querying this facility, the functions and macros of the OSMetaClass and OSMetaClassBase classes allow you to discover a range of information about arbitrary objects and classes:

- The class of which an object is a member
- Whether an object inherits from (directly or indirectly) a specified class
- Whether one object is equal to another object
- The current number of runtime instances for a class and its derived classes
- The size, name, and base class of a particular class

Table 1-1 (page 22) describes the introspection functions and macros.

Table 1-1 OSMetaClass and OSMetaClassBase introspection macros and member functions

Member Function or Macro	Description
`OSTypeID`	A macro that, given a class name (not quoted), returns the indicated OSMetaClass instance.
`OSTypeIDInst`	A macro that, given an instance of an OSObject derived class, returns the class of that object (as an OSMetaClass instance).
`OSCheckTypeInst`	A macro that determines whether a given object inherits directly or indirectly from the same class as a reference object (that is, an object whose class type is known).

Member Function or Macro	Description
`isEqualTo`	Returns `true` if two objects are equivalent. What equivalence means depends upon the libkern or I/O Kit derived class overriding this member function. OSMetaClassBase implements this function as a shallow pointer comparison.
`metaCast`	Determines whether an object inherits from a given class; variants of the member function let you specify the class as an OSMetaClass instance, an OSSymbol object, an OSString object, or a C-string. Defined by the OSMetaClassBase class.
`checkMetaCastWithName`	Similar to `metaCast`, but implemented as a static member function of OSMetaClass.
`getInstanceCount`	OSMetaClass accessor that returns the current number of instances for a given class (specified as an instance of OSMetaClass) and all derived classes of that class.
`getSuperClass`	OSMetaClass accessor function that returns the base class of the specified class.
`getClassName`	OSMetaClass accessor function that returns the name of the specified class.
`getClassSize`	OSMetaClass accessor function that returns the size of the specified class.

OSMetaClassBase also includes a useful macro named `OSDynamicCast`. This macro does basically the same thing as the standard C++ `dynamic_cast<`*type* `*>(`*object*`)` operator: It converts the class type of an object to another, compatible type. But before the `OSDynamicCast` macro converts the type of the instance, it verifies that the cast is valid—that is, it checks if the given instance inherits from the given type. If it doesn't, it returns zero. It also returns zero if either the class type or the instance specified as a parameter is zero. No type qualifiers, such as `const`, are allowed in the parameters.

Listing 1-4 (page 23) illustrates how you might do some introspection tests and dynamic casts.

Listing 1-4 Code showing dynamic casting and introspection

```
void testIntrospection() {

    unsigned long long val = 11;
    int count;
    bool yup;
    OSString *strObj = OSString::withCString("Darth Vader");
    OSNumber *numObj = OSTypeAlloc(OSNumber);
    numObj->init(val, 3);

    yup = OSCheckTypeInst(strObj, numObj);
    IOLog("strObj %s the same as numObj\n", (yup ? "is" : "is not"));

    count = (OSTypeIDInst(numObj))->getInstanceCount();
    IOLog("There are %d instances of OSNumber\n", count);

    if (OSDynamicCast(OSString, strObj)) {
    IOLog("Could cast strObj to OSString");
    } else {
```

```
    IOLog("Couldn't cast strObj to OSString");
    }
}
```

Binary Compatibility

Something that has long plagued C++ library developers is the fragile base class problem. The libkern library and, more specifically, the OSMetaClass class offer ways to avoid the danger to backward binary compatibility posed by fragile base classes. But before you learn about those APIs, you might find a summary of the fragile base class problem helpful.

Note: Also see the section "Allocating Objects Dynamically" (page 18) for a discussion of the use of the `OSTypeAlloc` macro as a factor in binary compatibility.

The Fragile Base Class Problem

The fragile base class problem affects only non-leaf classes, and then only when those classes are defined in different KEXTs. So if you never intend your libkern or I/O Kit class to be a base class, then fragile base classes won't be an issue (and you have no need to read further). But if your class could be a base class to some other class, and you later change your class in certain ways, all external KEXTs with dependencies on your class are likely to experience problems that will cause them to break.

Not all changes to a non-leaf class make it a fragile base class. For instance, you can add non-virtual functions or even new classes to a kernel extension without peril. If you have a function that is a polymorphed version of a base class virtual function you won't have a problem either. And you can always reimplement a member function, virtual or non-virtual. But there are certain additions and modifications that are almost certain to cause binary-compatibility problems:

- New virtual member functions
- Changes to the order of virtual-function declarations
- New instance variables (data members)
- Changes to the type or size of instance variables (for example, changing a `short` to a `long` or increasing the maximum size of an array)

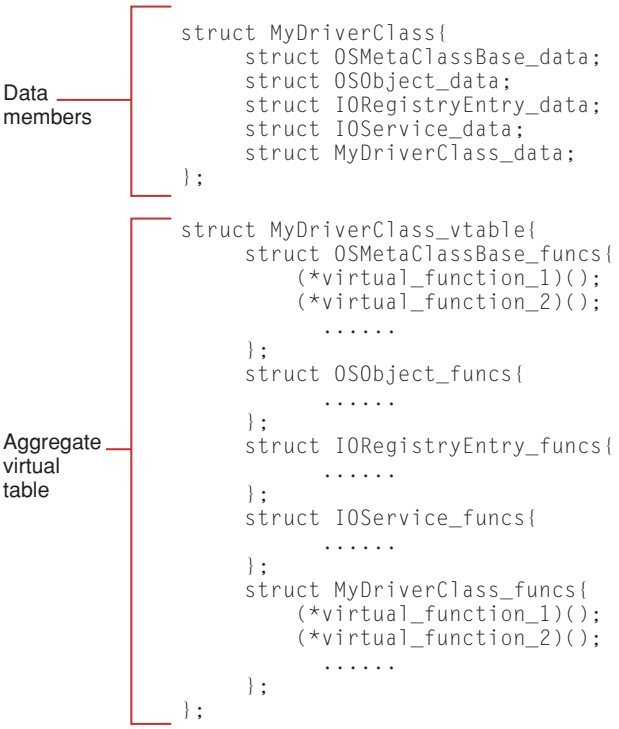
The reason these modifications to a non-leaf class introduce fragility is that KEXTs containing derived classes of that class incorporate knowledge of characteristics of the base class such as:

- The size of the object
- Offsets to protected or public data
- The size of your class's virtual table (vtable) as well as the offsets in it

Code that is dependent on these sizes and offsets will break if a size or offset subsequently changes. To look at it another way, each compiled C++ class is, internally, two structures. One structure holds the aggregate data members in an inheritance chain; the other structure contains the aggregate virtual tables of the same inheritance chain. (An instance of the class holds *all* of the aggregate data members and contains a pointer

to one instance of the aggregate virtual tables.) The C++ runtime uses the sizes and offsets created by these concatenated structures to locate data members and member functions. Figure 1-2 (page 25) illustrates in an abstract way how these structures might be laid out internally.

Figure 1-2 The aggregate data and vtable structures of a compiled class



The libkern C++ library has mechanisms that mitigate the fragile base class problem. Basically, these techniques let you create "pad slots" (that is, reserved fields) for both data members and virtual functions anticipated for future expansion. The following sections explain how to pad a class, and then how to adjust the padding whenever you add new data members or virtual functions.

Reserving Future Data Members

To prepare your non-leaf class for any future addition of data members, specify an empty `ExpansionData` structure and a `reserved` pointer to that structure. Then, when you add a field to this structure, allocate this additional data in the initialization member function for the class (typically `init`).

Note: The names `ExpansionData` and `reserved` are not defined in libkern but are conventions that Apple encourages you to follow to maintain consistency. Remember, there will be no name-space collisions because each `ExpansionData` is implicitly qualified by its class (for example, `com_acme_driver_MyDriverClass::ExpansionData`).

Enter the lines in Listing 1-5 (page 26) into your class header file in a section with protected scope (as indicated).

Listing 1-5 Initial declarations of the `ExpansionData` structure and `reserved` pointer

```
protected:
/*! @struct ExpansionData
    @discussion This structure helps to expand the capabilities of
    this class in the future.
    */
    struct ExpansionData { };

/*! @var reserved
    Reserved for future use.  (Internal use only)  */
    ExpansionData *reserved;
```

Later, when you need to add new data members to your class, add the field or fields to the `ExpansionData` structure and define a preprocessor symbol for the field as referenced by the `reserved` pointer. Listing 1-6 (page 26) shows how you might do this.

Listing 1-6 Adding a new field to the `ExpansionData` structure

```
struct ExpansionData { int eBlastIForgot; };
ExpansionData *reserved;
#define fBlastIForgot ((com_acme_driver_MyDriverClass::ExpansionData *)
            com_acme_driver_MyDriverClass::reserved)->eBlastIForgot)
```

Finally, at initialization time, allocate the newly expanded structure.

Padding the Virtual Table

The first step toward padding the virtual table of your class is to estimate how many pad slots you'll need—that is, how many virtual functions might be added to your class in the future. You don't want to add too many, for that could unnecessarily bloat the memory footprint of your code, and you don't want to add too few, for obvious reasons. A good estimation technique is the following:

1. Count the virtual functions in your class that are just polymorphed versions of a base class virtual function.
2. Subtract that value from the total number of virtual functions in your class.
3. Pick a number between the result of the subtraction and the total number of virtual functions. Your choice, of course, should be influenced by your sense of how much updating the API might undergo in the future.

When you've determined how many reserved slots you'll need for future expansion, specify the `OSMetaClassDeclareReservedUnused` macro in your class header file for each slot. This macro takes two parameters: the class name and the index of the pad slot. Listing 1-7 (page 26) shows how the IOHIDDevice class specifies the `OSMetaClassDeclareReservedUnused` macro.

Listing 1-7 The initial padding of a class virtual table (header file)

```
public:
    // ...
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  0);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  1);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  2);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  3);
```

```
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  4);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  5);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  6);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  7);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  8);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  9);
    OSMetaClassDeclareReservedUnused(IOHIDDevice, 10);
    // ...
```

In your class implementation file, type the corresponding `OSMetaClassDefineReservedUnused` macros. These macros also take the name of the class and the index of the pad slot as parameters. Listing 1-8 (page 27) shows these macros as they appear in the `IOHIDDevice.cpp` file

Listing 1-8 The initial padding of a class virtual table (implementation file)

```
// at end of function implementations
OSMetaClassDefineReservedUnused(IOHIDDevice,  0);
OSMetaClassDefineReservedUnused(IOHIDDevice,  1);
OSMetaClassDefineReservedUnused(IOHIDDevice,  2);
OSMetaClassDefineReservedUnused(IOHIDDevice,  3);
OSMetaClassDefineReservedUnused(IOHIDDevice,  4);
OSMetaClassDefineReservedUnused(IOHIDDevice,  5);
OSMetaClassDefineReservedUnused(IOHIDDevice,  6);
OSMetaClassDefineReservedUnused(IOHIDDevice,  7);
OSMetaClassDefineReservedUnused(IOHIDDevice,  8);
OSMetaClassDefineReservedUnused(IOHIDDevice,  9);
OSMetaClassDefineReservedUnused(IOHIDDevice, 10);
// ...
```

When you subsequently add a virtual function to your class, replace the `OSMetaClassDeclareReservedUnused` macro having the lowest index in your class header file with an `OSMetaClassDeclareReservedUsed` macro; be sure to use the same index number. To document the replacement, have the macro immediately precede the declaration of the function. The IOHIDDevice class does this as shown in Listing 1-9 (page 27).

Listing 1-9 Adjusting the pad slots when adding new virtual functions—header file

```
public:
// ...
    OSMetaClassDeclareReservedUsed(IOHIDDevice,  0);
    virtual IOReturn updateElementValues(IOHIDElementCookie * cookies,
                                         UInt32 cookieCount = 1);

protected:
    OSMetaClassDeclareReservedUsed(IOHIDDevice,  1);
    virtual IOReturn postElementValues(IOHIDElementCookie * cookies,
                                         UInt32 cookieCount = 1);

public:
    OSMetaClassDeclareReservedUsed(IOHIDDevice,  2);
    virtual OSString * newSerialNumberString() const;

    OSMetaClassDeclareReservedUnused(IOHIDDevice,  3);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  4);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  5);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  6);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  7);
```

```
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  8);
    OSMetaClassDeclareReservedUnused(IOHIDDevice,  9);
    OSMetaClassDeclareReservedUnused(IOHIDDevice, 10);
// ...
```

In the implementation file, replace the lowest-indexed `OSMetaClassDefineReservedUnused` macro with a `OSMetaClassDefineReservedUsed` macro for each new virtual function. Again, for clarity's sake, consider putting the macro immediately before the function implementation. See Listing 1-10 (page 28) for an example.

Listing 1-10 Adjusting the pad slots when adding new virtual functions—implementation file

```
OSMetaClassDefineReservedUsed(IOHIDDevice,  0);
IOReturn IOHIDDevice::updateElementValues(IOHIDElementCookie *cookies,
                                          UInt32 cookieCount) {
    // implementation code...
}

OSMetaClassDefineReservedUsed(IOHIDDevice,  1);
IOReturn IOHIDDevice::postElementValues(IOHIDElementCookie * cookies,
                                          UInt32 cookieCount) {
    // implementation code...
}

OSMetaClassDefineReservedUsed(IOHIDDevice,  2);
OSString * IOHIDDevice::newSerialNumberString() const
{    // implementation code ...
}
OSMetaClassDefineReservedUnused(IOHIDDevice,  3);
OSMetaClassDefineReservedUnused(IOHIDDevice,  4);
OSMetaClassDefineReservedUnused(IOHIDDevice,  5);
OSMetaClassDefineReservedUnused(IOHIDDevice,  6);
OSMetaClassDefineReservedUnused(IOHIDDevice,  7);
OSMetaClassDefineReservedUnused(IOHIDDevice,  8);
OSMetaClassDefineReservedUnused(IOHIDDevice,  9);
OSMetaClassDefineReservedUnused(IOHIDDevice, 10);
// ...
```

libkern Collection and Container Classes

Your driver's information property list contains at least one personality dictionary that specifies the type of device your driver can manage. A personality dictionary can also contain information pertinent to your driver's runtime configuration. When your driver is instantiated, it receives a copy of the I/O Kit personality for which it was loaded and it can use this information to make sure it is suitable to drive the device and, optionally, to configure itself to meet the device's needs.

The I/O Kit personality your driver receives is in the form of an OSDictionary, which is one of the libkern collection classes. The libkern C++ library defines container classes, which hold primitive values, such as numbers and strings, and collection classes, which hold groups of both container objects and other collection objects. This chapter first describes ways in which your driver can use the libkern collection and container classes, then it gives an overview of these classes and their methods. The chapter concludes with code samples that illustrate how your driver can use the libkern classes to configure itself.

For more information on the libkern library and how it supports loadable kernel modules, see "The libkern C++ Runtime" (page 15) and *I/O Kit Fundamentals*. For API reference documentation on the libkern classes, see Device Drivers Documentation. On Mac OS X, you can find the libkern library in `/System/Library/Frameworks/Kernel.framework/Headers/libkern/c++`.

The libkern Classes and Your Driver

The libkern collection and container classes offer powerful services your driver can use to configure its runtime environment. Although these services may seem perfect for use in your driver's I/O routines, they are too costly to use outside your driver's dynamic configuration phases. During your driver's `init`, `start`, and `probe` routines, however, the trade-off between cost and benefit make the libkern container and collection classes attractive.

The libkern classes offer many benefits, such as object introspection, encapsulation, and ability to serialize. In addition, the libkern container and collection classes closely correspond to the Core Foundation classes in both name and behavior. This allows the system to automatically translate between libkern and Core Foundation classes of the same type. For example, the libkern collection object OSDictionary is converted into the Core Foundation object CFDictionary when crossing the user-kernel boundary.

For a driver, one of the greatest benefits of the libkern classes lies in their ability to make a driver's XML property list available to the driver during its life span. There are three general ways your driver can use the OSDictionary representation of its I/O Kit personality.

- A driver can configure itself to drive a particular type of device by reading the contents of its property list and performing the appropriate set-up.
- A driver can modify its property list (and, optionally, that of its provider) based on information it receives from the device or other service objects currently in the I/O Registry. It can then use that information to customize its runtime environment.
- A user-space process can access a driver's property list and use it to send small amounts of data that the driver can use to customize its runtime environment.

Although this chapter's information on the libkern collection and container classes is applicable to all three situations, the code examples in "Configuring Your Driver Using the libkern Classes" (page 36) focus on the first two uses. For more information on user space–driver communication using the property list (including a code sample), see "Making Hardware Accessible to Applications" (page 61).

It is important to distinguish between manipulating the OSDictionary representation of your driver's I/O Kit personality and manipulating the information property list (or `Info.plist` file) itself. Note that all property list manipulation this chapter discusses concerns the OSDictionary representation of your driver's personality, *not* the XML personality defined in your driver's bundle settings. You cannot modify your driver's `Info.plist` file in any way during your driver's life span. When you manipulate your driver's personality during its `start` or `init` routines, you are only manipulating a kernel representation of the personality that exists in the IORegistryEntry parent class of your driver. Your driver (and other objects currently residing in the I/O Registry) can access this property list and even make changes to it, but those changes will be lost when your driver terminates and the I/O Kit removes the corresponding IORegistryEntry object from the I/O Registry.

libkern Collection and Container Class Overview

Although the libkern collection and container classes inherit from OSMetaClassBase, it's more helpful to think of them as inheriting from OSObject. OSObject defines the dynamic typing and allocation features loadable kernel modules need and its virtual methods and overridden operators define how objects are created, retained, and disposed of in the kernel. For more information on the OSObject class, see *I/O Kit Fundamentals*.

With the exception of Date (which is not used in the kernel because of overhead associated with routines needed to support date formats), the XML tags you use to specify data types in your driver's `Info.plist` file have a direct, one-to-one relationship with the libkern container and collection classes of similar names. Table 2-1 (page 30) shows this relationship.

Table 2-1 XML tags and libkern class names

XML tag	libkern class
Array	OSArray
Boolean	OSBoolean
Data	OSData
Date	No libkern equivalent
Dictionary	OSDictionary
Number	OSNumber
String	OSString

libkern Container Classes

The libkern library defines container classes that hold primitive values or raw data, similar to C scalar types. Table 2-2 (page 31) shows the libkern container classes along with the types of values they can hold.

Table 2-2 The libkern container classes

Container class name	Contents
OSBoolean	Boolean values `true` and `false`
OSData	arrays of bytes
OSNumber	numeric values
OSString	arrays of characters
OSSymbol	references to unique strings

OSData, OSNumber, and OSString objects are mutable objects, however, you can make them unchangeable by declaring them to be `const`. This is useful if you know your code should never change a particular object: Declare the object to be `const` and the compiler will catch any attempts to modify it.

OSSymbol and OSBoolean are different from the other container classes. An OSSymbol object is an immutable object representing a unique string value that usually resides in the OSSymbol Pool. Although the creation of OSSymbols is not cheap, they provide great efficiency when you need to access the same strings many times. For example, if you examine the I/O Registry, you'll see the same strings used over and over, such as "IOProviderClass" and "IOProbeScore". In fact, all these strings are OSSymbol objects. Because kernel space is limited, it is efficient to keep a pool of commonly used strings for ready access rather than to waste space storing the same strings more than once. For more information on how to create and use OSSymbol objects, see "Container Object Creation and Initialization" (page 32) and "Container Object Introspection and Access" (page 33).

OSBoolean is a variation of OSSymbol. Because there are only two values associated with an OSBoolean object, `true` and `false`, there is no need to create multiple copies of the same values.

Each container class defines several methods to manage the data it holds. Common among them are methods to initialize an instance with a particular value, return the current value, and compare the current value with a passed-in value. "Using the libkern Collection and Container Classes" (page 32) describes some of these methods in more detail.

libkern Collection Classes

The libkern collection classes manage groups of OSObject-derived objects, including both container objects and other collection objects. Inheriting from the libkern class OSCollection, the collection classes make their contents accessible through a positional or associative key and provide iteration services that allow you to peruse their contents one member at a time. All collection class objects are mutable. Table 2-3 (page 31) shows the libkern collection classes along with the types of objects they can hold.

Table 2-3 The libkern collection classes

Collection class name	Contents
OSArray	A list of references to OSObject-derived objects, each accessible by positional index
OSDictionary	A list of OSObject-derived objects, each accessible by unique associative key

Collection class name	Contents
OSOrderedSet	A sorted list of unique OSObject-derived objects, each accessible by numeric order or member function
OSSet	An unsorted list of unique OSObject-derived objects, each accessible by member function

Although you can use OSSet and OSOrderedSet in your driver (for statistics-gathering perhaps) you will probably be more interested in OSArray and OSDictionary because they are direct counterparts of the XML tags Array and Dictionary you use in your driver's property list.

The libkern collection classes have in common methods that retrieve objects, check for the presence of objects, and create instances with predefined groups of objects. "Collection Object Creation and Initialization" (page 34) and "Collection Object Introspection and Access" (page 34) describe some of these methods in more detail.

Because they inherit from OSCollection, the libkern collection classes also provide access to their contents through an iterator. OSCollectionIterator defines a consistent mechanism that iterates over any given collection. When you instantiate an OSCollectionIterator for your particular collection object, you automatically have access to each collection member without having to be concerned about how the collection object organizes its contents.

Using the libkern Collection and Container Classes

The libkern collection and container classes implement a variety of methods that create, initialize, access, and examine instances. This section presents an overview of these methods, along with information on libkern class reference-counting and thread safety.

Container Object Creation and Initialization

All container classes implement one or more static methods for creating instances. These method names follow the form `with`*Arguments* where *Arguments* describes the initialization arguments. For example, OSData defines a static creation method `withBytes` that creates an instance of OSData and initializes it with the provided buffer of data.

This example uses OSData's `withCapacity` creation method to create an empty OSData object initialized to a given size:

```
OSData *tmpData;
int size = 16; //Initial capacity of tmpData in bytes.
tmpData = OSData::withCapacity( size );
```

Some of the static creation methods specify a "no copy" variant, such as OSData's `withBytesNoCopy` and OSString's `withCStringNoCopy`. These creation methods create an instance of the container object but do not actually copy the provided initialization data into it. Instead, the method gives the container object a reference to the provided data.

To see why this is useful, suppose your driver needs to use the string `device_type`. You instantiate an OSString object to contain the string with the creation method `withCString`, as in the following

```
OSString * myString;
myString = OSString::withCString("device_type");
```

But because you define `device_type` as a literal string in your driver's executable code, bytes are already wired down for it and creating an OSString object to contain it simply wastes kernel space. Therefore, if you need to reference data that you define in your driver's executable code, such as the string `device_type`, and no one will need that data after your driver unloads, you should use the "no copy" creation methods. When you create libkern container objects with a "no copy" creation method, the resulting objects are immutable because they contain only a pointer to the data, not the data itself.

When you create an OSSymbol object, you pass a string (either an OSString object or a simple C-string) to one of OSSymbol's static creation methods. If your string already exists in the OSSymbol Pool, you receive a reference to the original string. The OSSymbol representing that string then increments its retain count to keep track of the additional reference to the string. If your string does not already exist in the symbol pool, you receive a pointer to a fresh OSSymbol object and your string is added to the pool.

If your driver defines a string that will be needed by other IORegistryEntry objects after your driver terminates, you can create an OSSymbol for it using OSSymbol's `withCStringNoCopy` creation method. If your string does not already exist in the OSSymbol Pool, the `withCStringNoCopy` method does not immediately add it. Instead, OSSymbol uses the string in your driver as its own, unique pool for as long as your driver stays loaded. During this time, if other IORegistryEntry objects create the same symbol, they receive a pointer to your string. When your driver is about to unload, OSSymbol then copies your string into the general OSSymbol Pool so it remains available to the other objects that have references to it.

The container classes OSData, OSString, and OSNumber also implement initialization methods that initialize existing instances of the container objects with the provided data. For example, OSString implements three initialization methods. The first two, `initWithCString` and `initWithString`, copy the provided C string or OSString object, respectively, into an instance of OSString. The third, `initWithCStringNoCopy`, is a "no copy" variant that initializes an instance of OSString but does not copy the provided C string into it. Similarly, OSData implements several initialization methods that initialize an OSData object with a block of data, including the variant `initWithBytesNoCopy` that initializes the OSData object but does not copy the bytes into it.

OSBoolean and OSSymbol do not implement `init` methods. OSBoolean can only be one of two predefined values and OSSymbol is never instantiated without referring to a particular string value.

Container Object Introspection and Access

All container objects implement various methods that provide access to the values they contain. For example, all container classes implement at least one `isEqualTo` method that tests the equality of the container's contents and the value of a provided object. Each `isEqualTo` method handles one of the different types of data a container object can hold. OSString, for example, implements four `isEqualTo` methods that test equality with simple C strings, other OSString objects, OSData objects, and unknown OSObject-derived objects.

OSSymbol implements three `isEqualTo` methods, two that test equality with string or OSObject-derived objects and one that tests equality with other OSSymbol objects. Because two OSSymbol objects are equal only if they reference the same string in the OSSymbol pool, this `isEqualTo` method merely performs an economical pointer comparison.

Most of the container classes implement `get` methods that return both information about their contents and the contents itself. OSString, for example, implements `getLength`, which returns the length of the string, and `getChar`, which returns the character at the provided position in the string. It also implements `getCStringNoCopy`, which returns a pointer to the internal string representation, rather than the string itself.

OSData implements several `get` methods you can use to find out the size of the object's internal data buffer and how much it will grow, and to get a pointer to the data buffer.

Collection Object Creation and Initialization

As with the container classes, the libkern collection classes each implement a number of creation methods of the form `with`*Arguments*. Each creation method uses the passed-in object described by *Arguments* to initialize a fresh object. OSArray, for example, implements three creation methods, `withArray`, `withCapacity`, and `withObjects`. As its name suggests, the `withArray` method creates a fresh OSArray object and populates it with the provided OSArray object. The `withCapacity` method creates an OSArray object that can hold the given number of references and the `withObjects` method creates an OSArray object and populates it with the members of a static array of OSObjects.

The OSDictionary `withCapacity` and `withDictionary` creation methods are similar to OSArray's but its two `withObjects` creation methods require a little more explanation. Because an OSDictionary object contains a collection of key-value pairs, you must supply both the keys and the values to populate a fresh OSDictionary object. The `withObjects` creation methods each require two static arrays, one of OSObject values and one of either OSString or OSSymbol keys. Both methods then create a fresh OSDictionary object and populate it with key-value pairs consisting of a member of the key array and the member of the value array at the same index.

The initialization methods mirror the creation methods except that they operate on existing collection objects instead of creating new ones first.

Collection Object Introspection and Access

All collection classes implement several methods that give you information about their contents and allow you to get references to particular members. In addition, the libkern library provides the OSCollectionIterator class which implements methods that allow you to iterate over any collection object and access its members. Because the collection classes inherit from OSCollection, an OSCollectionIterator object automatically knows how to iterate over all types of collection objects.

The collection classes each implement several `get` methods that give you information about the collection itself or about specific objects within the collection. OSDictionary, for example, implements six `get` methods, three of which get the value associated with a key of type OSString, OSSymbol, or `const char`. The other three return information about the dictionary itself, such as its storage capacity, the size of increment by which it grows, and the current number of objects it contains.

The collection classes implement methods to both set an object into and remove an object from a collection. Each collection class implements at least one `set` method that inserts a passed-in object into a collection object. Additionally, each collection class implements a `setCapacityIncrement` method that sets the increment size by which the collection will grow. Each collection class also implements at least one `removeObject` method that removes the specified object and automatically releases it. In the case of OSArray, the contents shift to fill the vacated spot.

Reference Counting for Collection and Container Classes

OSObject provides the retain-release mechanism the libkern container and collection objects use. A common source of problems in driver development is unmatched reference counting.

When you first create a libkern object, its reference count is equal to one. Therefore, if you created an object, you should also release it when you no longer need it. You do this by calling the `release`method and then setting the object's pointer to `NULL` so you can't accidentally refer to the released object again. If you have a pointer to an object that you did not create, you should retain that object only if you need to rely on its presence and release it when you no longer need it.

Some confusion arises with the use of the `get` and`set` methods common to all libkern collection classes. When you use a `get` method to get a member object from a collection, that member object's reference count is *not* incremented. Unless you also retain the object, you should not release the object you receive from a `get` method. Although this isn't a general feature of dictionaries, it is a feature of IORegistryEntry objects.

The collection classes also implement a range of `set` methods that allow you to place an object into a collection. The `set` methods do increment the new member object's reference count. If, for example, you create an OSString object and then use `setObject` to set it into an OSDictionary object, the OSString object will have a reference count of 2, one from its creation and one from OSDictionary's `setObject` method.

Table 2-4 (page 35) summarizes the retain behavior of the collection classes's methods.

Table 2-4 libkern methods and reference-counting behavior

Method	Retain behavior
get	Never retains
set	Always retains
`remove`	Always releases

Thread Safety and the Container and Collection Classes

None of the libkern container and collection class methods are thread-safe. If, for example, you use a `get` method to get an object from a dictionary and then you call `retain` on it, there is no guarantee that the object is still valid by the time you perform the `retain`. This is because the object you received is only valid as long as the collection containing it exists. If you do not also hold a reference to the collection, you cannot be sure that it won't be released before you get a chance to retain the object in it.

In most cases, however, you will be modifying your own dictionaries and you can use your driver's locking mechanism to protect them, if necessary. If you hold a reference to a collection whose member objects you are modifying, you can be reasonably sure that those objects will continue to exist as long as you don't release the collection. Because the I/O Kit will not free any of your objects until your driver terminates (implements its `free` method), you are responsible for maintaining your collection objects and not releasing them prematurely.

libkern Objects and XML

All container and collection objects, with the exception of OSSymbol and OSOrderedSet, implement a `serialize` method. You use a serialization method when you want to represent an arbitrarily complex data structure in a location-independent way. Each time you call `serialize` on a libkern object, an OSSerialize object is created that contains both the XML (in an array of bytes) and the state of the serialization.

A good example of serialization is the output of the `ioreg` tool. The I/O Registry is a collection of interconnected dictionaries. When you type `ioreg` on the command line, the entire collection is serialized for output on your screen.

Configuring Your Driver Using the libkern Classes

As described in "The libkern Classes and Your Driver" (page 29), the best uses for the libkern classes are in your driver's dynamic configuration phase, during routines such as `init`, `probe`, and `start`. This section presents code samples that illustrate how you can use the libkern container and collection classes to configure your driver's runtime environment.

Configuring a Subclass of IOAudioDevice

The PhantomAudioDevice is an example subclass of IOAudioDevice (the PhantomAudioDriver project is available in `/Developer/Examples/Kernel/IOKit/Audio/PhantomAudioDriver` and on the CVSWeb at http://developer.apple.com/darwin/tools/cvs). In its `createAudioEngines` method, PhantomAudioDevice uses libkern objects and methods to access its property list and create a new audio engine for each audio engine array in its personality. The `createAudioEngines` method is shown in Listing 2-1 (page 36).

Listing 2-1 Using libkern objects and methods in audio-engine creation

```
bool PhantomAudioDevice::createAudioEngine()
{
    bool result = false;
    OSArray *audioEngineArray;

    audioEngineArray = OSDynamicCast(OSArray,
                            getProperty(AUDIO_ENGINES_KEY));

    if (audioEngineArray) {
        OSCollectionIterator *audioEngineIterator;

        audioEngineIterator =
                OSCollectionIterator::withCollection(audioEngineArray);
        if (audioEngineIterator) {
            OSDictionary *audioEngineDict;

            while (audioEngineDict = (OSDictionary*)
                                audioEngineIterator->getNextObject()) {
                if (OSDynamicCast(OSDictionary, audioEngineDict) != NULL) {
                    PhantomAudioEngine *audioEngine;

                    audioEngine = new PhantomAudioEngine;
```

```
                    if (audioEngine) {
                        if (audioEngine->init(audioEngineDict)) {
                            activateAudioEngine(audioEngine);
                        }
                        audioEngine->release();
                    }
                }
            }
            audioEngineIterator->release();
        }
    } else {
        IOLog("PhantomAudioDevice[%p]::createAudioEngine() - Error:
                        no AudioEngine array in personality.\n", this);
        goto Done;
    }
    result = true;
Done:
    return result;
}
```

Configuring a Subclass of IOUSBMassStorageClass

The IOUSBMassStorageClass driver matches on mass storage class–compliant interfaces of USB devices that declare their device class to be composite. Sometimes, however, USB devices that have mass storage class–compliant interfaces declare their device class to be vendor-specific instead of composite. The IOUSBMassStorageClass driver can still drive these interfaces, but because the device's reported class type is vendor-specific, the IOUSBMassStorageClass driver won't match on them.

The solution is to provide a KEXT that consists of only an `Info.plist` file containing a personality for the device. This personality acts as a sort of bridge between the interface and the IOUSBMassStorageClass driver: It matches on the interface and causes the I/O Kit to instantiate the IOUSBMassStorageClass driver. The IOUSBMassStorageClass driver then examines the personality for a special dictionary named "USB Mass Storage Characteristics" that contains interface subclass and protocol values. If the dictionary is present, the IOUSBMassStorageClass driver uses these values instead of the corresponding values reported by the device.

Listing 2-2 (page 37) shows the portion of the IOUSBMassStorageClass `start` method that locates the USB Mass Storage Characteristics dictionary and gets its contents.

Listing 2-2 Partial listing of IOUSBMassStorageClass start method

```
bool IOUSBMassStorageClass::start( IOService * provider )
{
    // Code calling super::start, opening the interface,
    // and initializing some member variables not shown here.

    // Make sure provider is an IOUSBInterface object.
    SetInterfaceReference(OSDynamicCast(IOUSBInterface, provider));
    if (GetInterfaceReference() == NULL)
        return false;

    // Check if the personality for this device specifies a preferred
    // protocol.
    if (getProperty(kIOUSBMassStorageCharacteristics) == NULL)
    {
```

```
        // This device does not specify a preferred protocol, use the
        // protocol defined in the descriptor.
        // fPreferredProtocol and fPreferredSubclass are private member
        // variables of the IOUSBMassStorageClass class.
        fPreferredProtocol =
             GetInterfaceReference()->GetInterfaceProtocol();
        fPreferredSubclass =
             GetInterfaceReference()->GetInterfaceSubClass();
    } else {
        OSDictionary * characterDict;
        characterDict = OSDynamicCast(OSDictionary,
             getProperty(kIOUSBMassStorageCharacteristics));
        // Check if the personality for this device specifies a preferred
             protocol
        if (characterDict->getObject(kIOUSBMassStoragePreferredProtocol)
             == NULL)
        {
            // This device does not specify a preferred protocol, use the
            // protocol defined in the descriptor.
            fPreferredProtocol =
                 GetInterfaceReference()->GetInterfaceProtocol();
        } else {
            OSNumber * preferredProtocol;
            preferredProtocol = OSDynamicCast(OSNumber, characterDict->
                getObject(kIOUSBMassStoragePreferredProtocol);
            // This device has a preferred protocol, use that.
            fPreferredProtocol = preferredProtocol->unsigned32BitValue();
        }
        // Check if the personality for this device specifies a preferred
        // subclass.
        if (characterDict->getObject(kIOUSBMassStoragePreferredSubclass)
             == NULL)
        {
            // This device does not specify a preferred subclass, use the
            // subclass defined in the descriptor.
            fPreferredSubclass =
                 GetInterfaceReference()->GetInterfaceSubClass();
        } else {
            OSNumber * preferredSubclass;
            preferredSubclass = OSDynamicCast(OSNumber, characterDict->
                getObject(kIOUSBMassStoragePreferredSubclass));
            // This device has a preferred subclass, use that.
            fPreferredSubclass = preferredSubclass->unsigned32BitValue();
        }
    }
```

The IOService API

Even though the IOService class inherits directly from IORegistryEntry and, by extension, from OSObject, you can think of IOService as the root class of nearly every object in the I/O Registry and, at least indirectly, of every driver. The methods of the IOService API are numerous and wide-ranging, providing services for most aspects of device management, from driver matching and loading to device interrupts and power management.

The majority of these methods are for internal (IOService) and I/O Kit family use. Many IOService methods are helper methods that IOService uses to implement other methods. Many more are meant for the I/O Kit families to implement. As a driver developer for a family-supported device, you will implement or call only a small fraction of the methods in IOService. If you are developing a driver for a familyless device, such as a PCI device, you may need to implement some of the IOService methods that families typically implement.

This chapter explores the public IOService methods that are available for external use in each of the following categories:

- Driver life cycle, including matching
- Notifications and messaging
- Accessing objects
- Power management
- Memory mapping
- Interrupt handling

Driver Life-Cycle and Matching Functionality

In the dynamic I/O Kit environment, a driver can be loaded and unloaded, or activated and deactivated, at any time. The constant in this flurry of activity is the set of IOService and IORegistryEntry methods that define every driver's life cycle.

With the exception of `init` and `free`, which OSObject defines, the remaining driver life-cycle methods are IOService methods. Because these methods are well-documented elsewhere (see *I/O Kit Fundamentals*), this chapter does not cover them again. Instead, the following sections present the less well-known IOService methods related to the driver life cycle.

Driver Matching

IOService includes a number of methods used in the driver-matching process, but unless you are developing a family or writing a familyless driver, you do not need to implement any of them. It is important, however, to understand the driver-matching process and how the I/O Kit uses IOService's matching methods so you can successfully define your driver's personality dictionary.

Table 3-1 (page 40) shows the IOService matching methods and briefly describes how the I/O Kit uses them.

Table 3-1 IOService matching methods

IOService method	Usage
`compareProperties`	Helper function used in implementing `matchPropertyTable`
`compareProperty`	Helper function used in implementing `matchPropertyTable`
`getMatchingServices`	In-kernel counterpart of the user-space function `IOServiceGetMatching-Services`
`matchPropertyTable`	Optionally implemented by families (or familyless drivers) to examine family-specific match properties
`nameMatching`	In-kernel counterpart of the user-space function `IOServiceNameMatching`
`resourceMatching`	In-kernel matching function to create a dictionary to search for `IOResources` objects
`serviceMatching`	In-kernel counterpart of the user-space function `IOServiceMatching`

For more information on the user-space functions `IOServiceGetMatchingServices`, `IOServiceNameMatching`, and `IOServiceMatching`, see *Accessing Hardware From Applications* or the HeaderDoc documentation for `IOKitLib.h` in `/Developer/ADC Reference Library/documentation/Darwin/Reference/IOKit`.

Understanding how the I/O Kit uses your driver's personality in the driver-matching process is an important prerequisite to crafting successful personality dictionaries. As described in *I/O Kit Fundamentals*, the I/O Kit uses the driver's required `IOProviderClass` key during the class-matching step to eliminate all drivers with the wrong provider type. Usually, this step eliminates the majority of drivers, leaving only those that attach to the correct nub type.

In the passive-matching phase, the I/O Kit examines the remaining keys in a driver's personality dictionary and compares them to the provider–class specific information in the device nub. It is during this phase that the family can implement the `matchPropertyTable` method to more closely inspect the match candidate.

When a family implements `matchPropertyTable`, it can interpret the match candidate's property values in any way it chooses, without interference from the I/O Kit. Additionally, if a driver includes a family-defined property in its `Info.plist` file, the I/O Kit ignores it if the family does not implement the `matchPropertyTable` method. As a driver writer, you should be familiar with the properties your driver's family uses. For example, the IOSCSIPeripheralDeviceNub class (in the IOSCSIArchitectureModel family) implements `matchPropertyTable` to rank a match candidate according to how many family-specific properties it has. Listing 3-1 (page 40) shows a fragment of IOSCSIPeripheralDeviceNub's `matchPropertyTable` implementation.

Listing 3-1 A `matchPropertyTable` implementation

```
bool IOSCSIPeripheralDeviceNub::matchPropertyTable ( OSDictionary * table,
                                    SInt32 * score )
{
    bool returnValue = true;
    bool isMatch = false;
```

```
    SInt32 propertyScore = * score;
    /* Adjust a driver's initial score to avoid "promoting" it too far. */
    /* ... */
    /* Use IOSCSIPeripheralDeviceNub's sCompareProperty method to compare */
    /* the driver's properties with family-specific properties. */
    if ( sCompareProperty ( this,table, kIOPropertySCSIPeripheralDeviceType,
                            &isMatch ) )
    {
        if ( isMatch ) {
            *score = kDefaultProbeRanking;
        }
        else {
            *score = kPeripheralDeviceTypeNoMatch;
            returnValue = false;
        }
        if ( sCompareProperty ( this, table,
                                kIOPropertySCSIVendorIdentification,
                                &isMatch ) ) {
            if ( isMatch ) {
                *score = kFirstOrderRanking;
                /* Continue to test for additional properties, */
                /* promoting the driver to the next rank with each */
                /* property found. */
            }
        }
    } else {
        /* Take care of SCSITaskUserClient "driver" here. */
    }
    if ( *score != 0 )
        *score += propertyScore;
    return returnValue;
}
```

Passive-Matching Keys

In addition to the family-specific keys the family may require, there are several passive-matching keys the I/O Kit defines:

- `IOProviderClass`
- `IOPropertyMatch`
- `IONameMatch`
- `IOResourceMatch`
- `IOParentMatch`
- `IOPathMatch`

`IOProviderClass` is required for all driver personalities because it declares the name of the nub class the driver attaches to. The provider class name also determines the remaining match keys.

The `IOPropertyMatch` key represents a specific list of properties that must match exactly in order for the I/O Kit to load the driver. For example, the IOBlockStorageDriver defines the following personality:

```
<key>IOProviderClass</key>
```

```
<string>IOBlockStorageDevice</string>
<key>IOClass</key>
<string>IOBlockStorageDriver</string>
<key>IOPropertyMatch</key>
<dict>
    <key>device-type</key>
    <string>Generic</string>
</dict>
```

After the I/O Kit determines that the IOBlockStorageDriver is a match candidate for a nub of class IOBlockStorageDevice, it examines the value of the `IOPropertyMatch` key. If the nub has a `device-type` key with the value `Generic`, the I/O Kit loads this personality of the IOBlockStorageDriver.

The utility of the `IOPropertyMatch` key lies in the fact that the I/O Kit uses its value in the matching process regardless of whether the family implements the `matchPropertyTable` method. Unlike the `matchPropertyTable` method, however, the `IOPropertyMatch` key does not allow the family to interpret its value—if the `IOPropertyMatch` value does not match the nub's corresponding property exactly, the I/O Kit removes the driver from the pool of match candidates.

The `IONameMatch` key matches on the `compatible`, `name`, or `device-type` properties of a provider. The value of the `IONameMatch` key can be an array of strings representing a list of all such properties your driver can match on. After the I/O Kit has matched and loaded your driver, it places the `IONameMatched` property and the value of the actual property value your driver matched on in your driver's I/O Registry property table.

Listing 3-2 (page 42) shows part of one of the personalities of the GossamerPE (a Platform Expert for the Blue and White G3 computer):

Listing 3-2 A personality of the GossamerPE

```
<key>GossamerPE</key>
<dict>
    <key>IOClass</key>
    <string>GossamerPE</string>
    <key>IONameMatch</key>
    <array>
        <string>AAPL,Gossamer</string>
        <string>AAPL,PowerMac G3</string>
        <string>AAPL,PowerBook1998</string>
        <string>iMac,1</string>
        <string>PowerMac1,1</string>
        <string>PowerMac1,2</string>
        <string>PowerBook1,1</string>
    </array>
    <key>IOProviderClass</key>
    <string>IOPlatformExpertDevice</string>
</dict>
```

The `IONameMatch` key contains an array of several possible names. Note that there is no `IONameMatched` key in this personality. Listing 3-3 (page 42) shows part of the I/O Registry entry for the GossamerPE after the I/O Kit matched and loaded it for a particular device:

Listing 3-3 Partial I/O Registry entry for GossamerPE

```
GossamerPE  <class GossamerPE>
{
    "IOClass" = "GossamerPE"
    "IOProviderClass" = "IOPlatformExpertDevice"
```

```
    "IONameMatched" = "PowerMac1,1"
}
```

The `IOResourceMatch` key declares a dependency or connection between your driver and a specific resource, such as the BSD kernel or a particular resource on a device, like an audio-video jack. If you add the key-value pair

```
<key>IOResourceMatch</key>
<string>IOBSD</string>
```

to your driver's personality, for example, your driver will not load until the resource, in this case the BSD kernel, is available. In this way, you can effectively stall the loading of your driver until its required resource is available. You can examine the available resources in IOResources on a running Mac OS X system using the I/O Registry Explorer application (available in `/Developer/Applications`). In the IOService plane (I/O Registry Explorer's default plane), click Root, click the platform expert device for your machine (such as PowerMac3,3), and then click IOResources.

The remaining passive-matching keys, `IOParentMatch` and `IOPathMatch`, are useful for user-space driver-client matching and are very rarely used in in-kernel driver personalities. These keys depend on information about the location of an object or service in the I/O Registry rather than on the device-specific or service-specific information a nub publishes. An application developer can examine the I/O Registry to determine the location of a specific object and create a matching dictionary using that information. This is much more difficult for an in-kernel driver developer, however, because accurate I/O Registry location of objects may not be available at development time.

Driver State

IOService provides some methods that give you information about an IOService object's state which is described in the IOService private instance variable `state`. You can view the states of objects currently attached in the I/O Registry by typing `ioreg -s` on the command line.

When a driver is in the midst of registration, matching, or termination, its busy state is set to one. When these activities conclude, the busy state is reduced to zero. Any change in an IOService object's busy state causes an identical change in its provider's busy state, so that a driver or other IOService object is considered busy when any of its clients is busy.

A driver may need to wait for a change in another IOService object's state. The `waitQuiet`method allows a driver to block until the specified IOService object's busy state is zero.

A driver can call the `adjustBusy` method to mark itself busy if, for example, it wishes to asynchronously probe a device after exiting from its `start` method.

Two IOService methods return information about a driver's stage in its life cycle. The `isOpen` method determines if a specified client has an IOService object open. If you pass zero instead of a pointer to a particular client object, `isOpen` returns the open state for all clients of an object.

The second method, `isInactive`, indicates that an IOService object has been terminated. An inactive object does not support matching, attachments, or notifications.

Resources

The I/O Kit's resource service uses the matching and notification methods usually used for driver objects to make resources available system-wide through IOResources, an instance of IOService attached to the root of the I/O Registry. A resource might be an audio-output jack on a device or a service, such as the BSD kernel. The I/O Kit itself appears as a resource.

A driver, or other IOService object, can publish a resource in IOResources using the IOService method `publishResource`. The publication of a resource triggers any notifications set on its presence, and the objects holding those notifications can then access the resource.

The IOKernelDebugger (in the IONetworking family) publishes its presence as a resource in this way:

```
publishResource ( "kdp" );
```

A driver can request a notification about the resource, make the resource a matching condition (as described in"Passive-Matching Keys" (page 41)), or, as in the AppleUSBCDCDriver, call the `waitForService` method to wait until a particular resource appears:

```
waitForService ( resourceMatching ( "kdp" ) );
```

The `waitForService` method allows an IOService object to block until the object matching the specified matching dictionary is registered. Optionally, a driver can also specify a maximum time to wait.

User Clients

IOService provides the `newUserClient` method for families that support user clients. A family that provides user clients implements the `newUserClient` method to create a connection between a user-space client application and an in-kernel user-client object.

The default implementation of `newUserClient` looks up the `IOUserClientClass` key in the given IOService object's properties. If the key is found, IOService creates an instance of the class given in the value. Then, it calls the methods `initWithTask`, `attach`, and `start` on the newly instantiated user-client class object. If, on the other hand, there is no `IOUserClientClass` key (or its value is invalid) or any of the initialization methods fail, IOService returns `kIOReturnUnsupported`.

For more information on implementing a user-client object, see "Making Hardware Accessible to Applications" (page 61).

Probing

In addition to the `probe` method your driver can choose to implement to examine a device during the matching process (described in detail in *I/O Kit Fundamentals*), IOService provides the `requestProbe` method for discovering newly added or removed devices. The `requestProbe` method gives the families that do not automatically detect device addition and removal a way to rescan the bus and publish new devices and detach removed devices.

A family, or perhaps a bus-controller driver, implements `requestProbe` and passes it a set of options contained in an object of type IOOptionBits, which is not interpreted by IOService.

Notifications and Driver Messaging

A driver frequently communicates with other entities in the I/O Registry to perform its functions. During the course of its life, a driver must be prepared to receive status and other types of messages from its provider, register for and receive notifications of various events, and send messages to its clients. IOService provides a small number of methods to handle these tasks.

Notification Methods

The `addNotification` method expects a pointer to a notification handler that IOService calls when the specified IOService object attains the specified state. For example, a driver might need to know when a particular IOService object is published. Listing 3-4 (page 45) shows how an instance of IOBSDConsole requests notification of the appearance of an IOHIKeyboard object:

Listing 3-4 Using the `addNotification` method

```
OSObject * notify;
notify = addNotification( gIOPublishNotification,
    serviceMatching( "IOHIKeyboard" ),
    ( IOServiceNotificationHandler )
        &IOBSDConsole::publishNotificationHandler,
    this, 0 );
assert( notify );
```

The first parameter `addNotification` expects is of class OSSymbol; it defines the type of event or status change. IOService delivers the notification types shown in Table 3-2 (page 45).

Table 3-2 Notification types and events

Notification type	Event type
`gIOPublishNotification`	An IOService object is newly registered.
`gIOFirstPublishNotification`	Similar to `gIOPublishNotification`, but delivered only once for each IOService instance, even if the object is reregistered when its state changes.
`gIOMatchedNotification`	An IOService object is matched with all client objects, which have all been started.
`gIOFirstMatchNotification`	Similar to `gIOMatchedNotification`, but delivered only once for each IOService instance, even if the object is reregistered when its state changes.
`gIOTerminatedNotification`	An IOService object is terminated during its `finalize` stage.

The second parameter is a dictionary describing the IOService object to match on. In Listing 3-4 (page 45), the IOBSDConsole object uses the IOService method `serviceMatching` (introduced in "Driver Matching" (page 39)) to create a dictionary with the class name IOHIKeyboard. When a matching object is published in the I/O Registry, IOService calls the notification-handling method passed to `addNotification` in the third parameter.

IOService calls the notification handler with a reference to each of the matching IOService objects that attain the specified state, beginning with the objects already in that state. The notification handler code can then use that reference in any way it chooses. Often, as in the case of the IOBSDConsole object, the notification handler examines one of the properties of the matching IOService object to update its own object's properties. Listing 3-5 (page 46) shows a fragment of the IOBSDConsole's notification-handler code.

Listing 3-5 Implementing a notification-handling method

```
bool IOBSDConsole::publishNotificationHandler( IOBSDConsole * self,
    void * ref, IOService * newService ) {
    IOHIKeyboard * keyboard = 0;
    IOService * audio = 0;

    if ( ref ) {
        audio = OSDynamicCast( IOService,
            newService->metaCast( "IOAudioStream" ) );
        if ( audio != 0 ) {
            OSNumber * out;
            out = OSDynamicCast( OSNumber, newService->getProperty( "Out" );
            if ( out ) {
                if ( out->unsigned8BitValue == 1 ) {
                    self->fAudioOut = newService;
                }
            }
        }
    }
    else { /* Handle other case here */ }
}
```

The `installNotification` method is very similar to the `addNotification` method, except that you pass it an object of class OSIterator into which the method places an iterator over the set of matching IOService objects currently in the specified state.

Messaging Methods

The primary lines of communication between a driver and its provider and clients are the messaging methods. In order to receive messages from its provider, your driver must implement the `message` method. To send messages to its clients, your driver must implement the `messageClient` or `messageClients` method.

Frequently, your driver will receive messages defined by the I/O Kit in `IOMessage.h` (available in `/System/Library/Frameworks/Kernel.framework/Headers/IOKit`), but your driver's family may also define its own messages. The IOUSBDevice class, for example, implements the `message` method to handle a number of USB family-specific messages, in addition to messages defined in `IOMessage.h`. Listing 3-6 (page 46) shows a fragment of IOUSBDevice's implementation of the `message` method.

Listing 3-6 Implementing the `message` method

```
IOReturn IOUSBDevice::message( UInt32 type, IOService * provider, void
     *argument ) {

    /* local variable declarations */

    switch ( type ) {
        case kIOUSBMessagePortHasBeenReset:
```

```
            /* handle this case */
        case kIOUSBMessageHubIsDeviceConnected:
            /* handle this case */
        case kIOUSBMessagePortHasBeenResumed:
            /* handle this case */
        /* handle messages defined in IOMessage.h */
        /* ... */
    }
}
```

A provider uses the `messageClient` and `messageClients` methods to send messages to its clients. Although you can override the `messageClient` method, you rarely need to do so unless, for example, you're writing a framework. IOService implements the `messageClients` method by applying the `messageClient` method to each client in turn. Referring again to the IOUSBDevice implementation of `message` (shown in Listing 3-6 (page 46)), an instance of IOUSBDevice calls `messageClients` on its clients to forward the messages it received from its provider, as in the following code fragment:

```
/* Previous cases of the switch statement handled here. */
case kIOUSBMessagePortHasBeenResumed:
    // Forward the message to our clients.
    messageClients( kIOUSBMessagePortHasBeenResumed, this, _portNumber );
    break;
/* Following cases of the switch statement handled here. */
```

Access Methods

IOService provides a number of methods you can use to access your driver's provider, clients, and various other objects in the I/O Registry. A few of these methods are for internal use only, but most are available for your driver to call when it needs to access other objects currently in the I/O Registry.

Getting Work Loops

Perhaps the most widely used IOService access method is `getWorkLoop`. Any time you need to ensure single-threaded access to data structures or handle asynchronous events, such as timer events or I/O commands driver clients issue to their providers, you should use a work loop. For extensive information on the architecture and use of work loops, see *I/O Kit Fundamentals*.

Many drivers can successfully share their provider's work loop to protect sensitive data and handle events. A call to `getWorkLoop` returns your provider's work loop or, if you provider does not have one, the work loop of the object below your provider in the driver stack. At the root of the I/O Registry, the IOPlatformExpertDevice object holds a system-wide work loop your driver can share, even if your providers do not have work loops.

The ApplePCCardSample driver gets the PCCard family work loop with the following lines of code:

```
IOWorkLoop * workLoop;
workLoop = getWorkLoop();
if ( !workLoop ) {
    /* Handle error. */
}
```

Note: You can find the code for the ApplePCCardSample driver in `/Developer/Examples/Kernel/IOKit/pccard`.

Although most drivers do not need to create their own work loops, a driver for a PCI controller or PCI device or any driver that interacts directly with an interrupt controller should create its own, dedicated work loop. For an example of how to do this, see *I/O Kit Fundamentals*.

Getting Clients and Providers

The methods for accessing providers and clients fall into two categories: Those that return the provider or client object itself and those that return an iterator over a set of providers or clients. The methods in the first category, `getClient` and `getProvider`, return an IOService object's primary client and primary provider, respectively. IOService defines the primary client as the first client to attach to the IOService object and the primary provider as the provider to which the IOService object first attached. Most often, your driver will have only one provider and one client, so these methods provide a convenient way to access them without having to further specify the provider or client you want.

The IOService objects returned by the `getClient` and `getProvider` methods should not be released by the caller. The client object is retained as long as it is attached and the provider object is retained as long as the client is attached to it. It is unlikely you will need to override these methods, but one possibility is implementing them to narrow down the type of IOService object returned. For example, as a convenience to subclass developers, the IOCDMedia object overrides its superclass's `getProvider` method to return IOCDBlockStorageDriver rather than the more generic IOService:

```
IOCDBlockStorageDriver * IOCDMedia::getProvider() const
{
    return (IOCDBlockStorageDriver *) IOService::getProvider();
}
```

The methods `getClientIterator`, `getOpenClientIterator`, `getProviderIterator`, and `getOpenProviderIterator` comprise the set of IOService client-provider access methods that return iterators. When you use `getClientIterator` or `getProviderIterator` to access your driver's clients or providers, each object the iterator returns is retained as long as the iterator is valid, although it is possible that the object may detach itself from the I/O Registry during the iteration. In addition, you must release the iterator when you are finished with the iteration. Listing 3-7 (page 48) shows how the IOFireWireAVCUnit object uses `getClientIterator` to check for existing subunits before creating a new one.

Listing 3-7 `getClientIterator` example

```
/* Create propTable property table with the subunit type property. */
OSIterator * childIterator;
IOFireWireAVCSubUnit * found = NULL;
childIterator = getClientIterator();
if ( childIterator ) {
    OSObject * child;
    while ( ( child = childIterator->getNextObject() ) ) {
        found = OSDynamicCast( IOFireWireAVCSubUnit, child );
        if ( found && found->matchPropertyTable( propTable ) ) {
            break;
        }
        else
            found = NULL;
```

```
    }
    childIterator->release();
    if ( found )
        /* Continue searching for existing subunits. */
}
```

The `getOpenClientIterator` and `getOpenProviderIterator` methods differ from the `getClientIterator` and `getProviderIterator` methods in two important ways. First, as their names suggest, `getOpenClientIterator` and `getOpenProviderIterator` return iterators for only the open clients and providers. For `getOpenClientIterator`, this means an iterator for a provider's clients that have opened the provider, and for `getOpenProviderIterator`, it means an iterator for a client's providers that the client currently has open. Second, IOService uses the `lockForArbitration` method to lock the current object in the iteration so that its state does not change while you are accessing it.

The `getOpenClientIterator` and `getOpenProviderIterator` methods mirror the `getClientIterator` and `getProviderIterator` methods in that the objects the iterator returns are retained as long as the iterator is valid and in that you must release the iterator when you no longer need it.

Getting Other I/O Kit Objects

The remaining IOService access methods are:

- `getPlatform`
- `getResources`
- `getState`

Of these three, two are used almost exclusively by the I/O Kit and IOService. The `getResources` method allows for lazy allocation of resources during the registration process for an IOService object by deferring the allocation until a matching driver is found for the object. The `getState` method returns the IOService state for an IOService object and is not typically used outside of IOService's internal implementations.

In the unlikely event that your driver needs to find out which platform it is running on, it can call the `getPlatform` method, as in this example that determines the platform's computer type:

```
if ( getPlatform()->getMachineType() == kGossamerTypeYosemite )
    /* Do something here. */
```

The `getPlatform` method gives you a reference to the platform expert instance for the computer you're currently running on. With this reference, you can then call platform expert methods, such as `getModelName` and `getMachineName` to get specific information about it (for more information on platform expert methods, see `/System/Library/Frameworks/Kernel.framework/Headers/IOKit/IOPlatformExpert.h`).

Power Management

On a Mac OS X system, changes in power state can happen at any time. Whether it is due to the hot-unplugging of a device or system sleep, your driver must be prepared to handle a change in its power state whenever it happens.

The IOService class includes a large number of methods related to power management but unless your driver serves as a policy maker or as the power controller for your device, you will probably not need to implement or call any of them.

This section briefly introduces policy makers and power controllers and how they can interact with your driver to give context for the IOService power management methods. For more information on power management in general and how to implement policy makers and power controllers in particular, see*I/O Kit Fundamentals*.

The power-management subsystem of a Mac OS X system is responsible for providing enough power to function at the required level while at the same time prolonging display life and conserving power. To do this, power management keeps track of the power needs and states of all devices and subsystems, and uses that information to make power-supply decisions.

Power-Management Entities

A power domain is a switchable source of power in the system that provides power for one or more devices considered to be members of the domain. The root power domain represents the main power of the Mac OS X system itself and contains all other power domains. You can view the (current) hierarchical structure of the root power domain and all its dependent domains in the I/O Registry's IOPower plane by entering `ioreg -p IOPower` at the command line.

Each entity in power management, such as a power domain or device, is represented by an object that inherits from the IOService class. The power management methods IOService provides allow all power-management objects to communicate with each other to manage the power usage of the system.

Two types of power-management objects perform most of the power-related tasks for the system: policy makers and power controllers. A policy maker is usually an instance of an I/O Kit family class, but it can be any object at an appropriate level in a driver stack. The policy maker for a device (or domain) considers factors such as aggressiveness (a measure of how urgently a device should conserve power) when making decisions that affect the power level of that device or domain.

As its name suggests, the power controller for a device is the expert on that device's power capabilities and carries out the decisions of the policy maker. The communication between policy maker and power controller flows in both directions: The policy maker decides to change the power state of a device and tells the power controller and the power controller gives the policy maker information about the device that helps the policy maker make its decisions.

Because a policy maker needs to know about the type of device being controlled, including its power-usage patterns, it is usually implemented as an instance of an I/O Kit family class. For the same reasons, an I/O Kit family class may also choose to implement a power controller for its devices, although it is more common for the driver that actually controls a device's I/O to be that device's power controller. It's important to read the documentation for the family to which your driver belongs, because each family may apportion power management tasks a little differently.

Using the Power Management Methods

If you are implementing a driver that represents physical hardware, such as a PCI device, or a physical subsystem, such as an Ethernet driver, you may be responsible for being both policy maker and power controller for your device. At minimum, you should be familiar with the following subset of IOService power management methods:

- PMinit
- PMstop
- joinPMTree
- registerPowerDriver
- changePowerStateTo
- setAggressiveness
- setPowerState
- activityTickle
- setIdleTimerPeriod
- acknowledgePowerChange
- acknowledgeSetPowerState

If your driver is serving as the power controller for your device, it should first create an array of structures that describe your device's power requirements and capabilities. Each structure is of type `IOPMPowerState` (declared in `IOPMpowerState.h` available at `/System/Library/Frameworks/Kernel.framework/Headers/IOKit/pwr_mgt`). For example, the ApplePCCardSample driver defines the following array:

```
static const IOPMPowerState myPowerStates[ kIOPCCard16DevicePowerStateCount ]
{
    { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 1, 0, IOPMSoftSleep, IOPMSoftSleep, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 1, IOPMPowerOn, IOPMPowerOn, IOPMPowerOn, 0, 0, 0, 0, 0, 0, 0, 0 }
};
```

In order for a policy maker or a power controller to participate in the power-management subsystem, it must first initialize the protected power-management instance variables in the IOPMpriv and IOPMprot objects (for more information on these objects, see *I/O Kit Fundamentals*). A driver does this by calling the `PMinit` method, typically in its `start` method. At the end of a driver's life span, usually in its `stop` method, a driver resigns its power-management responsibilities and disables its power-management participation by calling the `PMstop` method, which erases the IOPMpriv and IOPMprot objects and serves to de-register the driver.

Not surprisingly, different I/O Kit families can handle the exact distribution of power-management responsibilities in different ways. For example, in its `stop` method, the IOFramebuffer superclass calls `PMstop` automatically for all its subclasses, so an IOFramebuffer subclass should not call `PMstop` itself. Before implementing power management in your driver, therefore, be sure to find out how your driver's I/O Kit family assigns power-management responsibilities.

After creating the array of power states, your driver can then volunteer as the power controller for your device by calling the method `registerPowerDriver`, passing it a pointer to itself, the power state array, and the number of states in the array, as in this example from ApplePCCardSample's `start` method:

```
registerPowerDriver ( this, (IOPMPowerState *) myPowerStates,
                    kIOPCCard16DevicePowerStateCount );
```

Although the `PMinit` method initializes the power-management variables, it does not attach the calling driver to the power-management hierarchy, or tree. Therefore, after registering as the power controller, the ApplePCCardSample then becomes a member of the power-management tree by calling the `joinPMTree` method on its provider:

```
provider->joinPMtree ( this );
```

When a power-controller driver needs to change its power state, it calls the `changePowerStateTo` method, passing in the ordinal value of the desired power state in the power state array. Conversely, when a policy maker needs to tell a power controller to change its power state, the policy maker calls the `setPowerState` method. It passes in the ordinal value of the desired power state in the power state array and a pointer to the particular IOService object whose power state the power controller should change.

Aggressiveness is a measure of how aggressive the power-management subsystem should be in conserving power. Policy makers need to be aware of the current aggressiveness in order to appropriately determine when a device or domain is idle. In general, a policy maker should implement the `setAggressiveness` method to receive information about changes in aggressiveness, but it should be sure to invoke its superclass's `setAggressiveness` method, as well.

The IODisplayWrangler, for example, is the policy maker for displays. It senses when a display is idle or active and adjusts the power accordingly. The IODisplayWrangler uses aggressiveness levels from its power domain parent in its calculation of an idle-timer period. When the timer expires and no user activity has occurred since the last timer expiration, the IODisplayWrangler lowers the power state of the display. Listing 3-8 (page 52) shows IODisplayWrangler's implementation of the `setAggressiveness` method.

Listing 3-8 The IODisplayWrangler `setAggressiveness` method

```
IOReturn IODisplayWrangler::setAggressiveness( unsigned long type, unsigned
                                               long newLevel )
{
    if( type == kPMMinutesToDim) {
        // Minutes to dim received.
        if( newLevel == 0) {
            // Power management turned off while idle?
            if( pm_vars->myCurrentState < kIODisplayWranglerMaxPowerState) {
                // Yes, bring displays up again.
                changePowerStateToPriv( kIODisplayWranglerMaxPowerState );
            }
        }
        fMinutesToDim = newLevel;
        fUseGeneralAggressiveness = false;
        // No. Currently in emergency level?
        if( pm_vars->aggressiveness < kIOPowerEmergencyLevel) {
            // No, set new timeout.
            setIdleTimerPeriod( newLevel*60 / 2);
        }

    // general factor received.
    } else if( type == kPMGeneralAggressiveness) {
        // Emergency level?
        if( newLevel >= kIOPowerEmergencyLevel ) {
            // Yes.
            setIdleTimerPeriod( 5 );
        }
        else {
            // No. Coming out of emergency level?
            if( pm_vars->aggressiveness >= kIOPowerEmergencyLevel ) {
                if( fUseGeneralAggressiveness) {
                    // Yes, set new timer period.
                    setIdleTimerPeriod( (333 - (newLevel/3)) / 2 );
                }
                else {
```

```
                    setIdleTimerPeriod( fMinutesToDim * 60 / 2);
                }
            }
            else {
                if( fUseGeneralAggressiveness) {
                    // No, maybe set period.
                    setIdleTimerPeriod( (333 - (newLevel/3)) / 2 );
                }
            }
        }
    }
    super::setAggressiveness(type, newLevel);
    return( IOPMNoErr );
}
```

A policy maker uses the `activityTickle` method to help it determine when a device is idle. If the policy maker is solely responsible for determining idleness, it should implement the `activityTickle` method to intercept any calls by other driver objects and use its own methods to determine idleness periods. In this situation, power-controller drivers call the `activityTickle` method when there is device activity or for any other reason the device needs to be fully powered, with the `kIOPMSubclassPolicy` parameter.

A policy maker that determines idleness in cooperation with IOService, on the other hand, should first call `setIdleTimerPeriod` on its superclass, passing in the number of seconds for the timer interval. If the policy maker needs other objects to inform it of device activity, it should implement `activityTickle` as described above. When it becomes aware of activity, the policy maker should call `activityTickle` on its superclass, passing in the `kIOPMSubclassPolicy1` parameter. Then, when the idle timer expires, the IOService superclass checks to see if `activityTickle(kIOPMSubclassPolicy1)` has been called on it. If it has, then there has been activity and the IOService superclass restarts the idleness timer. If it hasn't, and there has been no activity, the IOService superclass calls `setPowerState` on the power controller to tell it to lower the device's power state to the next lower level.

A policy maker tells interested IOService objects when a device's power state is changing. In return, such an object can tell the policy maker that it is either prepared for the change or needs time to prepare by calling the `acknowledgePowerChange` method on the policy maker.

When a policy maker tells a power controller to change a device's power state, the power controller uses the `acknowledgeSetPowerState` method to tell the policy maker either that it has made the change or that it needs time to make the change.

Memory Mapping and Interrupt Handling

IOService provides a number of methods for low-level access of device memory and interrupt-handling. Most driver writers will not need to use these because the I/O Kit families generally take care of such low-level access. In addition, unless you are implementing an interrupt controller, you should use the interrupt-handling facilities IOInterruptEventSource provides or, for PCI devices, IOFilterInterruptEventSource (both available in `/System/Library/Frameworks/Kernel.framework/Headers/IOKit`.)

If you're writing a device driver for a familyless device, however, you may need to gain direct access to your device's memory or implement an interrupt-controller driver. This section describes the IOService methods that can help you.

Accessing Device Memory

The IOService methods for accessing device memory handle IODeviceMemory objects. IODeviceMemory is a simple subclass of IOMemoryDescriptor, the abstract base class that defines common methods for describing physical or virtual memory. An IODeviceMemory object describes a single range of physical memory on a device and implements a handful of factory methods to create instances with particular memory ranges or subranges.

You can think of the memory-access methods in IOService as wrappers for IOMemoryDescriptor methods. If your device's provider is a memory-mapped device, IOService provides it with its own array of memory descriptors (IODeviceMemory objects). The first member of the array might be the IODeviceMemory object that represents the `vram` and the second might represent the device's registers. The IOService memory-access methods allow you to specify a member of this array so that you can then create a mapping for it and access the memory it represents.

Note: If you are writing a device driver for a PCI device , you can use the API in `IOPCIDevice.h` (available in `/System/Library/Frameworks/Kernel.framework/Headers/IOKit/pci`) for more device-specific support.

IOService defines three methods you can use to get information about and access your device's array of physical memory ranges:

- `getDeviceMemory`
- `getDeviceMemoryCount`
- `getDeviceMemoryWithIndex`

The `getDeviceMemory` method returns the array of IODeviceMemory objects that represent a device's memory-mapped ranges. It is most likely that you will need to access specific members of this array using the other two methods, but a nub might call `getDeviceMemory` on its provider and use the returned array in its `start` method to set up a corresponding array of IODeviceMemory objects.

The `getDeviceMemoryCount` method returns the number of physical memory ranges available for a device; in effect, it returns a value you can use as an index into the IODeviceMemory array. As its name suggests, `getDeviceMemoryWithIndex` returns the IODeviceMemory object representing a memory-mapped range at the specified index.

In its `start` method, the AppleSamplePCI driver uses the `getDeviceMemoryCount` and `getDeviceMemoryWithIndex` methods to display all the device's memory ranges, as shown in Listing 3-9 (page 54).

Listing 3-9 Using `getDeviceMemoryCount` and `getDeviceMemoryWithIndex`

```
bool AppleSamplePCI::start( IOService * provider )
{
    IOMemoryDescriptor * mem;
    /* Other code here */
    fPCIDevice = ( IOPCIDevice * ) provider;
    /* Use IOPCIDevice API to enable memory response from the device */
    fPCIDevice->setMemoryEnable( true );

    /* Use IOLog (defined in IOLib.h) to display the device's memory
        ranges. */
```

```
    for ( UInt32 index = 0;
        index < fPCIDevice->getDeviceMemoryCount();
        index++ ) {

        mem = fPCIDevice->getDeviceMemoryWithIndex( index );
        /* Use assert (defined in IOKit/assert.h) for debugging purposes */
        IOLog( "Range[%ld] %081x\n", index,
                mem->getPhysicalAddress(), mem->getLength() );
    }
    /* Work with a range based on the device's config BAR (base
     address register) here */
}
```

AppleSamplePCI's `start` method uses an IOMemoryDescriptor object instead of an IODeviceMemory object to contain the object returned by `getDeviceMemoryWithIndex` because it uses IOMemoryDescriptor's `getPhysicalAddress` and `getLength` methods.

IOService provides one device memory–mapping method, `mapDeviceMemoryWithIndex`, that maps a physical memory range of a device at the given index (passed in the first parameter). The second parameter of the `mapDeviceMemoryWithIndex` method contains the same options IOMemoryDescriptor's `map` method uses (defined in `/System/Library/Frameworks/IOKit.framwork/Headers/IOTypes.h`), as shown in Table 3-3 (page 55):

Table 3-3 Memory-mapping options for `mapDeviceMemoryWithIndex`

Option name	Description
`kIOMapAnywhere`	Create the mapping anywhere.
`kIOMapInhibitCache`, `kIOMapDefaultCache`, `kIOMapCopybackCache`, `kIOMapWriteThruCache`	Set the appropriate caching.
`kIOMapReadOnly`	Allow only read-only access to the mapped memory.
`kIOMapReference`	Create a new reference to a preexisting mapping.

If you choose to specify a combination of these options, you use a variable of type `IOOptionsBits` and perform a logical `OR` on the options you want.

Further along in its `start` method, the ApplePCCardSample driver calls the `mapDeviceMemoryWithIndex` method on its nub, using the number of windows (or mappings) the nub has created as the index, as Listing 3-10 (page 55) shows.

Listing 3-10 Part of the ApplePCCardSample class declaration

```
/* From ApplePCCardSample class declaration: */
unsigned windowCount;
IOMemoryMap * windowMap[10];

/* Other initialization and configuration performed here. */
/* ... */
/* Find out how many windows we have configured. */
windowCount = nub->getWindowCount();

/* Map in the windows. */
for ( unsigned i = 0; i < windowCount; i++ ) {
```

```
    UInt32 attributes;
    if ( !nub->getWindowAttributes( i, &attributes ))
        /* Log error. */
    windowMap[i] = nub->mapDeviceMemoryWithIndex( i );
    if ( !windowMap[i] )
        /* Log error, call stop on provider, and return false from start. */
}
```

Handling Interrupts

The vast majority of drivers should use the interrupt-handling facilities the IOInterruptEventSource class provides. The work-loop mechanism provides a safe, easy way to handle all types of asynchronous events, including interrupts.

For interrupt-controller or PCI device driver developers, however, IOService provides a handful of methods for low-level interrupt handling, outside the work-loop mechanism:

- `getInterruptType`
- `causeInterrupt`
- `disableInterrupt`
- `enableInterrupt`
- `registerInterrupt`
- `unregisterInterrupt`

The first task in implementing an interrupt controller is to determine which type of interrupt the device uses, edge-triggered or level-sensitive. Most PCI devices use level-sensitive interrupts by specification. The `getInterruptType` method is valid even before you register your interrupt handler so you can choose which handler to register based on the type of the interrupt. The implementation of IOInterruptEventSource does just this in its `init` method, as Listing 3-11 (page 56) shows.

Listing 3-11 Determining interrupt type with `getInterruptType`

```
bool IOInterruptEventSource::init( OSObject *inOwner,
                                   Action inAction = 0,
                                   IOService *inProvider = 0,
                                   int inIntIndex = 0 )
{
    bool res = true;
    if ( !super::init( inOwner, ( IOEventSourceAction) inAction ) )
            return false;

    provider = inProvider;
    autoDisable = false;
    intIndex = -1;

    if ( inProvider ) {
        int intType;
        res = ( kIOReturnSuccess ==
            inProvider->getInterruptType( intIntIndex, &intType ) );
        if ( res ) {
            IOInterruptAction intHandler;
```

```
            autoDisable = ( intType == kIOInterruptTypeLevel );
            if ( autoDisable ) {
                intHandler = ( IOInterruptAction )
                    &IOInterruptEventSource::disableInterruptOccurred;
            }
            else
                intHandler = ( IOInterruptAction )
                    &IOInterruptEventSource::normalInterruptOccurred;

            res = ( kIOReturnSuccess == inProvider->registerInterrupt
                    ( inIntIndex, this, intHandler ) );
            if ( res )
                intIndex = inIntIndex;
        }
    }
    return res;
}
```

After you've determined which type of interrupt is valid for your device, you should register an interrupt handler for the device's interrupt source. The `registerInterrupt` method requires an integer giving the index of the interrupt source in the device, a reference to the interrupt controller class (often the `this` pointer), a reference to the interrupt-handling routine, and an optional reference constant for the interrupt handler's use. Listing 3-11 (page 56) shows how IOInterruptEventSource registers the appropriate interrupt handler.

When you call `registerInterrupt` on an interrupt source, that interrupt source always starts disabled. It does not take interrupts until you enable it with the `enableInterrupt` method.

Important: As soon as you call the `enableInterrupt` method, you could start handling interrupts from the enabled source, even *before* `enableInterrupt` returns. Therefore, make sure you are prepared to handle interrupts before you call the `enableInterrupt` method.

The `disableInterrupt` method disables the physical interrupt source, disabling it for all consumers of the interrupt if it is shared. You should call the `disableInterrupt` method sparingly and only for very short periods of time. If, for some reason, you need to disable an interrupt source for a longer time, you should instead use the `unregisterInterrupt` method to unregister the interrupt handler for a particular source and then reregister it later.

Important: When you call `disableInterrupt`, you must be prepared to handle interrupts until *after* `disableInterrupt` returns. The `disableInterrupt` method waits until all currently active interrupts have been serviced before returning.

Internal services and top-level interrupt handlers sometimes use the `causeInterrupt` method to ensure that certain interrupts behave in an expected manner. Interrupt controllers are not required to implement this method so you should check before calling `causeInterrupt` to tickle your interrupt controller.

By using these methods instead of the IOInterruptEventSource and work-loop mechanism, you are responsible for keeping track of the state of your own interrupts. If, for example, your interrupt source is not registered, you must make sure that your device does not assert that interrupt, otherwise it could adversely affect other sources sharing that interrupt.

Miscellaneous IOService Methods

The remaining IOService methods do not fall neatly into any single, functional category. This section describes the following methods:

- `errnoFromReturn`
- `stringFromReturn`
- `callPlatformFunction`
- `lockForArbitration`
- `unlockForArbitration`

The `errnoFromReturn` and `stringFromReturn` methods are utilities that translate the error codes in `IOReturn.h` (available in `/System/Library/Frameworks/IOKit.framework/Headers`) into more usable formats. The `errnoFromReturn` method translates an `IOReturn` error code into the error codes BSD defines for its functions in `errno.h` (available in `/System/Library/Frameworks/Kernel.framework/Headers/sys`). The IOCDMediaBSDClient class, for example, uses `errnoFromReturn` to return a BSD error code when it encounters errors while processing an `ioctl` system call, as Listing 3-12 (page 58) shows.

Listing 3-12 Using `errnoFromReturn`

```
int IOCDMediaBSDClient::ioctl ( dev_t dev, u_long cmd, caddr_t data,
                                int flags, struct proc * proc )
{
    /* Process a CD-specific ioctl. */
    int error = 0;

    switch ( cmd )
    {
        /* Switch cases not shown. */
    }
    return error ? error : getProvider()->errnoFromReturn ( status );
}
```

The `stringFromReturn` method translates the `IOReturn` error code into an easier-to-read string. For example, IOService translates the error code `kIOReturnLockedRead` into the string "device is read locked".

You can override either `errnoFromReturn` or `stringFromReturn` to interpret family-dependent return codes or if you choose to support other `IOReturn` codes in addition to the ones IOService translates. If you do implement one of these methods, however, you should call the corresponding class in your superclass if you cannot translate the given error code.

The `callPlatformFunction` method is an internal method that routes requests to other I/O Registry objects or resources. There is no need for your driver to call this method.

The `lockForArbitration` and `unlockForArbitration` methods protect an IOService object from changes in its state or ownership. Most drivers do not need to use these methods because they get called when their state should change so they can then synchronize their internal state. Internally, IOService uses these methods extensively in its implementation of the driver life-cycle methods, such as `attach`, `detach`, `open`, and `close`.

Some I/O Kit families also use these methods to prevent changes in an IOService object's state or ownership while accessing it. For example, Listing 3-13 (page 59) shows a fragment of the IOBlockStorageDriver's `mediaStateHasChanged` method, which determines a course of action based on a media's new state.

Listing 3-13 Using `lockForArbitration` and `unlockForArbitration`

```
IOReturn IOBlockStorageDriver::mediaStateHasChanged( IOMediaState state )
{
    IOReturn result;
    /* Determine if media has been inserted or removed. */
    if ( state == kIOMediaStateOnline ) /* Media is now present. */
    {
        /* Allow a subclass to decide whether to accept or reject the
            media depending on tests like password protection. */
        /* ... */
        /* Get new media's parameters. */
        /* ... */
        /* Now make new media show in system. */
        lockForArbitration();
        result = acceptNewMedia(); /* Instantiate new media object */
                                   /* and attach to I/O Registry. */
        if ( result != kIOReturnSuccess )
        {
            /* Deal with error. */
        }
        unlockForArbitration();
        return ( result );
    }
    else { /* Deal with removed media. */
}
```

Making Hardware Accessible to Applications

Let's assume you have written a driver for a device, have thoroughly tested it, and are ready to deploy it. Is your job done? Not necessarily, because there is the perennial problem for driver writers of making their service accessible to user processes. A driver without clients in user space is useless (unless all of its clients reside in the kernel).

This chapter does a few things to help you on this score. It describes the architectural aspects of Mac OS X and the Darwin kernel that underlie the transport of data across the boundary separating the kernel and user space. It describes the alternative APIs on Mac OS X for cross-boundary transport between a driver stack and an application. And it describes how to roll your own solution by writing a custom user client, finally taking you on a detailed tour through an example implementation.

To better understand the information presented in this section, it is recommended that you become familiar with the material in*I/O Kit Fundamentals* and the relevant sections of *Kernel Programming Guide*.

Transferring Data Into and Out of the Kernel

The Darwin kernel gives you several ways to let your kernel code communicate with application code. The specific kernel–user space transport API to use depends on the circumstances.

- If you are writing code that resides in the BSD subsystem, you use the `syscall` or (preferably) the `sysctl` API. You should use the `syscall` API if you are writing a file-system or networking extension.
- If your kernel code is not part of the BSD subsystem (and your code is not a driver), you probably want to use Mach messaging and Mach Inter-Process Communication (IPC). These APIs allow two Mach tasks (including the kernel) to communicate with each other. Mach Remote Process Communication (RPC), a procedural abstraction built on top of Mach IPC, is commonly used instead of Mach IPC.
- You may use memory mapping (particularly the BSD `copyin` and `copyout` routines) and block copying in conjunction with one of the aforementioned APIs to move large or variably sized chunks of data between the kernel and user space.

Finally, there are the I/O Kit transport mechanisms and APIs that enable driver code to communicate with application code. This section describes aspects of the kernel environment that give rise to these mechanisms and discusses the alternatives available to you.

Note: For more information on the other Darwin APIs for kernel–user space transport, see *Kernel Programming Guide*.

Issues With Cross-Boundary I/O

An important feature of the Mac OS X kernel is memory protection. Each process on the system, including the kernel, has its own address space which other processes are not free to access in an unrestricted manner. Memory protection is essential to system stability. It's bad enough when a user process crashes because some other process trashed its memory. But it's catastrophic—a system crash—when the kernel goes down for the same reason.

Largely (but not exclusively) because of memory protection, there are certain aspects of the kernel that affect how cross-boundary I/O takes place, or should take place:

- **The kernel is a slave to the application.** Code in the kernel (such as in a driver) is passive in that it only reacts to requests from processes in user space. Drivers should not initiate any I/O activity on their own.
- **Kernel resources are discouraged in user space.** Application code cannot be trusted with kernel resources such as kernel memory buffers and kernel threads. This kind of exposure leaves the whole system vulnerable; an application can trash critical areas of physical memory or do something globally catastrophic with a kernel thread, crashing the entire system. To eliminate the need for passing kernel resources to user space, the system provides several kernel–user space transport mechanisms for a range of programmatic circumstances.
- **User processes cannot take direct interrupts.** As a corollary to the previous point, kernel interrupt threads cannot jump to user space. Instead, if your application must be made aware of interrupts, it should provide a thread on which to deliver a notification of them.
- **Each kernel–user space transition incurs a performance hit.** The kernel's transport mechanisms consume resources and thus exact a performance penalty. Each trip from the kernel to user space (or vice versa) involves the overhead of Mach RPC calls, the probable allocation of kernel resources, and perhaps other expensive operations. The goal is to use these mechanisms as efficiently as possible.
- **The kernel should contain only code that must be there.** Adding unnecessary code to the kernel—specifically code that would work just as well in a user process—bloats the kernel, potentially destabilizes it, unnecessarily wires down physical memory (making it unavailable to applications), and degrades overall system performance. See *Coding in the Kernel* for a fuller explanation of why you should always seek to avoid putting code in the kernel.

Mac OS 9 Compared

On Mac OS 9, applications access hardware in a way that is entirely different from the way it is done on Mac OS X. The difference in approach is largely due to differences in architecture, particularly in the relationship between an application and a driver.

Unlike Mac OS X, Mac OS 9 does not maintain an inviolable barrier between an application's address space and the address space of anything that would be found in the Mac OS X kernel. An application has access to the address of any other process in the system, including that of a driver.

This access affects how completion routines are invoked. The structure behind all I/O on a Mac OS 9 system is called a parameter block. The parameter block contains the fields typically required for a DMA transfer:

- Host address
- Target address
- Direction of transfer
- Completion routine and associated data

The completion routine is implemented by the application to handle any returned results. The driver maintains a linked list of parameter blocks as I/O requests or jobs for the DMA engine to perform. When a job completes, the hardware triggers an interrupt, prompting the driver to call the application's completion routine. The application code implementing the completion routine runs at "interrupt time"—that is, in the context of the hardware interrupt. This leads to a greater likelihood that a programming error in the completion routine can crash or hang the entire system.

If the same thing with interrupts happened on Mac OS X, there would additionally be the overhead of crossing the kernel–user space boundary (with its performance implications) as well as the risk to system stability that comes with exporting kernel resources to user space.

Programming Alternatives

The I/O Kit gives you several ready-made alternatives for performing cross-boundary I/O without having to add code to the kernel:

- I/O Kit family device interfaces
- POSIX APIs
- I/O Registry properties

When facing the problem of communication between driver and application, you should first consider whether any of these options suits your particular needs. Each of them has its intended uses and each has limitations that might make it unsuitable. However, only after eliminating each of these alternatives as a possibility should you decide upon implementing your own driver–application transport, which is called a custom user client.

Note: This section summarizes information from the document *Accessing Hardware From Applications* that explains how to use device interfaces and how to get device paths for POSIX I/O routines. Refer to that document for comprehensive descriptions of these procedures.

I/O Kit Family Device Interfaces

A device interface is the flip side of what is known as a user client in the kernel. A device interface is a library or plug-in through whose interface an application can access a device. The application can call any of the functions defined by the interface to communicate with or control the device. In turn, the library or plug-in talks with a user-client object (an instance of a subclass of IOUserClient) in a driver stack in the kernel. (See "The Architecture of User Clients" (page 67) for a full description of these types of driver objects.)

Several I/O Kit families provide device interfaces for applications and other user-space clients. These families include (but are not limited to) the SCSI, HID, USB, and FireWire families. (Check the header files in the I/O Kit framework to find out about the complete list of families providing device interfaces.) If your driver is a member of one of these families, your user-space clients need only use the device interface of the family to access the hardware controlled by your driver.

See Finding and Accessing Devices in *Accessing Hardware From Applications* for a detailed presentation of the procedure for acquiring and using device interfaces.

Using POSIX APIs

For each storage, network, and serial device the I/O Kit dynamically creates a device file in the file system's `/dev` directory when it discovers a device and finds a driver for it, either at system startup or as part of its ongoing matching process. If your device driver is a member of the I/O Kit's Storage, Network, or Serial families, then your clients can access your driver's services by using POSIX I/O routines. They can simply use the I/O Registry to discover the device file that is associated with the device your driver controls. Then, with that device file as a parameter, they call POSIX I/O functions to open and close the device and read and write data to it.

Because the I/O Kit dynamically generates the contents of the `/dev` directory as devices are attached and detached, you should never hard-code the name of a device file or expect it to remain the same whenever your application runs. To obtain the path to a device file, you must use device matching to obtain a device path from the I/O Registry. Once you have found the correct path, you can use POSIX functions to access the device. For information on using the I/O Registry to find device-file paths, see *Accessing Hardware From Applications*.

Accessing Device Properties

The I/O Registry is the dynamic database that the I/O Kit uses to store the current properties and relationships of driver objects in a Mac OS X system. APIs in the kernel and in user space give access to the I/O Registry, allowing code to get and set properties of objects in the Registry. This common access makes possible a limited form of communication between driver and application.

All driver objects in the kernel derive from IOService, which is in turn a subclass of the IORegistryEntry class. The methods of IORegistryEntry enable code in the kernel to search the I/O Registry for specific entries and to get and set the properties of those entries. A complementary set of functions (defined in `IOKitLib.h`) exist in the I/O Kit framework. Applications can use the functions to fetch data stored as properties of a driver object or to send data to a driver object.

This property-setting mechanism is suitable for situations where the following conditions are true:

- The driver does not have to allocate permanent resources to complete the transaction.
- The application is transferring—by copy—a limited amount of data (under a page)

  With the property-setting mechanism, the application can pass arbitrary amounts of data by reference (that is, using pointers).
- The data sent causes no change in driver state or results in a single, permanent change of state.
- You control the driver in the kernel (and thus can implement the `setProperties` method described below).

The property-setting mechanism is thus suitable for some forms of device control and is ideal for one-shot downloads of data, such as for loading firmware. It is not suitable for connection-oriented tasks because such tasks usually require the allocation of memory or the acquisition of devices. Moreover, this mechanism does not allow the driver to track when its clients die.

The general procedure for sending data from an application to a driver object as a property starts with establishing a connection with the driver. The procedure for this, described in "The Basic Connection and I/O Procedure" (page 74), consists of three steps:

1. Getting the I/O Kit master port
2. Obtaining an instance of the driver
3. Creating a connection

Once you have a connection, do the following steps:

1. Call the `IOConnectSetCFProperties` function, passing in the connection and a Core Foundation container object, such as a CFDictionary.

   The Core Foundation object contains the data you want to pass to the driver. Note that you can call `IOConnectSetCFProperty` instead if you want to pass only a single, value-type Core Foundation object, such as a CFString or a CFNumber and that value's key. Both function calls cause the invocation of the `IORegistryEntry::setProperties` method in the driver.

2. In the driver, implement the `setProperties` method.

   Before it invokes this method, the I/O Kit converts the Core Foundation object passed in by the user process to a corresponding libkern container object (such as OSDictionary). In its implementation of this method, the driver object extracts the data from the libkern container object and does with it what is expected.

Note: Instead of calling `IOConnectSetCFProperties` you can call `IORegistryEntrySetCFProperties`. This latter function is more convenient in those instances where you have an `io_service_t` handle available, such as from calling `IOIteratorNext`.

The Core Foundation object passed in by the user process must, of course, have a libkern equivalent. Table 4-1 (page 65) shows the allowable Core Foundation types and their corresponding libkern objects.

Table 4-1 Corresponding Core Foundation and libkern container types

Core Foundation	libkern
CFDictionary	OSDictionary
CFArray	OSArray
CFSet	OSSet
CFString	OSString
CFData	OSData

Core Foundation	libkern
CFNumber	OSNumber
CFBoolean	OSBoolean

The following example (Listing 4-1 (page 66)) shows how the I/O Kit's Serial family uses the I/O Registry property-setting mechanism to let a user process make a driver thread idle until a serial port is free to use (when there are devices, such as a modem and a fax, competing for the port).

Listing 4-1 Controlling a serial device using `setProperties`

```
IOReturn IOSerialBSDClient::
setOneProperty(const OSSymbol *key, OSObject *value)
{
    if (key == gIOTTYWaitForIdleKey) {
        int error = waitForIdle();
        if (ENXIO == error)
            return kIOReturnOffline;
        else if (error)
            return kIOReturnAborted;
        else
            return kIOReturnSuccess;
    }

    return kIOReturnUnsupported;
}

IOReturn IOSerialBSDClient::
setProperties(OSObject *properties)
{
    IOReturn res = kIOReturnBadArgument;

    if (OSDynamicCast(OSString, properties)) {
        const OSSymbol *propSym =
            OSSymbol::withString((OSString *) properties);
        res = setOneProperty(propSym, 0);
        propSym->release();
    }
    else if (OSDynamicCast(OSDictionary, properties)) {
        const OSDictionary *dict = (const OSDictionary *) properties;
        OSCollectionIterator *keysIter;
        const OSSymbol *key;

        keysIter = OSCollectionIterator::withCollection(dict);
        if (!keysIter) {
            res = kIOReturnNoMemory;
            goto bail;
        }

        while ( (key = (const OSSymbol *) keysIter->getNextObject()) ) {
            res = setOneProperty(key, dict->getObject(key));
            if (res)
                break;
        }
```

```
        keysIter->release();
    }

bail:
    return res;
}
```

Custom User Clients

If you cannot make your hardware properly accessible to applications using I/O Kit's off-the-shelf device interfaces, POSIX APIs, or I/O Registry properties, then you'll probably have to write a custom user client. To reach this conclusion, you should first have answered "no" the following questions:

- If your device a member of an I/O Kit family, does that family provide a device interface?
- Is your device a serial, networking, or storage device?
- Are I/O Registry properties sufficient for the needs of the application? (If you need to move huge amounts of data, or if you don't have control over the driver code, then they probably aren't.)

If you have determined that you need to write a custom user client for your hardware and its driver, read on for the information describing how to do this.

Writing a Custom User Client

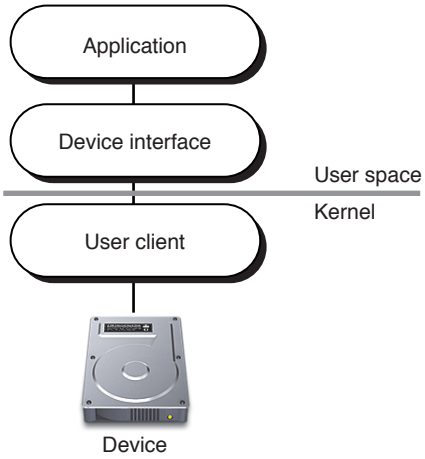
This section discusses the architecture of custom user clients, offers considerations for their design, and describes the API and procedures for implementing a custom user client. See the concluding section "A Guided Tour Through a User Client" (page 93) for a guided tour through a fairly sophisticated user client.

The Architecture of User Clients

A user client provides a connection between a driver in the kernel and an application or other process in user space. It is a transport mechanism that tunnels through the kernel–user space boundary, enabling applications to control hardware and transfer data to and from hardware.

A user client actually consists of two parts, one part for each side of the boundary separating the kernel from user space (see *Kernel Programming Guide* for a detailed discussion of the kernel–user space boundary). These parts communicate with each other through interfaces conforming to an established protocol. For the purposes of this discussion, the kernel half of the connection is the user client proper; the part on the application side is called a device interface. Figure 4-1 (page 68) illustrates this design

Figure 4-1 Architecture of user clients



Although architecturally a user client (proper) and its device interface have a close, even binding relationship, they are quite different programmatically.

- A user client is a driver object (a category that includes nubs as well as drivers). A user client is thus a C++ object derived from IOService, the base class for I/O Kit driver objects, which itself ultimately derives from the libkern base class OSObject.

  Because of its inheritance from IOService, a driver object such as a user client participates in the driver life cycle (initialization, starting, attaching, probing, and so on) and within a particular driver stack has client-provider relationships with other driver objects in the kernel. To a user client's provider—the driver that is providing services to it, and the object with which the application is communicating—the user client looks just like another client within the kernel.

- A device interface is a user-space library or other executable associated with an application or other user process. It is compiled from any code that can call the functions in the I/O Kit framework and is either linked directly into a Mach-O application or is indirectly loaded by the application via a dynamic shared library or a plug-in such as afforded by the Core Foundation types CFBundle and CFPlugIn. (See "Implementing the User Side of the Connection" (page 74) for further information.)

Custom user-client classes typically inherit from the IOUserClient helper class. (They could also inherit from an I/O Kit family's user-client class, which itself inherits from IOUserClient, but this is not a recommended approach; for an explanation why, see the introduction to the section "Creating a User Client Subclass" (page 80).) The device-interface side of the connection uses the C functions and types defined in the I/O Kit framework's `IOKitLib.h`.

The actual transport layer enabling communication between user processes and device drivers is implemented using a private programming interface based on Mach RPC.

Types of User-Client Transport

The I/O Kit's APIs enable several different types of transport across the boundary between the kernel and user space:

- **Passing untyped data**: This mechanism uses arrays of structures containing pointers to the methods to invoke in a driver object; the methods must conform to prototypes for primitive functions with parameters only indicating general type (scalar for a single, 32-bit value or structure for a group of values), number of scalar parameters, size of structures, and direction (input or output). The passing of untyped data using this mechanism can be synchronous or asynchronous.
- **Sharing memory**: This is a form of memory mapping in which one or more pages of memory are mapped into the address space of two tasks—in this case, the driver and the application process. Either process can then access or modify the data stored in those shared pages. The user-client mechanism for shared memory uses IOMemoryDescriptor objects on the kernel side and buffer pointers `vm_address_t` on the user side to map hardware registers to user space. This method of data transfer is intended for hardware that is not DMA-based and is ideal for moving large amounts of data between the hardware and the application. User processes can also map their memory into the kernel's address space.
- **Sending notifications**: This mechanism passes notification ports in and out of the kernel to send notifications between the kernel and user processes. These methods are used in asynchronous data-passing.

An important point to keep in mind is that the implementation of a user client is not restricted to only one of the mechanisms listed above. It can use two or more of them; for example, it might used the synchronous untyped-data mechanism to program a DMA engine and shared memory for the actual data transfer.

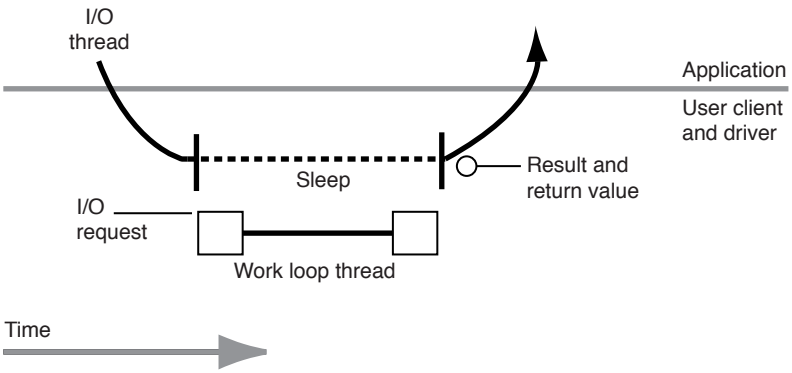
Synchronous Versus Asynchronous Data Transfer

Two styles of untyped-data passing are possible with the I/O Kit's user-client APIs: Synchronous and asynchronous. Each has its strengths and drawbacks, and each is more suitable to certain characteristics of hardware and user-space API. Although the asynchronous I/O model is somewhat comparable to the way Mac OS 9 applications access hardware, it is different in some respects. The most significant of these differences is an aspect of architecture shared with the synchronous model: In Mac OS X, the client provides the thread on which I/O completion routines are called, but the kernel controls the thread. I/O completion routines execute outside the context of the kernel.

The following discussion compares the synchronous (blocking) I/O and asynchronous (non-blocking, with completion) I/O models from an architectural perspective and without discussion of specific APIs. For an overview of those APIs, see "Creating a User Client Subclass" (page 80).

In synchronous I/O, the user process issues an I/O request on a thread that calls into the kernel and blocks until the I/O request has been processed (completed). The actual I/O work is completed on the work-loop thread of the driver. When the I/O completes, the user client wakes the user thread, gives it any results from the I/O operation, and returns control of the thread to the user process. After handling the result of the I/O, the thread delivers another I/O request to the user client, and the process starts again.

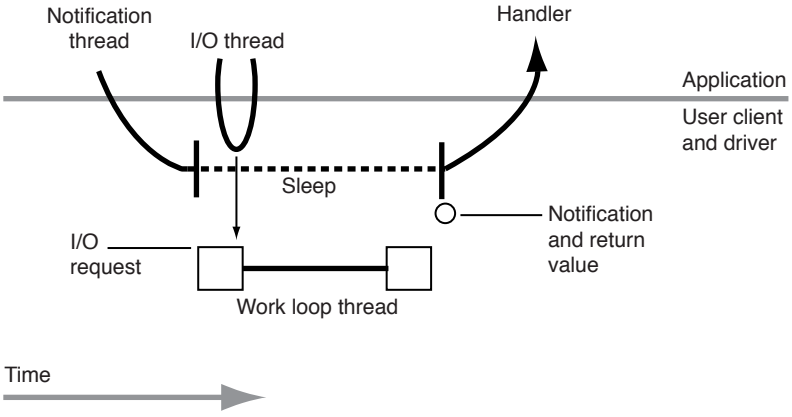
Figure 4-2 Synchronous I/O between application and user client



The defining characteristic of the synchronous model is that the client makes a function call into the kernel that doesn't return until the I/O has completed. The major disadvantage of the synchronous approach is that the thread that issues the I/O request cannot do any more work until the I/O completes. However, it is possible to interrupt blocking synchronous routines by using signals, for example. In this case, the user client has to know that signals might be sent and how to handle them. It must be prepared to react appropriately in all possible situations, such as when an I/O operation is in progress when the signal is received.

In asynchronous I/O, the user-process client has at least two threads involved in I/O transfers. One thread delivers an I/O request to the user client and returns immediately. The client also provides a thread to the user client for the delivery of notifications of I/O completions. The user client maintains a linked list of these notifications from prior I/O operations. If there are pending notifications, the user client invokes the notification thread's completion routine, passing in the results of an I/O operation. Otherwise, if there are no notifications in this list, the user client puts the notification thread to sleep.

Figure 4-3 Asynchronous I/O between application and user client



The user process should create and manage the extra threads in this model using some user-level facility such as BSD pthreads. This necessity points at the main drawback of the asynchronous model: The issue of thread management in a multithreaded environment. This is something that is difficult to do right. Another problem with asynchronous I/O is related to performance; with this type of I/O there are two kernel–user space round-trips per I/O. One way to mitigate this problem is to batch completion notifications and have

the notification thread process several of them at once. For the asynchronous approach, you also might consider basing the client's I/O thread on a run-loop object (CFRunLoop); this object is an excellent multiplexor, allowing you to have different user-space event sources.

So which model for I/O is better, synchronous or asynchronous? As with many aspects of design, the answer is a definite "it depends." It depends on any legacy application code you're working with, it depends on the sophistication of your thread programming, and it depends on the rate of I/O. The asynchronous approach is good when the number of I/O operations per second is limited (well under 1000 per second). Otherwise, consider the synchronous I/O model, which takes better advantage of the Mac OS X architecture.

Factors in User Client Design

Before you start writing the code of your user client, take some time to think about its design. Think about what the user client is supposed to do, and what is the best programmatic interface for accomplishing this. Keeping some of the points raised in "Issues With Cross-Boundary I/O" (page 62) in mind, consider the following questions:

- What will be the effect of your design on performance, keeping in mind that each kernel–user space transition exacts a performance toll?

  If your user client's API is designed properly, you should need at most one boundary crossing for each I/O request. Ideally, you can batch multiple I/O requests in a single crossing.

- Does your design put any code in the kernel that could work just as well in user space?

  Remember that code in the kernel can be destabilizing and a drain on overall system resources.

- Does the API of your device interface (the user-space side of the user client) expose hardware details to clients?

  A main feature of the user-space API is to isolate applications from the underlying hardware and operating system.

The following sections describe these and other issues in more detail.

Range of Accessibility

The design of the user side of a user-client connection depends on the probable number and nature of the applications (and other user processes) that want to communicate with your driver. If you're designing your user client for only one particular application, and that application is based on Mach-O object code, then you can incorporate the connection and I/O code into the application itself. See "The Basic Connection and I/O Procedure" (page 74) for the general procedure.

However, a driver writer often wants his driver accessible by more than one application. The driver could be intended for use by a family of applications made by a single software developer. Or any number of applications—even those you are currently unaware of—should be able to access the services of the driver. In these situations, you should put the code related to the user side of the user-client connection into a separate module, such as a shared library or plug-in. This module, known as a device interface, should abstract common connection and I/O functionality and present a friendly programmatic interface to application clients.

So let's say you've decided to put your connection and I/O code into a device interface; you now must decide what form this device interface should take. The connection and I/O code must call functions defined in the I/O Kit framework, which contains a Mach-O dynamic shared library; consequently, all device interfaces should

be built as executable code based on the Mach-O object-file format. The device interface can be packaged as a bundle containing a dynamic shared library or as a plug-in. In other words, the common API choice is between CFBundle (or Cocoa's NSBundle) or CFPlugIn.

The decision between bundle and plug-in is conditioned by the nature of the applications that will be the clients of your user client. If there is a good chance that CFM-based applications will want to access your driver, you should use the CFBundle APIs because CFBundle provides cross-architecture capabilities. If you require a more powerful abstraction for device accessibility, and application clients are not likely to be CFM-based, you can use the CFPlugIn APIs. As an historical note, the families of the I/O Kit use CFPlugIns for their device interfaces because these types of plug-ins provide a greater range of accessibility by enabling third-party developers to create driver-like modules in user space.

If only one application is going to be the client of your custom user client, but that application is based on CFM-PEF object code, you should create a Mach-O bundle (using CFBundle or NSBundle APIs) as the device interface for the application.

In most cases, you can safely choose CFBundle (or NSBundle) for your device interface. In addition to their capability for cross-architecture calling, these bundle APIs make it easy to create a device interface.

Design of Legacy Applications

A major factor in the design of your user client is the API of applications that currently access the hardware on other platforms, such as Mac OS 9 or Windows. Developers porting these applications to Mac OS X will (understandably) be concerned about how hard it will be to get their applications to work with a custom user client. They will probably want to move over as much of their application's hardware-related code as they can to Mac OS X, but this may not be easy to do.

For example, if the application API is based on interrupt-triggered asynchronous callbacks, such as on Mac OS 9, that API is not suitable for Mac OS X, where the primary-interrupt thread must remain in the kernel. Although the I/O Kit does have APIs for asynchronous I/O, these APIs are considerably different than those in Mac OS 9. Moreover, the preferred approach for Mac OS X is to use synchronous calls. So this might be a good opportunity for the application developer to revamp his hardware-API architecture.

If application developers decide to radically redesign their hardware API, the design of that API should influence the choices made for kernel–user space transport. For example, if the high-level API is asynchronous with callbacks, a logical choice would be to base the new application API on the I/O Kit's asynchronous untyped data–passing API. On the other hand, if the high-level API is already synchronous, than the I/O Kit's synchronous untyped data–passing API should clearly be used. The synchronous API is much easier and cleaner to implement, and if done properly does not suffer performance-wise in comparison with the asynchronous approach.

Hardware

The design for your user client depends even more on the hardware your driver is controlling. Your user-client API needs to accommodate the underlying hardware. Two issues here are data throughput and interrupt frequency. If the data rates are quite large, such as with a video card, then try using mapped memory. If the hardware delivers just a few interrupts a second, you can consider handling those interrupts in user space using some asynchronous notification mechanism. But there are latency problems in such mechanisms, so if the hardware produces thousands of interrupts a second, you should handle them in the kernel code.

Finally, there is the issue of hardware memory management. Perhaps the most important aspect of a device, in terms of user-client design, is its memory-management capabilities. These capabilities affect how applications can access hardware registers. The hardware can use either PIO (Programmed Input/Output) or DMA (Direct

Memory Access). With PIO, the CPU itself moves data between a device and system (physical) memory; with DMA, a bus controller takes on this role, freeing up the microprocessor for other tasks. Almost all hardware now uses DMA, but some older devices still use PIO.

The simplest approach to application control of a device is to map device resources (such as hardware registers or frame buffers) into the address space of the application process. However, this approach poses a considerable security risk and should only be attempted with caution. Moreover, mapping registers into user space is not a feasible option with DMA hardware because the user process won't have access to physical memory, which it needs. DMA hardware requires that the I/O work be performed inside the kernel where the virtual-memory APIs yield access to physical addresses. If your device uses DMA memory-management, it is incumbent upon you to find the most efficient way for your driver to do the I/O work.

Given these requirements, you can take one of four approaches in the design of your user client. The first two are options if your hardware memory management is accessed through PIO:

- **Full PIO memory management**. Because you don't require interrupts or physical-memory access, you can map the hardware registers into your application's address space and control the device from there.
- **PIO memory management with interrupts**. If the PIO hardware uses interrupts, you must attempt a modified version of the previous approach. You can map the registers to user space, but the user process has to provide the thread to send interrupt notifications on. The drawback of this approach is that the memory management is not that good; it is suitable only for low data throughput.

See "Mapping Device Registers and RAM Into User Space" (page 92) for more information on these two memory-mapping approaches.

The next two design approaches are appropriate to hardware that uses DMA for memory management. With DMA, your code requires the physical addresses of the registers and must deal with the interrupts signalled by the hardware. If your hardware fits this description, you should use a design based on the untyped data–passing mechanism of the IOUserClient class and handle the I/O within your user client and driver. There are two types of such a design:

- **Function-based user client**. This kind of user client defines complementary sets of functions on both sides of the kernel–user space boundary (for example, `WriteBlockToDevice` or `ScanImage`). Calling one function on the user side results in the invocation of the function on the kernel side. Unless the set of functions is small, you should not take this approach because it would put too much code in the kernel.
- **Register-based task files**. A task file is an array that batches a series of commands for getting and setting register values and addresses. The user client implements only four "primitive" functions that operate on the contents of the task file. Task files are fully explained in the following section, "Task Files" (page 73).

User clients using register-based task files are the recommended design approach when you have DMA hardware. See "Passing Untyped Data Synchronously" (page 85) for examples.

Task Files

Task files give you an efficient way to send I/O requests from a user process to a driver object. A task file is an array containing a series of simple commands for the driver to perform on the hardware registers it controls. By batching multiple I/O requests in this fashion, task files mitigate the performance penalty for crossing the boundary between kernel and user space. You only need one crossing to issue multiple I/O requests.

On the kernel side, all you need are four "primitive" methods that perform the basic operations possible with hardware registers:

- Get value in register *x*
- Set register *x* to value *y*
- Get address in register *x*
- Set register *x* to address *y*

This small set of methods limits the amount of code in the kernel that is dedicated to I/O for the client and moves most of this code to the user-space side of the design. The device interface presents an interface to the application that is more functionally oriented. These functions are implemented, however, to "break down" functional requests into a series of register commands.

See "Passing Untyped Data Synchronously" (page 85) for an example of task files.

A Design Scenario

The first step in determining the approach to take is to look at the overall architecture of the system and decide whether any existing solutions for kernel–user space I/O are appropriate. If you decide you need a custom user client, then analyze what is the design approach to take that is most appropriate to your user-space API and hardware.

As an example, say you have a PCI card with digital signal processing (DSP) capabilities. There is no I/O Kit family for devices of this type, so you know that there cannot be any family device interface that you can use. Now, let's say the card uses both DMA memory management and interrupts, so there must be code inside the kernel to handle these things; hence, a driver must be written to do this, and probably a user client. Because a large amount of DSP data must be moved to and from the card, I/O Registry properties are not an adequate solution. Thus a custom user client is necessary.

On another platform there is user-space code that hands off processing tasks to the DSP. This code works on both Mac OS 9 and Windows. Fortunately, the existing API is completely synchronous; there can be only one outstanding request per thread. This aspect of the API makes it a logical step to adapt the code to implement synchronous data passing in the user client and map the card memory into the application's address space for DMA transfers.

Implementing the User Side of the Connection

Of course, if you're writing the code for the kernel side of the user-client connection, you're probably going to write the complementary code on the user side. First, you should become familiar with the C functions in the I/O Kit framework, especially the ones in `IOKitLib.h`. These are the routines around which you'll structure your code. But before setting hand to keyboard, take a few minutes to decide what your user-space code is going to look like, and how it's going to be put together.

The Basic Connection and I/O Procedure

You must complete certain tasks in the user-side code for a connection whether you are creating a library or incorporating the connection and I/O functionality in a single application client. This section summarizes those tasks, all of which involve calling functions defined in `IOKitLib.h`.

Note: This procedure is similar to the one described in *Accessing Hardware From Applications* for accessing a device interface.

Defining Common Types

The user client and the application or device interface must agree on the indexes into the array of method pointers. You typically define these indexes as `enum` constants. Code on the driver and the user side of a connection must also be aware of the data types that are involved in data transfers. For these reasons, you should create a header file containing definitions common to both the user and kernel side, and have both application and user-client code include this file.

As illustration, Listing 4-2 (page 75) shows the contents of the SimpleUserClient project's common header file:

Listing 4-2 Common type definitions for SimpleUserClient

```
typedef struct MySampleStruct
{
    UInt16 int16;
    UInt32 int32;
} MySampleStruct;

enum
{
    kMyUserClientOpen,
    kMyUserClientClose,
    kMyScalarIStructImethod,
    kMyScalarIStructOmethod,
    kMyScalarIScalarOmethod,
    kMyStructIStructOmethod,
    kNumberOfMethods
};
```

Get the I/O Kit Master Port

Start by calling the `IOMasterPort` function to get the "master" Mach port to use for communicating with the I/O Kit. In the current version of Mac OS X, you must request the default master port by passing the constant `MACH_PORT_NULL`.

```
kernResult = IOMasterPort(MACH_PORT_NULL, &masterPort);
```

Obtain an Instance of the Driver

Next, find an instance of the driver's class in the I/O Registry. Start by calling the `IOServiceMatching` function to create a matching dictionary for matching against all devices that are instances of the specified class. All you need to do is supply the class name of the driver. The matching information used in the matching dictionary may vary depending on the class of service being looked up.

```
classToMatch = IOServiceMatching(kMyDriversIOKitClassName);
```

Next call `IOServiceGetMatchingServices`, passing in the matching dictionary obtained in the previous step. This function returns an iterator object which you use in a call to `IOIteratorNext` to get each succeeding instance in the list. Listing 4-3 (page 76) illustrates how you might do this.

Listing 4-3 Iterating through the list of current driver instances

```
    kernResult = IOServiceGetMatchingServices(masterPort, classToMatch,
&iterator);

    if (kernResult != KERN_SUCCESS)
    {
        printf("IOServiceGetMatchingServices returned %d\n\n", kernResult);
        return 0;
    }

    serviceObject = IOIteratorNext(iterator);
    IOObjectRelease(iterator);
```

In this example, the library code grabs the first driver instance in the list. With the expandable buses in most computers nowadays, you might have to present users with the list of devices and have them choose. Be sure to release the iterator when you are done with it.

Note that instead of calling `IOServiceMatching`, you can call `IOServiceAddMatchingNotification` and have it send notifications of new instances of the driver class as they load.

Create a Connection

The final step is creating a connection to this driver instance or, more specifically, to the user-client object on the other side of the connection. A connection, which is represented by an object of type `io_connect_t`, is a necessary parameter for all further communication with the user client.

To create the connection, call `IOServiceOpen`, passing in the driver instance obtained in the previous step along with the current Mach task. This call invokes `newUserClient` in the driver instance, which results in the instantiation, initialization, and attachment of the user client. If a driver specifies the `IOUserClientClass` property in its information property list, the default `newUserClient` implementation does these things for the driver. In almost all cases, you should specify the `IOUserClientClass` property and rely on the default implementation.

Listing 4-4 (page 76) shows how the SimpleUserClient project gets a Mach port, obtains an instance of the driver, and creates a connection to the user client.

Listing 4-4 Opening a driver connection via a user client

```
    // ...
    kern_return_t   kernResult;
    mach_port_t     masterPort;
    io_service_t    serviceObject;
    io_connect_t    dataPort;
    io_iterator_t   iterator;
    CFDictionaryRef classToMatch;
    // ...
    kernResult = IOMasterPort(MACH_PORT_NULL, &masterPort);

    if (kernResult != KERN_SUCCESS)
    {
        printf( "IOMasterPort returned %d\n", kernResult);
        return 0;
    }
    classToMatch = IOServiceMatching(kMyDriversIOKitClassName);
```

```
    if (classToMatch == NULL)
    {
        printf( "IOServiceMatching returned a NULL dictionary.\n");
        return 0;
    }
    kernResult = IOServiceGetMatchingServices(masterPort, classToMatch,
                                              &iterator);

    if (kernResult != KERN_SUCCESS)
    {
        printf("IOServiceGetMatchingServices returned %d\n\n", kernResult);
        return 0;
    }


    serviceObject = IOIteratorNext(iterator);

    IOObjectRelease(iterator);

    if (serviceObject != NULL)
    {
        kernResult = IOServiceOpen(serviceObject, mach_task_self(), 0,
                                   &dataPort);

        IOObjectRelease(serviceObject);

        if (kernResult != KERN_SUCCESS)
        {
            printf("IOServiceOpen returned %d\n", kernResult);
            return 0;
        }
    // ...
```

Open the User Client

After you have created a connection to the user client, you should open it. The application or device interface should always give the commands to open and close the user client. This semantic is necessary to ensure that only one user-space client has access to a device of a type that permits only exclusive access.

The basic procedure for requesting the user client to open is similar to an I/O request: The application or device interface calls an `IOConnectMethod` function, passing in an index to the user client's `IOExternalMethod` array. The SimpleUserClient project defines `enum` constants for both the open and the complementary close commands that are used as indexes in the user client's `IOExternalMethod` array.

```
enum{    kMyUserClientOpen,    kMyUserClientClose,    // ...};
```

Then the application (or device-interface library) calls one of the `IOConnectMethod` functions; any of these functions can be used because no input data is passed in and no output data is expected. The SimpleUserClient project uses the `IOConnectMethodScalarIScalarO` function (see Listing 4-5 (page 77), which assumes the prior programmatic context shown in Listing 4-4 (page 76)).

Listing 4-5 Requesting the user client to open

```
// ...
    kern_return_t   kernResult;
// ...
    kernResult = IOConnectMethodScalarIScalarO(dataPort, kMyUserClientOpen,
```

```
                                                    0, 0);
    if (kernResult != KERN_SUCCESS)
        {
            IOServiceClose(dataPort);
            return kernResult;
        }
    // ...
```

As this example shows, if the result of the call is not `KERN_SUCCESS`, then the application knows that the device is being used by another application. The application (or device interface) then closes the connection to the user client and returns the call result to its caller. Note that calling `IOServiceClose` results in the invocation of `clientClose` in the user-client object in the kernel.

For a full description of the open-close semantic for enforcing exclusive device access, see "Exclusive Device Access and the open Method" (page 82).

Send and Receive Data

Once you have opened the user client, the user process can begin sending data to it and receiving data from it. The user process initiates all I/O activity, and the user client (and its provider, the driver) are "slaves" to it, responding to requests. For passing untyped data, the user process must use the `IOConnectMethod` functions defined in `IOKitLib.h`. The names of these functions indicate the general types of the parameters (scalar and structure) and the direction of the transfer (input and output). Table 4-2 (page 78) lists these functions.

Table 4-2 IOConnectMethod functions

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

The parameters of these functions include the connection to the user client and the index into the array of method pointers maintained by the user client. Additionally, they specify the number of scalar values (if any) and the size of any structures as well as the values themselves, the pointers to the structures, and pointers to buffers for any returned values,.

For instance, the `IOConnectMethodScalarIStructureO` function is defined as:

```
kern_return_t
IOConnectMethodScalarIStructureO(
    io_connect_t    connect,
    unsigned int    index,
    IOItemCount     scalarInputCount,
    IOByteCount *   structureSize,
    ... );
```

The parameters of this function are similar to those of the other `IOConnectMethod` functions.

- The *`connect`* parameter is the connection object obtained through the `IOServiceOpen` call (`dataPort` in the code snippet in Listing 4-4 (page 76)).
- The *`index`* parameter is the index into the user client's `IOExternalMethod` array.
- The *`scalarInputCount`* parameter is the number of scalar input values.
- The *`structureSize`* parameter is the size of the returned structure.

Because these functions are defined as taking variable argument lists, following *`structureSize`* are, first, the scalar values and then a pointer to a buffer the size of *`structureSize`*. The application in the SimpleUserClient project uses the `IOConnectMethodScalarIStructureO` function as shown in Listing 4-6 (page 79).

Listing 4-6 Requesting I/O with the `IOConnectMethodScalarIStructureO` function

```
kern_return_t
MyScalarIStructureOExample(io_connect_t dataPort)
{
    MySampleStruct  sampleStruct;
    int             sampleNumber1 = 154;    // This number is random.
    int             sampleNumber2 = 863;    // This number is random.
    IOByteCount     structSize = sizeof(MySampleStruct);
    kern_return_t   kernResult;

    kernResult = IOConnectMethodScalarIStructureO(dataPort,
                        kMyScalarIStructOmethod, // method index
                        2,  // number of scalar input values
                        &structSize, // size of ouput struct
                        sampleNumber1,  // scalar input value
                        sampleNumber2,  // another scalar input value
                        &sampleStruct   // pointer to output struct
                        );

    if (kernResult == KERN_SUCCESS)
    {
        printf("kMyScalarIStructOmethod was successful.\n");
        printf("int16 = %d, int32 = %d\n\n", sampleStruct.int16,
                (int)sampleStruct.int32);
        fflush(stdout);
    }
    return kernResult;
}
```

Close the Connection

When you have finished your I/O activity, first issue a close command to the user client to have it close its provider. The command takes a form similar to that used to issue the open command. Call an `IOConnectMethod` function, passing in a constant to be used as an index into the user client's `IOExternalMethod` array. In the SimpleUserClient project, this call is the following:

```
kernResult = IOConnectMethodScalarIScalarO(dataPort, kMyUserClientClose, 0,
                                           0);
```

Finally, close the connection and free up any resources. To do so, simply call `IOServiceClose` on your `io_connect_t` connection.

Aspects of Design for Device Interfaces

When you design your device interface, try to move as much code and logic into it as possible. Only put code in the kernel that absolutely has to be there. The user-interface code in the kernel should be tightly associated with the hardware, especially when the design is based on the task-file approach.

One reason for this has been stressed before: Code in the kernel can be a drain on performance and a source of instability. But another reason should be just as important to developers. User-space code is *much* easier to debug than kernel code.

Creating a User Client Subclass

When you create a user client for your driver, you must create a subclass of IOUserClient. In addition to completing certain tasks that all subclasses of IOUserClient must do, you must write code that is specific to *how* the user client transfers data:

- Using the untyped-data mechanism synchronously (blocking)
- Using the untyped-data mechanism asynchronously (non-blocking, with invocation of completion routine)
- Using the memory-mapping APIs (for PIO hardware)

This section describes all three approaches, but only the first one is covered in detail because it is the most common case. It also discusses the synchronous untyped-data mechanism in the context of register task files because that is the recommended approach for user clients of this sort.

This section does not cover aspects of subclassing family user-client classes. Such classes tend to be complex and tightly integrated into other classes of the family. Of course, you could look at the open-source implementation code to understand how the are constructed and integrated, but still subclassing a family user client is not a recommended approach.

Note: Some of the sample code in this section is taken from the SimpleUserClient example project, which you can download from Darwin Sample Code.

User-Client Project Basics

A user client is, first and foremost, a driver object. It is at the "top" of a driver stack between its provider (the driver) and its client (the user process). As a driver object, it must participate in the driver life-cycle by implementing the appropriate methods: `start`, `open`, and so on (see *I/O Kit Fundamentals* for a description of the driver life cycle). It must communicate with its provider at the appropriate moments. And it must also maintain a connection with its client (the user process) along with any state related to that connection; additionally, it's the user client's responsibility to clean up when the client goes away.

Given the close relationship between a driver and its user client, it's recommended that you include the source files for your user client in the project for your driver. If you want the I/O Kit, in response to `IOServiceOpen` being called in user space, to automatically allocate, start, and attach an instance of your user-client subclass, specify the `IOUserClientClass` property in the information property list of your driver.

The value of the property should be the full class name of your user client. Alternatively, your driver class can implement the IOService method `newUserClient` to create, attach, and start an instance of the your IOUserClient subclass.

In the user client's header file, declare the life-cycle methods that you are overriding; these can include `start`, `message`, `terminate`, and `finalize`. These messages are propagated up the driver stack, from the driver object closest to the hardware to the user client. Also declare `open` and `close` methods; messages invoking these methods are propagated in the opposite direction, and are originated by the application or device interface itself. The `open` method, in which the user client opens its provider, is particularly important as the place where exclusive device access is enforced. For more on the user client's `open` and `close` methods, see "Exclusive Device Access and the open Method" (page 82); for the application's role in this, see "Open the User Client" (page 77).

There is one particular thing to note about the `start` method. In your implementation of this method, verify that the passed-in provider object is an instance of your driver's class (using `OSDynamicCast`) and assign it to an instance variable. Your user client needs to send several messages to its provider during the time it's loaded, so it's helpful to keep a reference to the provider handy.

You'll also have to declare and implement some methods specific to initialization of the user client and termination of the client process. The following sections discuss these methods.

Initialization

The IOUserClient class defines the `initWithTask` method for the initialization of user-client instances. The default implementation simply calls the IOService `init` method, ignoring the parameters. The `initWithTask` method has four parameters:

- The Mach task of the client that opened the connection (type `task_t`)
- A security token to be passed to the `clientHasPrivilege` method when you are trying to determine whether the client is allowed to do secure operations (for which they need an effective UID of zero)
- A type to be passed to the `clientHasPrivilege` method when you are trying to determine whether the client is allowed to do secure operations (for which they need an effective UID of zero).
- Optionally, an OSDictionary containing properties specifying how the user client is to be created (currently unused)

The most significant of these parameters is the first, the user task. You probably should retain this reference as an instance variable so that you can easily handle connection-related activities related to the user process. Listing 4-7 (page 81) shows a simple implementation of `initWithTask`:

Listing 4-7 An implementation of `initWithTask`

```
bool
com_apple_dts_SimpleUserClient::initWithTask(task_t owningTask,
                                    void *security_id , UInt32 type)
{
    IOLog("SimpleUserClient::initWithTask()\n");

    if (!super::initWithTask(owningTask, security_id , type))
        return false;

    if (!owningTask)
    return false;
```

```
    fTask = owningTask;
    fProvider = NULL;
    fDead = false;

    return true;
}
```

Exclusive Device Access and the open Method

The user client must allow for device sharing or device exclusiveness, as required by the hardware. Many kinds of devices are designed to allow only one application at a time to access the device. For example, a device such as a scanner requires exclusive access. On the other hand, a device like a DSP PCI card (described in the scenario presented in"A Design Scenario" (page 74)) permits the sharing of its services among multiple application clients.

The ideal place for the user client to check and enforce exclusive access for devices is in the `open` method. At this point in the driver life cycle, the user client can ask its provider to open; if the provider's `open` method fails, that means another application is accessing the services of the provider. The user client refuses access to the requesting application by returning the appropriate result code, `kIOReturnExclusiveAccess`.

As with all commands, the application issues the initial command to open. It treats the open command just as it does any command issued by calling an `IOConnectMethod` function. The SimpleUserClient project defines `enum` constants for both the open and the complementary close commands that are used as indexes in the user client's `IOExternalMethod` array. Then the application (or device interface) calls one of the `IOConnectMethod` functions to issue the open command. If the result of the call is not `KERN_SUCCESS`, then the application or device interface knows that the device is being used by another user process. See "Open the User Client" (page 77) for more details.

For its part, the user-client subclass defines entries in the `IOExternalMethod` array for the open and close commands (Listing 4-8 (page 82)).

Listing 4-8 `IOExternalMethod` entries for the open and close commands

```
static const IOExternalMethod sMethods[kNumberOfMethods] =
    {
        {   // kMyUserClientOpen
            NULL,
            (IOMethod) &com_apple_dts_SimpleUserClient::open,
            kIOUCScalarIScalarO,
            0,
            0
        },
        {   // kMyUserClientClose
            NULL,
            (IOMethod) &com_apple_dts_SimpleUserClient::close,
            kIOUCScalarIScalarO,
            0,
            0
        },
      // ...
   );
```

In its implementation of the `getTargetAndMethodForIndex` method, when the user client receives an index of `kMyUserClientOpen`, it returns both a pointer to the `open` method and the target object on which to invoke this method (the user client itself). The implementation of the `open` method in SimpleUserClient looks like the code in Listing 4-9 (page 83).

Listing 4-9 Implementation of a user-client `open` method

```
IOReturn
com_apple_dts_SimpleUserClient::open(void)
{
    if (isInactive())
        return kIOReturnNotAttached;

    if (!fProvider->open(this))
        return kIOReturnExclusiveAccess;

    return kIOReturnSuccess;
}
```

This implementation first checks for a provider by invoking the IOService method `isInactive`. The `isInactive` method returns `true` if the provider has been terminated and is thus prevented from attaching; in this case, the user client should return `kIOReturnNotAttached`. Otherwise, if the user client has its provider attached, it can call `open` on it. If the `open` call fails, then the user client returns `kIOReturnExclusiveAccess`; otherwise it returns `kIOReturnSuccess`.

The `close` method in SimpleUserClient is similar to the `open` method, except that the implementation checks if the provider is open before calling `close` on it Listing 4-10 (page 83)).

Listing 4-10 Implementation of a user-client `close` method

```
IOReturn
com_apple_dts_SimpleUserClient::close(void)
{
    IOLog("SimpleUserClient::close()\n");

    if (!fProvider)
        return kIOReturnNotAttached;

    if (fProvider->isOpen(this))
        fProvider->close(this);

    return kIOReturnSuccess;
}
```

The user client's `close` method can be invoked for a number of reasons in addition to the user-space code issuing a close command. The user process could gracefully end the connection to the user client by calling the `IOServiceClose` function or the user process could die, in which case the `clientDied` method is invoked in the user client (see "Cleaning Up" (page 84)). If an unexpected event happens in a driver stack (for example, a device is removed), the user client receives a `didTerminate` message, to which it should respond by calling its `close` method. In its implementation of the `close` method, the user client should, after taking proper precautions, close its provider to unload it properly.

One consequence of this open-close design is that it's up to the user client's provider (in its `open` method) to determine when or whether another client is acceptable. For example, imagine that a DSP PCI card has a hardware limitation in that it can support only 256 clients. You could work around this limitation by multiplexing hardware access among the clients, but that would be a lot of work for an unlikely case. Instead you might choose to have the card's driver (the user client's provider) count the number of clients—incrementing the count in its `open` method and decrementing it in its `close` method—and return an error from `open` if the client limit has been exceeded.

The examples given above are greatly simplified and the exact implementation of `open` will depend on the nature of the application as well as the hardware. But the general open-close procedure as described here is highly recommended as it provides an enforced exclusive-access semantic that is usually appropriate for devices.

Cleaning Up

A user client cannot trust the user process that is its client. A user process can create and destroy a user client at any time. Moreover, there is no guarantee that the process will correctly close and release its user clients before quitting. The system tracks the user clients opened by each process and automatically closes and releases them if the process terminates, either gracefully or by crashing.

For these exigencies, the IOUserClient class has defined two methods, `clientClose` and `clientDied`. The `clientClose` method is called if the client process calls the `IOServiceClose` function. The `clientDied` method is called if the client process dies without calling `IOServiceClose`. The typical response of a user client in either case is to call `close` on its provider. Listing 4-11 (page 84) shows how the SimpleUserClient class does it.

Listing 4-11 Implementations of `clientClose` and `clientDied`

```
IOReturn
com_apple_dts_SimpleUserClient::clientClose(void)
{

    // release my hold on my parent (if I have one).
    close();
    terminate();

    if (fTask)
    fTask = NULL;

    fProvider = NULL;

    // DON'T call super::clientClose, which just returns notSupported

    return kIOReturnSuccess;
}

IOReturn
com_apple_dts_SimpleUserClient::clientDied(void)
{
    IOReturn ret = kIOReturnSuccess;

    IOLog("SimpleUserClient::clientDied()\n");

    // do any special clean up here
    ret = super::clientDied();

    return ret;
}
```

Passing Untyped Data Synchronously

The primary IOUserClient mechanism for passing data between a driver and an application is, at its core, an array of pointers to member functions implemented in the driver or, in some cases, the user client. The user client and the user process agree upon a set of constants that act as indexes into the array. When the user process makes a call to read or write some data, it passes this index along with some input and output parameters. Using the index, the user client finds the desired method and invokes it, passing along the required parameters.

That's the basic mechanism in a nutshell, but the description leaves out important details. These start with the fact that the data passed into or out of the kernel is essentially untyped. The kernel cannot know or predict data types in user space and so can only accept the most generalized types of data. The ensuing discussion describes how the I/O Kit's user-client API (on both sides of the kernel boundary) accommodates this restriction and how your subclass of IOUserClient must implement its part of the untyped data–passing mechanism. As you read along, refer to Figure 4-4 (page 89), which graphically shows the relationships among the pieces of the untyped data–passing API.

Scalar and Structure

The user-client mechanism uses only two generalized types for parameters: scalar and structure. These are further qualified by the direction of data: input or output.

Methods invoked in a driver object through the untyped-data mechanism must conform to the `IOMethod` type, which is deliberately elastic in terms of allowable parameters:

```
typedef IOReturn (IOService::*IOMethod)(void * p1, void * p2, void * p3,
                                        void * p4, void * p5, void * p6 );
```

However, the parameters of an `IOMethod` method must conform to one of four generalized prototypes identified by constants defined in `IOUserClient.h`:

`kIOUCScalarIScalarO`	Scalar input, scalar output
`kIOUCScalarIStructO`	Scalar input, structure output
`kIOUCStructIStructO`	Structure input, structure output
`kIOUCScalarIStructI`	Scalar input, structure input

On the user-process side of a connection, `IOKitLib.h` defines four functions corresponding to these constants:

```
IOConnectMethodScalarIScalarO
IOConnectMethodScalarIStructureO
IOConnectMethodStructureIStructureO
IOConnectMethodScalarIStructureI
```

For further information, see section "Send and Receive Data" (page 78) which discusses how code in user space uses these functions.

Including the Header File of Common Types

Make sure your user-client subclass includes the header file that you have created to define data types common to both kernel and user-space code. The types would include the `enum` constants to use as indexes into the `IOExternalMethod` array and any structures involved in I/O.

For more on these common type definitions, see "Defining Common Types" (page 75).

Constructing the IOExternalMethod Array

The distinctive action that a subclass of IOUserClient must perform when implementing its part of the untyped-data mechanism is identifying the driver method that the user process wants invoked at a particular moment. An important part of this task is the construction of the array of pointers to the methods to invoke. However, the contents of this array are actually more than a simple table of method pointers; each element of the array is an `IOExternalMethod` structure, of which only one member is a method pointer.

The other members of the `IOExternalMethod` structure designate the object implementing the method to invoke (the target), identify the general types of the parameters (scalar or structure, input or output), and provide some information about the parameters. Table 4-3 (page 86) describes the `IOExternalMethod` fields.

Table 4-3 Fields of the `IOExternalMethod` structure

Field (with type)	Description
`IOService * object`	The driver object implementing the method, usually the user client's provider (the "target"). Can be `NULL` if the target is to be dynamically determined at run time.
`IOMethod func`	A pointer to the method to invoke in the target; include the class name (for example, "com_acme_driver_MyDriver::myMethod")
`IOOptionBits flags`	One of the `enum` constants defined in `IOUserClient.h` for specifying general parameter types
`IOByteCount count0`	If first parameter designates scalar, the number of scalar values; if first parameter designates structure, the size of the structure
`IOByteCount count1`	If second parameter designates scalar, the number of scalar values; if second parameter designates structure, the size of the structure

You can initialize the `IOExternalMethod` array in any of the likely places in your code:

- In static scope
- In the `start` method
- In your implementation of the IOUserClient `getTargetAndMethodForIndex` method

It is in this last method that the I/O Kit requests the `IOExternalMethod` structure to use. Listing 4-12 (page 87) shows how the SimpleUserClient example project initializes the array.

Listing 4-12 Initializing the array of `IOExternalMethod` structures

```
static const IOExternalMethod sMethods[kNumberOfMethods] =
{
    {   // kMyUserClientOpen
        NULL,                   // Target determined at runtime.
        (IOMethod) &com_apple_dts_SimpleUserClient::open,
        kIOUCScalarIScalarO,    // Scalar Input, Scalar Output.
        0,                      // No scalar input values.
        0                       // No scalar output values.
    },
    {   // kMyUserClientClose
        NULL,                   // Target determined at runtime.
        (IOMethod) &com_apple_dts_SimpleUserClient::close,
        kIOUCScalarIScalarO,    // Scalar Input, Scalar Output.
        0,                      // No scalar input values.
        0                       // No scalar output values.
    },
    {   // kMyScalarIStructImethod
        NULL,                   // Target determined at runtime.
        (IOMethod) &com_apple_dts_SimpleDriver::method1,
        kIOUCScalarIStructI,    // Scalar Input, Struct Input.
        1,                      // One scalar input value.
        sizeof(MySampleStruct)  // The size of the input struct.
    },
    {   // kMyScalarIStructOmethod
        NULL,                   // Target determined at runtime.
        (IOMethod) &com_apple_dts_SimpleDriver::method2,
        kIOUCScalarIStructO,    // Scalar Input, Struct Output.
        2,                      // Two scalar input values.
        sizeof(MySampleStruct)  // The size of the output struct.
    },
    {   // kMyScalarIScalarOmethod
        NULL,                   // Target determined at runtime.
        (IOMethod) &com_apple_dts_SimpleDriver::method3,
        kIOUCScalarIScalarO,    // Scalar Input, Scalar Output.
        2,                      // Two scalar input values.
        1                       // One scalar output value.
    },
    {   // kMyStructIStructOmethod
        NULL,                   // Target determined at runtime.
        (IOMethod) &com_apple_dts_SimpleDriver::method4,
        kIOUCStructIStructO,    // Struct Input, Struct Output.
        sizeof(MySampleStruct), // The size of the input struct.
        sizeof(MySampleStruct)  // The size of the output struct.
    },
};
```

Implementing getTargetAndMethodForIndex

All subclasses of IOUserClient that use the untyped-data mechanism for data transfer between application and driver must implement the `getTargetAndMethodForIndex` method (or, for asynchronous delivery, `getAsyncTargetAndMethodForIndex`). A typical implementation of `getTargetAndMethodForIndex` has to do two things:

- Return directly a pointer to the appropriate `IOExternalMethod` structure identifying the method to invoke.

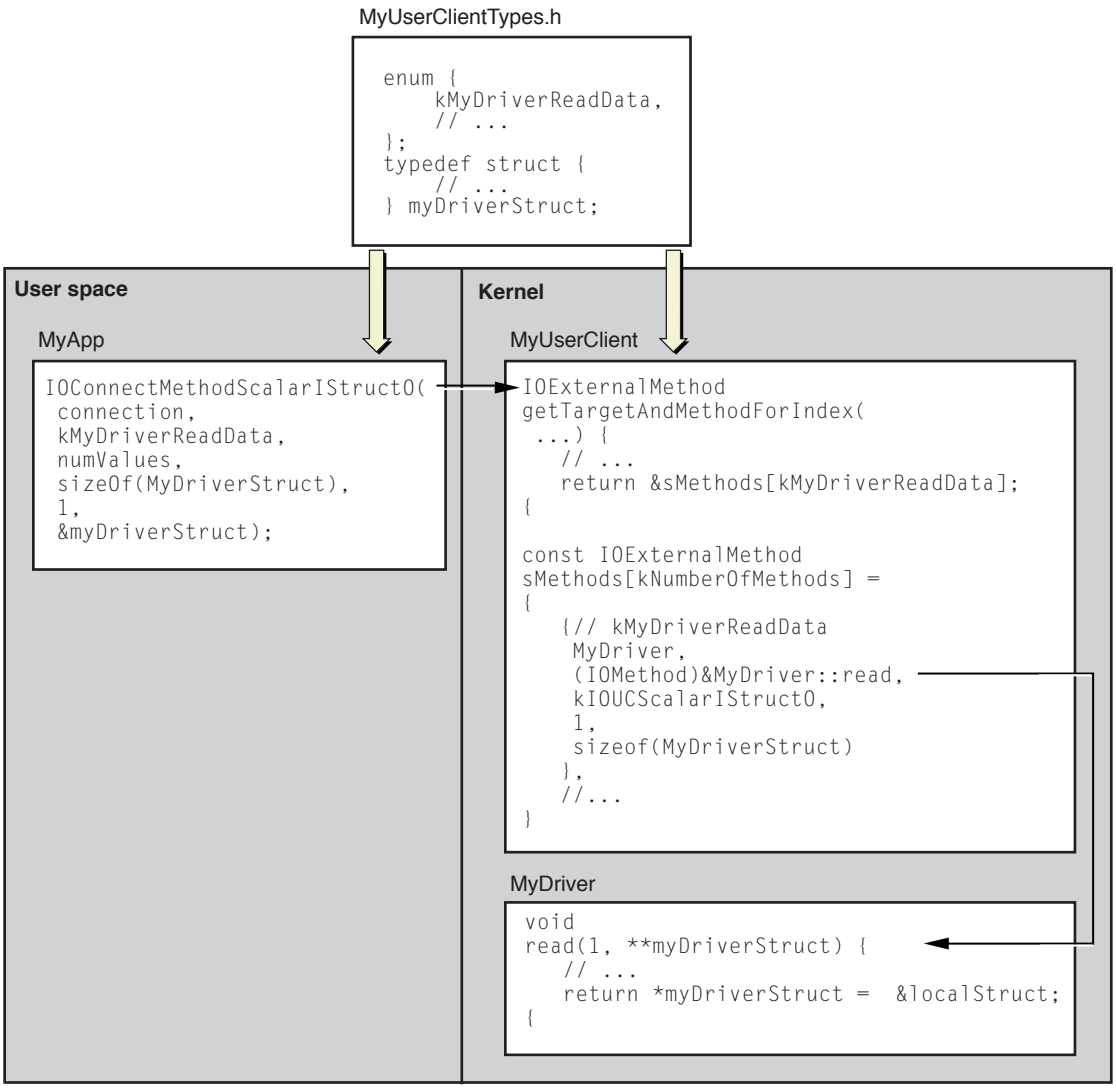
- Return a reference to the object that implements the method (the target).

You can either statically assign the target to the `IOExternalMethod` field when you initialize the structure, or you can dynamically determine the target at run time. Because the target is usually the user client's provider, often all you need to do is return your reference to your provider (assuming you've stored it as an instance variable). Listing 4-13 (page 88) shows one approach for doing this.

Listing 4-13 Implementing the `getTargetAndMethodForIndex` method

```
IOExternalMethod *
com_apple_dts_SimpleUserClient::getTargetAndMethodForIndex(IOService **
                                                target, UInt32 index)
{

     // Make sure IOExternalMethod method table has been constructed
    // and that the index of the method to call exists

    if (index < (UInt32)kNumberOfMethods)
    {
        if (index == kMyUserClientOpen || index == kMyUserClientClose)
        {
            // These methods exist in SimpleUserClient
            *target = this;
        }
        else
        {
            // These methods exist in SimpleDriver
            *target = fProvider;
        }

        return (IOExternalMethod *) &sMethods[index];
    }

    return NULL;
};
```

Figure 4-4 The essential code for passing untyped data



Validation

A user client should thoroughly validate all data that it handles. It shouldn't just blindly trust its client, the user process; the client could be malicious. Some of the validation checks might be internal consistency among input and output commands and buffers, spurious or poorly defined commands, and erroneous register bits. For long and complicated code, you might want to create a configurable validation engine.

Passing Untyped Data Asynchronously

The defining characteristic of the procedure described in "Passing Untyped Data Synchronously" (page 85) is the behavior of the user-space thread making an I/O request. When the client in user space requests an I/O transfer by calling one of the `IOConnectMethod` functions, the thread bearing the request must block and wait for the user client to return when the I/O completes. This behavior is synchronous. However, the IOUserClient class also provides APIs for asynchronous data transfer between user process and user client.

With these APIs, when the user process calls an `IOConnectMethod` function, it can go on immediately to other tasks because it is not blocked in the function. Later, the user client invokes a callback in the client application, passing it the result and any resulting data.

Although you can dedicate a separate application thread for notifications of I/O completions (as described in "Synchronous Versus Asynchronous Data Transfer" (page 69)), a notification thread is not necessary for asynchronous I/O. In fact, there is an alternative to a notification thread that is much easier to implement.

You can accomplish the same asynchronous behavior using the application's run loop (CFRunLoop). Each application's main thread has a run-loop object that has receive rights on a Mach port set on which all event sources pertinent to the application (mouse events, display events, user-space notifications, and so on) have send rights. Running in a tight loop, the CFRunLoop checks if any of the event sources in its Mach port set have pending events and, if they do, dispatches the event to the intended destination.

Because run loops are so ubiquitous in user space—every application has one—they offer an easy solution to the problem of posting completions of I/O from the kernel to user space. An I/O notification source for the user client just needs to be added to the run loop. Then the application (or device interface) just needs to pass this port to the user client as well as a pointer to a completion routine for the user client to invoke when the I/O completes.

This section describes the general procedures that the user client and the application should follow to implement asynchronous I/O using CFRunLoop and *some* of the APIs in the I/O Kit framework and the IOUserClient class. (Many of the APIs in the IOUserClient class that are tagged with "async" are not essential for implementing asynchronous I/O.) Although this section illustrates the procedure with only one CFRunLoop (associated with the application's main thread) and one run-loop source, it is possible to have multiple run loops and multiple run-loop sources.

Application Procedure Using CFRunLoop

To implement its part of asynchronous untyped data-passing, the application or device interface must first obtain receive rights to a port on the application's CFRunLoop port set, thereby becoming a run-loop source for that run loop. Then it passes this Mach port to its user client. It must also implement a callback function that the user client calls when an I/O completes.

The following procedure itemizes the steps that the application must complete to accomplish this; all APIs mentioned here are defined in `IOKitLib.h`, except for the CFRunLoop APIs which are defined in `CFRunLoop.h` in the Core Foundation framework:

1. Implement a function that conforms to the callback prototype `IOAsyncCallback` (or one of the related callback types; see `IOKitLib.h`).The initial parameter of this function (`void *refcon`) is an identifier of the I/O request; subsequent parameters are for result and return data (if any). Your implementation of the `IOAsyncCallback` routine should properly handle the result and returned data for each particular I/O request.

2. Create a notification-port object (which is based on a Mach port).

   Call `IONotificationPortCreate`, passing in the master port obtained from the earlier call to `IOMasterPort`. This call returns an `IONotificationPortRef` object.

3. Obtain a run-loop source object from the notification-port object.

   Call the `IONotificationPortGetRunLoopSource` function, passing in the `IONotificationPortRef` object. This call returns a `CFRunLoopSourceRef` object.

4. Register the run-loop source with the application's run loop.

Call `CFRunLoopAddSource`, specifying as parameters a reference to the application's CFRunLoop, the `CFRunLoopSourceRef` object obtained in the previous step, and a run-loop mode of `kCFRunLoopDefaultMode`.

5. Get the Mach port backing the run-loop source for the I/O completion notifications.

   Call the `IONotificationPortGetMachPort` function, passing in the `IONotificationPortRef` object again. This call returns a Mach port typed as `mach_port_t`.

6. Give the Mach notification port to the user client.

   Call `IOConnectSetNotificationPort`, passing in the port and the connection to the user client; the two remaining parameters, type and reference, are defined per I/O Kit family and thus not needed in your case. Calling `IOConnectSetNotificationPort` results in the invocation of `registerNotificationPort` in the user client.

7. When your application is ready for an I/O transfer, issue an I/O request by calling one of the `IOConnectMethod` functions. The parameter block (such as a task file) containing the I/O request should include as its first two fields a pointer to the `IOAsyncCallback` callback routine implemented in the application or device interface. It should also include a `(void *)` refcon field to provide context for the request. Otherwise, the procedure is exactly the same as for synchronous untyped data-passing except that the `IOConnectMethod` function returns immediately.

8. When there are no more I/O transfers to make, the application or device interface should dispose of the port and remove the run-loop source it has created and registered. This involves completing the following steps:

   a. Remove the run-loop source (`CFRunLoopSourceRef`) from the run loop by calling `CFRunLoopRemoveSource`.

   b. Destroy the notification-port object by calling `IONotificationPortDestroy`. Note that this call also releases the run-loop source you obtained from the call to `IONotificationPortGetRunLoopSource` in Step 3.

User Client Procedure

The procedure the IOUserClient subclass must follow using the asynchronous I/O APIs and CFRunLoop is similar to the synchronous approach described in "Passing Untyped Data Synchronously" (page 85) in that some of the same APIs are used. However, the asynchronous I/O procedure (using the application's run loop) differs in many significant details from the synchronous approach.

1. Construct the `IOExternalMethod` array and implement the `getTargetAndMethodForIndex` method as you would in the synchronous approach.

2. Implement the IOUserClient method `registerNotificationPort` to retain a reference to the Mach notification port that is passed in. Recall that this method is invoked as a result of the `IOConnectSetNotificationPort` call in user space, and the notification port is a run-loop source of the application's CFRunLoop object.

3. In your implementation of the `IOMethod` method that is invoked, do the following:

   a. Check the `IOAsyncCallback` pointer field in the parameter block. If it is non-`NULL`, then you know this is an asynchronous I/O request. (If it is a `NULL` pointer, then process the request synchronously.)

b. Call the IOUserClient `setAsyncReference` method, passing in an empty `OSAsyncReference` array, the notification port on the application's CFRunLoop, the pointer to the application's callback routine, and the pointer to the refcon information. The `setAsyncReference` method initializes the `OSAsyncReference` array with the following constants (in this order): `kIOAsyncReservedIndex`, `kIOAsyncCalloutFuncIndex`, and `kIOAsyncCalloutRefconIndex`. (These types are defined in the header file `OSMessageNotification.h`.)

c. Send off the I/O request for processing by lower objects in the driver stack and return.

4. When the I/O operation completes on a driver work loop, the driver notifies the user client and gives it the result of the I/O operation and any resulting output. To notify the application, the user client calls `sendAsyncResult`, passing in the result and data from the I/O operation. This call results in the invocation of the `IOAsyncCallback` callback routine in the user-space code.

> **Note:** The first parameter of the `sendAsyncResult` method is the `OSAsyncReference` array. The last five slots in this array are reserved for the user client to pass data back to the user process.

5. When there are no more I/O transfers, the user client should, in its `close` method, tear down its part of the asynchronous I/O infrastructure by calling `mach_port_deallocate` on the notification port.

Mapping Device Registers and RAM Into User Space

If your driver has hardware with full PIO memory management, your user client may map the hardware registers into the address space of the user process. In this case, the user process does not require access to physical addresses. However, if the PIO hardware does require the use of interrupts, you will have to factor this requirement into your code. It is always possible, even with DMA hardware, to publish device RAM to user space.

The user process initiates a request for mapped memory by calling `IOConnectMapMemory`, passing in, among other parameters, a pointer to (what will become) the mapped memory in its own address space. The `IOConnectMapMemory` call results in the invocation of the `clientMemoryForType` method in the user client. In its implementation of this method, the user client returns the IOMemoryDescriptor object it created in its `open` or `start` method that backs the mapping to the hardware registers. The user process receives back this mapping in terms of virtual-memory types of `vm_address_t` and `vm_size_t`. It is now free to read from and write to the hardware registers.

Instead of creating an IOMemoryDescriptor object, a user client can (in most cases) simply get the IODeviceMemory object created by its provider's nub. Then, in its `clientMemoryForType` method, it returns a pointer to the IODeviceMemory object. (IODeviceMemory is a subclass of IOMemoryDescriptor.) If it does this, it should ensure the no-caching flag is turned on (by OR-ing the appropriate flag in the `IOOptionBits` parameter). The nub of the providing driver uses the IODeviceMemory object to map the registers of the PCI device's physical address space to the kernel's virtual address space. Through this object, it can get at the kernel's address space, which most drivers do when they want to talk to hardware registers. Returning the provider's IODeviceMemory object in the `clientMemoryForType` method is also how you publish (PCI) device RAM out to user space.

Make sure you retain the IODeviceMemory or any other IOMemoryDescriptor object in your implementation of `clientMemoryForType` so a reference to it can be returned to the caller.

If the user client creates an IOMemoryDescriptor object in its `open` method, it should release the object in its `close` method; if the user client creates an IOMemoryDescriptor object in its `start` method, it should release the object in its `stop` method. If the user client passes on an IODeviceMemory object created by its provider, it should not release it at all (the provider should release it appropriately). At the close of I/O, the device interface or application should call `IOConnectUnmapMemory`.

For example implementations of `IOConnectMapMemory` and `clientMemoryForType`, see Listing 4-18 (page 96) and Listing 4-24 (page 101), respectively.

A Guided Tour Through a User Client

Double X Technologies makes a number of products. (This is a fictional company, so don't go searching for it on the Internet.) One product is a video-capture board. This hardware uses a DMA engine for data transfer at high rates and does not permit device sharing. The company has written a Mac OS X (Darwin) driver for it, and now wants to make device and driver accessible to as many user-space clients as possible. So they decide they need to write a user client and complementary user-space library—a device interface—for it.

The Double X engineers decide upon a design that is structured around register task files, and that uses a combination of synchronous untyped-data passing and shared memory. This section takes you on a guided tour through their design, illustrating the salient parts with code examples. The tour has three major stops corresponding to the three major parts of the design:

- Definitions of types common to both the kernel and user-space code
- The user-space library that functions as the device interface
- The subclass of IOUserClient

Common Type Definitions

The project for the Double X Capture driver also includes the user-client header and source files. One of these header files contains the type definitions that are used by both the user client and the device interface. This file contains the `enum` constants used as indexes into the array of `IOExternalMethod` structures maintained by the user client. It also contains structures for the various types of task files, operation codes, and macro initializers for the task files.

Listing 4-14 (page 93) shows the definition of the `enum` constants used as method-array indexes.

Listing 4-14 The indexes into the `IOExternalMethod` array

```
typedef enum XXCaptureUCMethods {
    kXXCaptureUserClientActivate        // kIOUCScalarIScalar0,  3,  0
    kXXCaptureUserClientDeactivate      // kIOUCScalarIScalar0,  0,  0
    kXXCaptureUserClientExecuteCommand, // kIOUCStructIStruct0, -1, -1
    kXXCaptureUserClientAbortCommand,   // kIOUCScalarIScalar0,  1,  0
    kXXCaptureLastUserClientMethod,
} XXCaptureUCMethods;
```

As you can see, the methods invoked by the user client allocate (activate) and deallocate (deactivate) kernel and hardware resources, and execute and abort commands. The activate command also results in the invocation of the user client's `open` method (for enforcing exclusive access) and the deactivate command causes the invocation of `close`. This walk-through focuses on the execute command (`kXXCaptureUserClientExecuteCommand`).

Each task file begins with an "op code" (or operation code) that indicates the type of task to perform with the task file. The Double X Capture user client defines about a dozen op codes as `enum` constants (see Listing 4-15 (page 94)).

Listing 4-15 Operation codes for task files

```
typedef enum XXCaptureOpCodes {
    kXXGetAddress = 0,                      //  0
    kXXGetRegister,                         //  1
    kXXSetAddress,                          //  2
    kXXSetRegister,                         //  3
    kXXClearRegister,                       //  4
    kXXSetToRegister,                       //  5

    // DMA control commands
    kXXDMATransferBlock,                    //  6
    kXXDMATransferNonBlock,                 //  7
    kXXLastOpCode
} XXCaptureOpCodes;
```

The op codes cover a range of register-specific actions, from getting an address or register value to setting an address or register. There are also two op codes for two kinds of DMA transfers: blocking and non-blocking. This section focuses on task files with two op codes: atomic set register to value (`kXXSetToRegister`) and blocking DMA transfer (`kXXDMATransferBlock`).

The Double X user-client code defines a task file as a structure of type `XXUCCommandData`. The first few fields of this structure are reserved for status and configuration information; the last field (`fCmds`) is an array of `XXCaptureCommandBlock` structures. Listing 4-16 (page 94) shows the definitions of these structures.

Listing 4-16 Task-file structures used in some operations

```
typedef struct XXUCCommandData {
    IOReturn fErrorCode;    // Out:What type of error
    UInt32 fNumCmds;        // In:Number of commands, Out:Cmd in error
    XXCaptureCommandBlock fCmds[0];
} XXUCCommandData;

typedef struct XXCaptureCommandBlock {
    UInt16 fOp;
    UInt16 fReg;
    UInt32 fReserved[3];
} XXCaptureCommandBlock;

typedef struct XXCaptureRegisterToValue {
    UInt16 fOp;
    UInt16 fReg;            // In, register address in BAR
    UInt32 fValue;          // In|Out, value of bits to be set
    UInt32 fMask;           // In|Out, mask of bits to be set
    UInt32 fReserved2;      // 0, Do not use
} XXCaptureRegisterToValue;
```

```
typedef struct XXCaptureDMATransfer {
    UInt16 fOp;
    UInt16 fDirection;      // Direction of transfer
    UInt8 *fUserBuf;
    UInt8 fDevOffset;
    UInt32 fLength;         // In, length in bytes to transfer
} XXCaptureDMATransfer;
```

Also shown in this example are two types of structures specific to the kinds of commands we are tracing in this tour: Set register value and contiguous blocking DMA transfer. The structure types are, respectively, `XXCaptureRegisterToValue` and `XXCaptureDMATransfer`. When the device interface builds a task file containing these command structures, it puts them in the `XXCaptureCommandBlock` array even though they are not of that type. Note that `XXCaptureCommandBlock` ends with some padding (`fReserved`), guaranteeing it to be at least the same size as any other command-block structure. You just need to do the appropriate casting on an item in the `XXCaptureCommandBlock` array to get a structure of the correct type.

The common header file also defines macros that initialize commands and put them into a task file. Listing 4-17 (page 95) shows the macro initializers for the `XXCaptureRegisterValue` and `XXCaptureDMATransfer` command structures.

Listing 4-17 Macro initializers for some task files

```
#define cmdGetRegister(cmd, reg) do {                                   \
    XXCaptureCommandBlock *c = (XXCaptureCommandBlock *) (cmd);     \
    c->fOp   = kXXCaptureGetRegister;                                       \
    c->fReg  = (reg);                                                   \
    cmd++;                                                              \
} while (0)

#define cmdSetToRegister(cmd, reg, val) do {                            \
    XXCaptureRegisterValue *c = (XXCaptureRegisterValue *) (cmd);       \
    c->fOp    = kXXSetToRegister;                                       \
    c->fReg   = (reg);                                                  \
    c->fValue = (val);                                                  \
    c->fMask  = (mask);                                                 \
    cmd++;                                                              \
} while (0)

#define cmdDMA(cmd, op, dir, u, d, len) do {                            \
    XXCaptureDMATransfer *c = (XXCaptureDMATransfer *) (cmd);           \
    c->fOp      = (op);                                                 \
    c->fDirection = (dir);                                              \
    c->fUserBuf = (u);                                                  \
    c->fDevOffset = (d);                                                \
    c->fLength  = (len);                                                \
    cmd++;                                                              \
} while (0)

#define cmdDMABlock(cmd, dir, u, d, len)                                \
        cmdDMA(cmd, kXXDMATransferBlock, dir, u, d, len)#define
cmdDMANonBlock(cmd, dir, u, d, len)                             \
        cmdDMA(cmd, kXXDMATransferNonBlock, dir, u, d, len)
```

The `#define` preprocessor statement was used to construct these macros because it enables the code to increment the index automatically to the next command in the `XXCaptureCommandBlock` array.

The Device Interface Library

The library implemented by Double X to function as the device interface for its video-capture hardware presents a functional programmatic interface to applications. (Application developers shouldn't have to know low-level details of the hardware.) When it receives a functional request (equivalent to something like "write *n* blocks of data"), the device interface breaks it down into the required hardware-register commands, puts these commands in a task file, and sends this task file to the user client for execution.

First, the application requests the Double X device interface to establish a connection to the user client. The device interface defines function `XXDeviceOpen` (not shown) for this purpose. This function completes the steps described earlier in the following sections:

1. "Get the I/O Kit Master Port" (page 75)
2. "Obtain an Instance of the Driver" (page 75)
3. "Create a Connection" (page 76)

Even though the device interface now has a connection to the user client, I/O transfers cannot yet occur. The user client must first open its provider and prepare for I/O by allocating the necessary kernel and hardware resources. The `XXDeviceConnect` function, which the application calls after it has opened a connection, is defined for this purpose (see Listing 4-18 (page 96)).

Listing 4-18 Invoking the user client's activate method, mapping card memory

```
// knXXCaptureUserClientActivate, kIOUCScalarIScalarO,  3,  0
kern_return_t
XXDeviceConnect(XXDeviceHandle handle,
                Boolean fieldMode, UInt32 storeSize)
{
    XXDeviceDataRef device = (XXDeviceDataRef) handle;
    kern_return_t ret;

    ret = IOConnectMethodScalarIScalarO(device->fUCHandle,
            kXXCaptureUserClientActivate, 3, 0,
            (int) &device->fStatus, (int) fieldMode,
            storeSize);
    if (KERN_SUCCESS != ret)
        goto bail;

    ret = IOConnectMapMemory(device->fUCHandle,
                        kXXCaptureUserClientCardRamO,
                        mach_task_self(),
                        (vm_address_t *) &device->fCardRAM,
                        &device->fCardSize,
                        kIOMapAnywhere);

bail:
    return ret;
}
```

The call to `IOConnectMethodScalarIScalarO` in this example invokes the user client's `activate` method. This method opens the provider, gets the provider's DMA engine, adds a filter interrupt source to the work loop and maps a status block into shared memory.

The `XXDeviceConnect` function also maps the capture card's physical memory into the address space of the user process. It stores the returned addressing information in the `XXDeviceHandle` "global" structure. This step is necessary before any DMA transfers can take place.

For actual I/O transfers, the Double X device interface defines a number of functions that present themselves with names and signatures whose significance an application developer can easily grasp. This functional I/O interface might consist of functions with names such as `XXCaptureWriteData`, `XXCaptureReadData`, and (for hardware that understands blocks) `XXCaptureWriteBlocks`. Listing 4-19 (page 97) shows how the device interface implements the `XXCaptureWriteBlocks` function.

Listing 4-19 The device interface's function for writing blocks of data

```
#define writeCmdSize \
    (sizeof(XXUCCommandData) + kMaxCommands* sizeof(XXCaptureCommandBlock))

int XXCaptureWriteBlocks(XXDeviceHandle device, int blockNo, void *addr,
                            int *numBlocks)
{
    int res;
    int length;
    UInt8 buffer[writeCmdSize];
    XXCaptureCommandBlock *cmd, *getCmd;
    XXUCCommandData *cmds = (XXUCCommandData *) buffer;

    cmd = &cmds->fCmds[0];

    cmdSetToRegister(cmd, kXXCaptureControlReg, -1, kXXCaptureDMAResetMask);
    cmdDMABlock(cmd, kIODirectionOut, adddr, blockNo, *numBlocks);
    getCmd = cmd;
    cmdGetRegister(cmd, kXXCaptureDMATransferred);

    length = (UInt8 *) cmd - (UInt8 *) cmds;
    res = (int) IOConnectMethodStructureIStructureO(device->fUCHandle,
                kXXCaptureUserClientExecuteCommand,
                length, length, cmds, cmds);

    if (res)
        goto bail;

    res = cmds->fErrorCode;
    *numBlocks = getCmd->fValue;

bail:
    return res;
}
```

This function basically takes a request to write a block of data (whose user-space address and size is supplied in the parameters) and constructs a task file composed of three register-based commands:

- The `cmdSetToRegister` macro constructs a command that tells the hardware to reset the DMA engine.
- The `cmdDMABlock` macro puts together a command that programs the DMA engine for the I/O transfer using the passed-in parameters and specifying the direction of the transfer (`kIODirectionOut`).
- The `cmdGetRegister` macro creates a command that reads a register containing the number of blocks that were transferred.

The `XXCaptureWriteBlocks` function next calls `IOConnectMethodStructureIStructureO`, passing in a pointer to the `XXCaptureCommandBlock` task file just created. This function call results in the invocation of the user client's `execute` method (see "The User Client" (page 98)). Finally, the function determines if there was an error in the execution of the task file; if there was, it extracts the information from the output `XXCaptureCommandBlock` structure (`getCmd`) and returns this information to the calling application.

The User Client

Think back to the `IOConnectMethodStructureIStructureO` call in the device interface's `XXCaptureWriteBlocks` function.Through the magic of Mach messaging and the I/O Kit, that call comes out on the kernel side as an invocation of the user client's `getTargetAndMethodForIndex` method with an index argument of `kXXCaptureUserClientExecuteCommand`. The IOUserClient subclass for the Double X device implements the `getTargetAndMethodForIndex` to handle this as shown inListing 4-20 (page 98).

Listing 4-20 Returning a pointer to an `IOExternalMethod` structure

```
#define kAny ((IOByteCount) -1 )

static IOExternalMethod sXXCaptureUserClientMethods[] =
{
    // kXXCaptureUserClientActivate,
    { 0, Method(activate),   kIOUCScalarIScalarO,    3,    0 },
    // kXXCaptureUserClientDeactivate,
    { 0, Method(deactivate), kIOUCScalarIScalarO,    0,    0 },
    // kXXCaptureUserClientExecuteCommand,
    { 0, Method(execute),    kIOUCStructIStructO, kAny, kAny },
    // kXXCaptureUserClientAbortCommand,
    { 0, Method(abort),      kIOUCScalarIScalarO,    1,    0 },
};

IOExternalMethod *XXCaptureUserClient::
getTargetAndMethodForIndex(IOService **targetP, UInt32 index)
{
    IOExternalMethod *method = 0;

    if (index < kXXCaptureLastUserClientMethod)
    {
        *targetP = this;
        method = &sXXCaptureUserClientMethods[index];
    }

    return method;
}
```

As a result of the returned index and target, the I/O Kit invokes the `execute` method in the user client, passing in pointers to the task-file buffers (`vInCmd` and `vOutCmd`). As you can see from Listing 4-21 (page 98), the `execute` method begins by declaring local variables and assigning the parameters to them.

Listing 4-21 The `execute` method—preparing the data

```
// kXXCaptureUserClientExecuteCommand, kIOUCStructIStructO, -1, -1
IOReturn XXCaptureUserClient::
execute(void *vInCmd,  void *vOutCmd,
```

```
        void *vInSize, void *vOutSizeP, void *, void *)
{
    XXCaptureCommandBlock *cmd;
    UInt32 index, numCmds, cmdSize;
    bool active;

    XXUCCommandData *inCmdBuf = (XXUCCommandData *) vInCmd;
    XXUCCommandData *outCmdBuf = (XXUCCommandData *) vOutCmd;
    UInt32 *outSizeP = (UInt32 *) vOutSizeP;
    UInt32  inSize = (UInt32) vInSize, outSize = *outSizeP;
    IOReturn ret = kIOReturnInternalError;
    UInt32 numOutCmd;
```

Before it does any I/O work, the `execute` method performs a series of validation checks on the parameters. It validates the sizes of input and output command blocks and ensures that they are internally consistent. It also checks for command blocks with spurious data, such as registers whose bits cannot be set. Listing 4-22 (page 99) illustrates how the user client performs one such check on the `kXXSetToRegister` command.

Listing 4-22 A validation check on `kXXSetToRegister` command

```
        case kXXSetToRegister: {
            XXCaptureRegisterToValue *regCmd;
            regCmd = (XXCaptureRegisterToValue *) cmd;

            if ( !isValidBitsForRegister(cmd->fReg, regCmd->fMask)
            {
                DebugLog(("%s(%x)::execute() "
                          "can't set bit %x for reg(%d)\n",
                          getName(), (int) this,
                          (int) regCmd->fValue, (int) cmd->fReg));
                ret = kIOReturnBadArgument;
                goto bail;
            }
            break;
        }
```

This section of code validates the register operation, determining if the register is valid and whether it's permitted to modify the intended bits.

After completing the validation phase, the user client carries out the I/O request. Listing 4-23 (page 99) shows how the `execute` method handles the `kXXSetToRegister` and `kXXContTransferBlock` commands.

Listing 4-23 The `execute` method—executing the I/O command

```
    cmd = inCmdBuf->fCmds;
    for (index = 0; index < numCmds; index += cmdSize, cmd += cmdSize)
    {
        cmdSize = 1;            // Setup default command size

        switch (cmd->fOp)
        case kXXSetToRegister: {
            XXCaptureRegisterToValue *regCmd =
                (XXCaptureRegisterToValue *) &inCmdBuf->fCmds[index];
            fNub->toValueBitAtomic(cmd->fReg, regCmd->fValue, regCmd->fMask);
            break;
        }
        case kXXGetRegister: {
```

```
            XXCaptureRegisterValue *out =
                (XXCaptureRegisterValue *) &outCmdBuf->fCmds[index];

            out->fValue = fNub->getReg(regP);
            break;
        }
        case kXXDMATransferBlock:
        case kXXDMATransferNonBlock: {

            XXCaptureDMATransfer *c =
                (XXCaptureDMATransfer *) &inCmdBuf->fCmds[index];

            DMARequest *req;
            bool blocking = c->fOp == kXXDMATransferBlock;

            req = fProvider->createDMARequest(
                        (vm_address_t) c->fUserBuf,
                        fClient,
                        c->fDevOffset,
                        c->fDirection,
                        c->fLength,
                        this,
                        (XXDMAEngine::Completion)
                            &XXCaptureUserClient::dmaCompleteGated,
                        (void *) blocking);
            if (!req) {
                ret = kIOReturnError;
                goto bail;
            }
            ret = fGate->runAction(gatedFunc(runDMAGated),
                (void *) req, (void *) blocking);
            break;
        }

            // other code here ...
        }
bail:
    outCmdBuf->fErrorCode = ret;
    outCmdBuf->fNumCmds = index;

    return kIOReturnSuccess;
}
```

If the command is "set register to value" (`kXXSetToRegister`), `execute` calls a method implemented by its provider called `toValueBitAtomic`. This method sets the specified register to the specified value in an atomic manner.

If the command is "get register value" (`kXXGetRegister`), `execute` calls its provider to get the value of the specified register (in this case, holding the number of blocks transferred). The code assigns this value to the appropriate field (`fValue`) of the `kXXGetRegister` command.

If the command is "program DMA engine for contiguous blocking I/O transfer" (`kXXDMATransferBlock`), the `execute` method programs the DMA engine by completing the following steps:

1. It creates a DMA request by calling the `createDMARequest` method, which is implemented by an object representing the DMA engine.

2. It programs the DMA engine by running the `runDMAGated` method in the command gate.

3. It puts the result of the I/O operation and the number of blocks transferred in the appropriate fields of the output task file; if an error has occurred, the `fNumCmds` field of this structure (`XXUCCommandData`) holds the index of the command causing the error.

Before leaving this example, let's look at the user client's role in preparing shared memory for the DMA transfers. Recall how the device interface, in its `XXDeviceConnect` function, called `IOConnectMapMemory` to map the hardware registers of the video-capture board into its address space. The `IOConnectMapMemory` call causes the invocation of the `clientMemoryType` method in the user client. Listing 4-24 (page 101) shows how the Double X user client implements this method to return an IOMemoryDescriptor.

Listing 4-24 Implementation of `clientMemoryForType`

```
IOReturn XXCaptureUserClient::
clientMemoryForType(UInt32 type, UInt32 *flags,
                    IOMemoryDescriptor **memory)
{
    IOMemoryDescriptor *mem;
    IOReturn ret;

    switch(type)
    {
    case kXXCaptureUserClientCardRam0:
        mem = fNub->getDeviceRAM();
        break;

    default:
        ret = kIOReturnUnsupported;
        goto bail;
    }

    if (!mem)
    {
        ret = kIOReturnNoMemory;
        goto bail;
    }

    mem->retain();
    *memory = mem;
    ret = kIOReturnSuccess;

bail:

    return ret;
}
```

In this method, the user client simply gets the memory descriptor of its provider if the type of the memory requested is `kXXCaptureUserClientCardRam0`. This memory-descriptor object is an instance of the IODeviceMemory class, which inherits from IOMemoryDescriptor. It represents the registers of a PCI device (the video-capture card, in this case) and, more particularly, the mapping of those physical addresses into the kernel's virtual address space. Before returning the IODeviceMemory object, this implementation retains it. When the user client returns this IODeviceMemory object (by indirection), the I/O Kit converts it to the application's address space in terms of `vm_address_t` memory pointer and length.

Kernel-User Notification

Important: The Kernel-User Notification Center APIs are not available to KEXTs that declare dependencies on the sustainable kernel programming interfaces (KPIs) introduced in Mac OS X v10.4. However, if your driver currently uses the KUNC APIs and you have not revised it to depend on KPIs, your driver will still work. See Kernel Extension Dependencies for more information on KPIs and which KEXTs should declare dependencies on them.

Once in a while a driver—or for that matter any type of kernel extension—may need to communicate some information to the user. In some circumstances, it might even have to prompt the user to choose among two or more alternatives. For example, a USB driver might gently remind the user that, next time, she should dismount a volume before physically disconnecting it. Or a mass storage driver might ask a user if he's really sure he wants to reinitialize the device, giving him an option to cancel the operation.

For these situations you can use the Kernel-User Notification Center (KUNC). The APIs of the KUNC facility enable you to do the following:

- Put up a non-blocking notification dialog.

  These notifications are strictly informative and permit the calling code to return immediately. Clicking the sole button (entitled "OK") merely closes the dialog.

- Present a blocking notification dialog.

  In this instance, the dialog has two or more buttons, each offering the user a different choice. Your driver blocks until the user clicks a button, and then responds in a manner appropriate to the button clicked.

- Launch a user-space program or open a preferences pane (System Preferences application).

  You can use this API to allow the user to install items or configure your driver.

- Load a more sophisticated user interface from a bundle.

  The bundle is typically the driver's kernel extension and the item loaded is an XML property list. This alternative allows elements such as pop-up lists, radio buttons, and text fields, as well as button titles and a customizable message. Because the information displayed is stored in a bundle, it can be localized.

The KUNC APIs should be used sparingly. You should send notifications to user space only when it is essential that the user knows about or reacts to a particular situation related to hardware. For example, a mass media driver should *not* put up a notification dialog when it encounters a bad block. However, if the user mounts an encrypted disk, the driver should provide some user interface enabling the user to unlock the disk.

Of course, any communication carried out through the KUNC facility can only be initiated by a kernel extension. If you have a user program that needs to communicate with a driver, you must use a suitable alternative, such as an Apple-provided device interface, a custom user client, POSIX I/O calls, or I/O Registry properties.

The KUNC APIs are defined in `Kernel.framework/Headers/UserNotification/`.

Presenting Notification Dialogs

For the simpler forms of notification, the KUNC APIs provide a function that displays a notice dialog (`KUNCNotificationDisplayNotice`) and a function that displays an alert dialog (`KUNCNotificationDisplayAlert`). The distinction between a notice and an alert is straightforward:

- A notice (Figure 5-1 (page 104)) is used merely to inform the user about some situation related to hardware; because it does not require a response from the user, the calling code is not blocked. A notice has only one button or, if a time-out is used, no button. Clicking the button simply dismisses the dialog.
- An alert (Figure 5-2 (page 104)) not only tells the user about some event related to hardware, but requests the user to make a choice among two or three alternatives related to that event. The code calling the KUNC alert function blocks until the user clicks one of the buttons to indicate a choice; it then must interpret which button was clicked and act appropriately. You can specify a time-out period, and if the period expires without the user making a choice, KUNC closes the dialog and returns the default action (associated with the blinking button).

Figure 5-1 A KUNC notice

Figure 5-2 A KUNC alert

The `KUNCUserNotificationDisplayNotice` and `KUNCUserNotificationDisplayAlert` functions have similar signatures, the latter's parameter list being a superset of the former's. Listing 5-1 (page 105) shows the prototype for `KUNCUserNotificationDisplayNotice`.

Listing 5-1 Definition of KUNCUserNotificationDisplayNotice

```
kern_return_t
KUNCUserNotificationDisplayNotice(
    int     timeout,
    unsigned    flags,
    char        *iconPath,
    char        *soundPath,
    char        *localizationPath,
    char        *alertHeader,
    char        *alertMessage,
    char        *defaultButtonTitle);
```

Table 5-1 (page 105) describes the parameters of this function.

Table 5-1 Parameters of the KUNCUserNotificationDisplayNotice function

Parameter	Description
timeout	Seconds to elapse before KUNC dismisses the dialog (and the user doesn't click the default button). If zero is specified, the dialog is not automatically dismissed.
flags	Flags that represent the alert level (stop, note, caution, or plain), and that indicate whether no default button should appear. These flags can be OR'd together. Check the constants defined in Core Foundation's CFUserNotification for the constants to use.
iconPath	File-system location of a file containing an ICNS or TIFF icon to display with the message text. Specify an absolute path or `NULL` if there is no icon file.
soundPath	File-system location of a file containing a sound to play when the dialog is displayed. Sounds in AIFF, WAV, and NeXT ".snd" formats are supported. Specify an absolute path or `NULL` if there is no icon file. (Currently unimplemented.able
localizationPath	File-system location of a bundle containing a `Localizable.strings` file as a localized resource. If this is the case, the text and titles specified for the dialog are keys to the localized strings in this file. Specify an absolute path or `NULL` if there is no `Localizable.strings` file.
alertHeader	The heading for the dialog.
alertMessage	The message of the dialog.
defaultButtonTitle	The title of the default button (the only button in this dialog). The conventional title "OK" is suitable for most situations.

The `KUNCUserNotificationDisplayAlert` function appends three more parameters (Table 5-2 (page 105)).

Table 5-2 Additional parameters of KUNCUserNotificationDisplayAlert

Parameter	Description
alternateButtonTitle	The title of the alternative button (for example, "Cancel").

Parameter	Description
otherButtonTitle	The title of a third button.
responseFlags	A constant representing the button clicked: `kKUNCDefaultResponse`, `kKUNCAlternateResponse`, `kKUNCOtherResponse`, `kKUNCCancelResponse`.

Listing 5-2 (page 106) illustrates how you might call `KUNCUserNotificationDisplayAlert` and (trivially) interpret the response. If the `kern_return_t` return value is not `KERN_SUCCESS`, then the call failed for some reason; you can print a string describing the reason with the `mach_error_string` function.

Listing 5-2 Calling KUNCUserNotificationDisplayAlert

```
kern_return_t ret;
unsigned int rf;
ret = KUNCUserNotificationDisplayAlert(
    10,
    0,
    NULL,
    NULL,
    NULL,
    "Something's Gone Terribly Wrong!",
    "Do you want to do anything about it?",
    "Yes",  // Default
    "No", // Alternative
    "Think About It", // Other
    &rf);

    if (ret == KERN_SUCCESS) {
        if (rf == kKUNCDefaultResponse)
            IOLog("Default button clicked");
        else if (rf == kKUNCAlternateResponse)
            IOLog("Alternate button clicked");
        else if (rf == kKUNCOtherResponse)
            IOLog("Other button clicked");
    }
```

Launching User-Space Executables

Your kernel code can use the Kernel-User Notification Center to launch applications. You can thereby engage the user's involvement in complex installation, configuration, and other tasks related to a driver or other kernel extension. You can use KUNC not only to launch applications, but to present preference panes and run daemons or other (non-GUI) tools.

The KUNC function that you call to launch an application is `KUNCExecute`. This function takes three parameters:

- **Path to executable** (*executionPath*). The file-system path to the application or other executable. The form of the path is indicated by the third parameter (*pathExecutionType*).

  Important: In upcoming versions of Darwin and Mac OS X, you will be able to specify an application's bundle identifier (`CFBundleIdentifier`) for this parameter. This will be the preferred method for identifying applications to launch.

- **User authorization** (*openAsUser*). The user-authorization level given the launched application. It can be either as the logged-in user (`kOpenAppAsConsoleUser`) or as root (`kOpenAppAsRoot`). Root-user access permits tasks such as copying to `/System/Library` without the need to authenticate; you should use root-user access judiciously.
- **Type of executable path** (*pathExecutionType*). The form of the executable path specified in the first parameter (*executionPath*). The parameter must be one of the following constants:

Constant	Description
`kOpenApplicationPath`	The absolute path to the executable. For an application, this is the binary in the application bundle's `MacOS` directory (for example, `/Applications/Foo.app/Contents/MacOS/Foo`).
`kOpenPreferencePanel`	The name of a preference pane in `/System/Library/PreferencePanes` (for example, `ColorSync.prefPane`).
`kOpenApplication`	The name of an application (minus the `.app`) in any of the system locations for applications: `/Applications`, `/Network/Applications`, or `~/Applications`.

Important: Future versions of the Kernel-User Notification Center will include a constant to indicate that an application's bundle identifier is being used in *executionPath*.

The code shown in Listing 5-3 (page 107) launches the Text Edit application and opens the Internet preferences pane.

Listing 5-3 Launching applications from a KEXT

```
kern_return_t ret;
ret = KUNCExecute("/Applications/TextEdit.app/Contents/MacOS/TextEdit",
kOpenAppAsRoot, kOpenApplicationPath);
ret = KUNCExecute("Internet.prefPane", kOpenAppAsConsoleUser,
kOpenPreferencePanel);
```

It's usually a good idea to put up a notice (function `KUNCUserNotificationDisplayNotice`) at the same time you launch an application or open a preference pane to inform users what they need to do with the application or preference pane.

Presenting Bundled Dialogs

The dialogs created through the `KUNCUserNotificationDisplayNotice` and `KUNCUserNotificationDisplayAlert` functions have a limited collection of user-interface elements to present to users. Apart from the header, icon, and message, they permit up to three buttons, and that's it. The only information they can elicit is the choice made through the click of a button.

The `KUNCUserNotificationDisplayFromBundle` function, however, enables you to present a much richer set of user-interface controls and to gather more information from the user. The elements that you can have on a bundled dialog include the following:

- Header, message text, sound, and icon
- Up to three buttons (default, alternative, other)
- Text fields, including password fields
- Pop-up lists
- Check boxes
- Radio buttons

Your kernel extension can obtain the selections that users make and any text that they type and process this information accordingly.

The controls and behavior of these kernel-user notification dialogs are captured as XML descriptions stored in a file within a bundle (typically the KEXT bundle itself). By virtue of being bundled, the XML descriptions are localizable. You can also update them without the need for a recompile. The XML file can be named anything. It consists of an arbitrary number of dictionary elements, one each for a dialog that the kernel extension might present. Each dialog description itself is a dictionary whose keys can be any of those described in Table 5-3 (page 108).

Table 5-3 KUNC XML notification dialog properties

Key	Type	Description
Header	string	The title of the dialog.
Message	string	Text of the message in the dialog. The text can include "%@" placeholders that, when the bundle is loaded, are replaced by "@"-delimited substrings in the *tokenString* parameter of the function `KUNCUserNotificationDisplayFromBundle`. See Table 5-4 (page 110) for this and other parameters.
OK Button Title	string	The title of the default button of the dialog (usually entitled "OK"). This button is the one closest to the right edge of the dialog.
Alternate Button Title	string	The title of the alternate button of the dialog (often entitled "Cancel"). This button is the one closest to the left edge of the dialog.
Other Button Title	string	The title of the third allowable button, which is placed between the other two buttons.
Icon URL	string	An absolute POSIX path identifying a file containing a TIFF or ICNS image to display next to the message text.
Sound URL	string	An absolute POSIX path identifying a file containing a sound to play when the dialog is displayed. Sounds in AIFF, WAV, and NeXT ".snd" formats are supported. (Currently unimplemented.)
Localization URL	string	An absolute POSIX path identifying a bundle containing a `Localizable.strings` file from which to retrieve localized text. If this property is specified, all text-related values in the XML description are assumed to be keys to items in `Localizable.strings`.

Key	Type	Description
Timeout	string	A number indicating the number of seconds before the dialog is automatically dismissed. If zero is specified, the dialog will not time out.
Alert Level	string	Takes "Stop", "Notice", or "Alert" to identify the severity of the alert. Each alert level has an icon associated with it. If you don't specify an alert level, "Notice" becomes the default.
Blocking Message	string	Either "0" or "1". If "1", the dialog has no buttons regardless of any specified.
Text Field Strings	array of strings	A list of labels to display just above the text fields they identify. See Figure 5-3 (page 112) to see the size and placement of these fields.
Password Fields	array of strings	One or more numbers identifying which of the text fields (which must also be specified) is a password field. In other words, each number is an index into the array of text fields. A password field displays "•" for each character typed in it.
Popup Button Strings	array of strings	A list of the titles for each element in a popup button.
Selected Popup	string	A number (index) identifying the element to display in the popup button. If this is not specified, the first element is displayed.
Radio Button Strings	array of strings	A list of the titles for each radio button to display. The radio buttons are aligned vertically. Only a single button can be selected.
Selected Radio	string	A number (index) identifying the radio button to show as selected.
Check Box Strings	array of strings	A list of the titles for each check box to display. The check boxes are aligned vertically. Multiple checkboxes can be selected.
Checked Boxes	array of strings	A number (index) identifying the check box (or check boxes) that are selected.

As indicated in the description of the `Localization URL` property above, you can access localized text in two ways. You can have localized versions of the XML description file itself. Or you can have a single XML description file and a `Localizable.strings` file for each localization. In the latter case, the `Localization URL` property points to the bundle containing the `Localizable.strings` files and the values of textual properties in the XML description file are keys into `Localizable.strings`.

A welcome aspect of the bundled-notifications feature of KUNC is that it is asynchronous. When you call the `KUNCUserNotificationDisplayFromBundle` function to request a dialog, the call returns immediately. Your code can go on to do other things. KUNC handles delivery of the request to user space, where the dialog is constructed and displayed. Throughout its journey, the request is identified by a notification ID that you supplied to `KUNCUserNotificationDisplayFromBundle`. When a user clicks one of the dialog buttons, a callback function implemented in your KEXT is invoked. There you can identify the request through its notification ID and handle the user's response.

Listing 5-4 (page 110) shows the signature of the `KUNCUserNotificationDisplayFromBundle` function.

Listing 5-4 Declaration of `KUNCUserNotificationDisplayFromBundle`

```
kern_return_t
KUNCUserNotificationDisplayFromBundle(
    KUNCUserNotificationID      notificationID,
    char                        *bundleIdentifier,
    char                        *fileName,
    char                        *fileExtension,
    char                        *messageKey,
    char                        *tokenString,
    KUNCUserNotificationCallBackcallback,
    int                         contextKey);
```

Table 5-4 (page 110) describes the parameters of this function.

Table 5-4 Parameters of `KUNCUserNotificationDisplayFromBundle`

Parameter	Description
notificationID	A value of type `KUNCUserNotificationID` that identifies the dialog request. You obtain this value through the `KUNCGetNotificationID` function.
bundleIdentifier	The location (specified as an absolute path) of the bundle containing the file with the XML descriptions of notification dialogs. An example of such a path might be "/System/Library/Extensions/MyDriver.kext". Upcoming versions of KUNC will permit the specification of the bundle's `CFBundleIdentifier` instead; this will be the preferred way to identify bundles.
fileName	The name of the file from which to retrieve the XML descriptions of dialogs. This name should omit the file's extension.
fileExtension	The extension of the XML file, minus the period; this can be anything, such as "dict" or "xml".
messageKey	Name of the XML key identifying the dialog description you want to have KUNC retrieve from the file.
tokenString	A string that can contain one or more token substrings. Each substring, which is separated from adjacent substrings by the "@" character, is substituted in turn for each "%@" placeholder in the message text of the dialog (XML key "Message"). Here is an example: if the message text is "Access denied after %@ attempts to access account %@." and the token string is "4@jdoe", the displayed string is "Access denied after 4 attempts to access account jdoe."
callback	The callback function that is invoked when the user clicks one of the buttons of the dialog. The function must conform to the `KUNCUserNotificationCallback` prototype.
contextKey	Any context information you wish to provide to help your kernel extension identify the notification request (in addition to the notification ID).

The final step in implementing bundled notifications for your kernel extension is to implement a callback function to handle user responses. This function must conform to the `KUNCUserNotificationCallback` prototype:

```
typedef void (*KUNCUserNotificationCallBack)(
    int     contextKey,
    int     responseFlags,
    void    *xmlData);
```

A couple of the parameters may seem familiar. The *contextKey* parameter is the same value you passed into the KUNCUserNotificationDisplayFromBundle function to help identify the request. The *responseFlags* parameter holds one of the constants that identify the dialog button the user clicked; these constants are discussed in "Presenting Notification Dialogs" (page 104). What is new is the *xmlData* parameter. This is the XML string data associated with the user's response; if the user has entered any information (such as user name and password) or has selected a popup item, radio button, or checkbox, your code should parse the XML and extract this information.

When you parse the XML, you should look for the corresponding element of the array you passed into KUNC. For example, if you have a user-name field and a password field, you have a Text Field Strings array with two elements. When you parse the *xmlData* parameter, the first element of Text Field Strings is the user name entered by the user; the second element is the password. Similarly, you parse the Selected Popup array to see which index into that array has been selected.

To illustrate how you might use the KUNC APIs for bundled notification dialogs, let's step through a couple of examples. You create an XML file called Messages.dict and put it in your KEXT's localized .lproj directories. This file contains the key "Password Message"; Listing 5-5 (page 111) shows the associated XML description for this key.

Listing 5-5 XML description of a password dialog

```
<key>Password Message</key>
<dict>
    <key>Alternate Button Title</key>
    <string>Cancel</string>
    <key>OK Button Title</key>
    <string>OK</string>
    <key>Header</key>
    <string>Authentication Message</string>
    <key>Message</key>
    <string>Please enter a user name and password.</string>
    <key>Timeout</key>
    <string>0</string>
    <key>Alert Level</key>
    <string>Stop</string>
    <key>Text Field Strings</key>
    <array>
        <string>User Name:</string>
        <string>Password:</string>
    </array>
    <key>Password Fields</key>
    <array>
        <string>1</string>
    </array>
    <key>Blocking Message</key>
    <string>1</string>
</dict>
```

In the code of your kernel extension, you call KUNCUserNotificationDisplayFromBundle, specifying "Password Message" as the message key (Listing 5-6 (page 112)).

Listing 5-6 Displaying the bundled password dialog

```
context = 2;
ret = KUNCUserNotificationDisplayFromBundle(KUNCGetNotificationID(),
    "/tmp/ExampleDriver/ExampleDriver.kext",
    "Messages",
    "dict",
    "Password Message",
    "",
    MyKUNCBundleCallback,
    context);
```

When this code is executed, the dialog shown in Listing 5-6 (page 112) appears on the user's screen, as in Figure 5-3 (page 112).

Figure 5-3 The password dialog

Of course, you can have even more complicated dialogs with a mix of user-interface elements. Listing 5-7 (page 112) shows the XML description of a dialog that includes text and password fields and radio buttons.

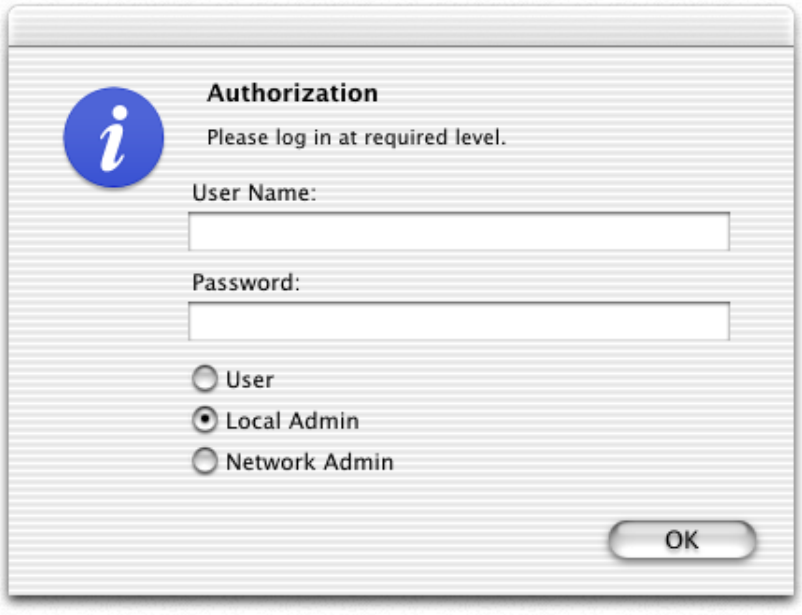
Listing 5-7 XML description of a dialog with various controls

```
    <key>Mixed Message</key>
    <dict>
        <key>OK Button Title</key>
        <string>OK</string>
        <key>Header</key>
        <string>Authorization</string>
        <key>Message</key>
        <string>Please log in at required level.</string>
        <key>Timeout</key>
        <string>20</string>
        <key>Alert Level</key>
        <string>Stop</string>
        <key>Text Field Strings</key>
        <array>
            <string>User Name:</string>
            <string>Password:</string>
```

```
        </array>
        <key>Password Fields</key>
        <array>
            <string>1</string>
        </array>
        <key>Radio Button Strings</key>
        <array>
            <string>User</string>
            <string>Local Admin</string>
            <string>Network Admin</string>
        </array>
        <key>Selected Radio</key>
        <string>1</string>
    </dict>
```

When your loaded kernel extension calls the `KUNCUserNotificationDisplayFromBundle` function, supply this time "Mixed Message" as the XML dictionary key, the dialog shown in Figure 5-4 (page 113) appears on the user's screen.

Figure 5-4 Display of a mixed control dialog



Displaying Localized Information About Drivers

A frequently repeated fact in Darwin documentation is that a kernel extension is a bundle. What does this mean, and what are the implications?

A bundle is a directory with a strictly defined internal structure that (usually) contains executable code and the resources that support that code. A bundle on Mac OS X is the primary form of packaging for executables in the file system. Applications and frameworks are bundles, as are all other loadable bundles such as kernel extensions. Although some executables are not bundles—most notably command-line tools, some libraries, and CFM-PEF applications—they are definitely in the minority. The Finder presents most bundles to users opaquely, as a file.

A bundle can be programmatically represented by a NSBundle object (for Cocoa code) or by a CFBundle opaque type (for all other application environments). The APIs of NSBundle and CFBundle permit your executable to access and manipulate the resources in the represented bundle.

Among the benefits that the bundle form of packaging brings to executables is internationalization. Internationalization is the mechanism by which executables on Mac OS X store localized resources in bundles and access them in order to present them in a graphical user interface. Localized resources are those that have been translated and adapted to particular languages and locales; text, images, and sounds are among the resources that are frequently localized. Mac OS X attempts to present those localizations that match a language preference of the user.

Device drivers and other kernel extensions occasionally need to have localized information displayed to users during tasks that require user input, such as configuration and installation. But you should remember that kernel extensions are a different breed of bundle. The executable code in these bundles is loaded into the kernel and kernel code, by definition, does not play a direct role in user interfaces. If a kernel extension contains localized resources for display, it must have a user-space agent to do the displaying for it. In many cases, the kernel extension will have a helper application or preference pane to present the localized user interface.

One might ask: why not put all localized resources in the helper application instead of the kernel extension? Doing so would seem to make it easier to locate the resources. That may be true when there is only one kernel extension per helper application. But typically a helper application is responsible for a family of kernel extensions, both current and anticipated products. If the application is to contain all localized resources, it must have prior knowledge of every kernel extension it might display information about. Because this is impractical, it makes more sense for a kernel extension to encapsulate its own localizations.

This section provides an overview of internationalizing kernel extensions and describes the special APIs and procedures that code in user space must use to access localized resources stored in these bundles.

Important: The Kernel–User Notification Center (KUNC) enables a kernel extension to display a dialog that can have a fairly sophisticated set of user-interface elements and whose text, images, and sounds can be localized. You must internationalize the bundle containing these localized resources, but you need only use the KUNC APIs to have them displayed. For more on the KUNC facility, see "Kernel-User Notification" (page 103).

Internationalizing Kernel Extensions

Internationalizing your kernel extension is a fairly painless process. You need to set up your project files in a certain way and you may need to modify code. Of course, you must have localized resources to put into your project. This means someone must complete all necessary translations and adaptations of the following resources:

- Images (for example, the equivalent of a U.S. stop sign in other cultures)
- Sounds (for example, a recorded voice speaking a phrase)
- Long text files (for example, an HTML help file)
- Nib files (user-interface archives created by the Interface Builder application)
- Strings that appear in the user interface (for example, titles of buttons and text fields)

This section summarizes the steps you must follow to internationalize your kernel extension. For all single-file resources such as images, sounds, nib files, and help files, internationalization is a simple matter of putting the resource files in certain subdirectories of the kernel extension. In your user-space code, you may then have to use CFBundle (or NSBundle) APIs to obtain a path to a resource file in the current localization prior to loading that resource. The procedure for internationalizing user-interface strings is a bit more involved; this procedure is summarized in "Internationalizing Strings" (page 117).

Note: The procedures for internationalizing bundles are introduced in *Mac OS X Technology Overview*. For more in-depth information, see Internationalization Documentation.

Creating and Populating the Localization Directories

If you looked at the internal structure of a simple internationalized kernel extension in the file system, you might see something like this:

```
MyDriver.kext/
    Contents/
        MacOS/              // contains kernel binary
        Resources/
            Logo.tiff           // non-localized resource
            en.lproj/
                Stop.tiff       // English-localized resource
            fr.lproj/
                Stop.tiff       // French-localized resource
```

As you can see, a bundle's resources go (logically enough) in the `Resources` subdirectory of the bundle. Resources at the top level of `Resources` are non-localized and thus are displayed in the user interface regardless of the logged-in user's language preference. Localized resources (the two image files named `Stop.tiff` in the above examples) are put in subdirectories of `Resources`; each of these directories has an extension of `.lproj` and a name that is either the English name of a major language or an ISO 639 abbreviation for a language (such as "en"). The name can also have a suffix that is an ISO 3166 abbreviation for locale (for example, "en_US" for U.S. English). When an application needs to display a localized resource, Mac OS X gets the logged-in user's language preferences—this is a sorted list of preferred languages set through the International pane of System Preferences—and goes down the list until it finds the first language matching one of the `.lproj` localization directories in the bundle. This is how the internationalization mechanism basically works.

You can create the internal localization structure of your bundle by hand, but it is more convenient to let the Project Builder application do much of the work for you. The procedure is fairly straightforward. In Project Builder:

1. Create a group for localized and non-localized resources.

   Click the Files tab. Choose New Group from the Project menu and name the new group. (Go ahead, name it "Resources".)

2. Add a resource to the group.

   Choose Add Files from the Project menu. In the browser, locate and select a file containing a sound, image, text, or other resource. When this file appears in the Files pane of Project Builder, drag it into the Resources group, if necessary.

3. Mark the localization attribute of the file.

   a. Select the file and choose Show Info from the Project menu.

   b. From the Localization & Platforms pop-up list, select Add Localized Variant.

   c. In the sheet that next appears, either select a language name from the combo box or, if the localization is for some other language, type the ISO 639 abbreviation in the text field. Click OK.

   d. If you later decide the resource should not be localized, select it and choose Make Global from the Localization & Platforms pop-up list.

Complete the above procedure for every resource, localized and non-localized, that you want included in your project. When you build the project, Project Builder will create an `.lproj` directory for each localization and put all resources designated for that localization in the directory. The name and extension of the resource file must be exactly the same for each localization.

Internationalizing Strings

If an application gets the localized strings for its user interface from a nib file, you need only store localized nib files in each `.lproj` directory of the kernel extension to internationalize them. However, if an application is to display localized strings for a kernel extension *programmatically*, you need to have a strings file in each `.lproj` directory of the kernel extension. A strings file is so-called because its extension is `.strings`; the conventional name before the extension is `Localizable`, but this is not a requirement.

The format of a strings file is a simple key-value pairing for each string entry; the entry is terminated by a semicolon and a carriage return (that is, one entry per line). The key is a string in the development language; the value is the localized string. Between the key and the value is an equal sign. For example:

```
/* a comment */
/* "key" = "value"; */
"Yes" = "Oui";
"The same text in English" = "Le meme texte en francais";
```

Once you've created a localized strings file, put it in the appropriate `.lproj` directory (as described in "Creating and Populating the Localization Directories" (page 116)). The next step is to modify the code of the application to display the localized string. At each point where a localized string is needed, use the macro `CFCopyLocalizedString` (and variants) if you are using CFBundle APIs; if you are using Cocoa's NSBundle, use the `NSLocalizedString` macro (and variants) instead. In the simplest version of the macro, you specify two parameters: the key and a comment. For example:

```
CFStringRef locstr = CFCopyLocalizedString(CFSTR("Yes"), CFSTR(""));
```

If the logged-in user's language preference is French, the variable `locstr` holds the value of "Oui". The comment is there for a particular reason. Instead of creating a strings file from scratch, you can put all required `CFCopyLocalizedString` (or `NSLocalizedString`) calls in your application code first. In the comment parameter, give any instructions to the translator for the string. Then run the `genstrings` command-line utility on your source-code file to generate a strings file. You can then give a copy of the generated file to the translator for each localization.

Important: The XML description file used by the Kernel-User Notification Center (KUNC) is similar to the `Localizable.strings` file in that, for each dialog, a key (in the development language) can be paired with a localized string value (or an array of such values). Alternatively, you can create `Localizable.strings` files for each localization and the values of the XML keys can themselves be keys in the strings file. Also, KUNC handles the localization details and not CFBundle or NSBundle (directly, at least).

Getting the Path to a KEXT From User Space

If you're responsible for the user-space code that must display your kernel extension's localized information, you'll need to get the path to your KEXT. CFBundle and NSBundle require a path to a bundle to create a bundle object, which you need to programmatically access the resources of the bundle.

The solution to this is to pass the `CFBundleIdentifier` of your KEXT to a special KEXT Manager function to obtain the bundle path. The KEXT Manager defines the following function for this purpose (in `IOKit.framework/Headers/kext/KextManager.h`):

```
CFURLRef KextManagerCreateURLForBundleIdentifier(
    CFAllocatorRef allocator,
    CFStringRef    bundleIdentifier);
```

The `KextManagerCreateURLForBundleIdentifier` function returns a CFURLRef object representing the path to any KEXT currently installed in `/System/Library/Extensions` identified by the passed-in `CFBundleIdentifier`. For the CFAllocatorRef parameter, you can obtain the current default allocator with a call to `CFAllocatorGetDefault`.

Once you have the path to the kernel extension (as a CFURLRef object), you can create a bundle object (CFBundleRef) from it using the following CFBundle function:

```
CF_EXPORT
CFBundleRef CFBundleCreate(CFAllocatorRef allocator, CFURLRef bundleURL);
```

You can make the bundle object the subsequent target of calls for returning localized resources, such as strings, images, and icons.

Debugging Drivers

The hardest part of debugging code—especially kernel code—is not learning how to use the tools but how to analyze the data generated by the tools. The success you have with your analysis depends to a large degree on the skills and experience you bring to it, and these can always be enhanced by learning from others who have "earned their stripes" debugging kernel code.

This chapter offers a collection of driver-debugging tips and techniques from Apple engineers. It starts by listing a few general suggestions for driver debugging and itemizing the tools at your disposal. It then discusses these general areas of driver debugging:

- Debugging drivers during the matching and loading stages
- Two-machine debugging, including the set-up procedure, the kernel debugging macros, and debugging panics and hangs
- Logging, including custom event logging

Note: For information on debugging a universal binary version of a device driver running on an Intel-based Macintosh, see "Debugging on an Intel-Based Macintosh" (page 161).

Some Debugging Basics

Before delving into the nitty-gritty of driver debugging, it might be worth your while to consider a few general debugging tips and to survey the debugging tools available to you.

General Tips

To help you get started, here are a few debugging tips based on the experiences of Apple kernel engineers.

- Develop in incremental steps. In other words, don't write a big chunk of driver code before debugging it. And at each stage verify that the driver can unload as well as load; a driver that can't unload suggests a memory leak (most likely caused by a missing `release` somewhere).

  As a suggestion, get driver matching and the basic driver life-cycle methods working first (especially `start`). Then begin talking with your hardware. Once you're sure that's working properly, then write the code that talks with the rest of the system.
- Keep earlier versions of your code. If bugs start to appear in your code, you can "diff" the current version with the prior version to locate exactly which code is new (and thus the probable source of the problem).
- Obtain the source code for all other parts of the kernel that your driver interacts with or depends on (directly or indirectly). Things go much easier when you have full source-level debugging.

- Obtain symbols for each kernel extension that might affect, or be affected by, your driver. This would include your driver's provider and its clients. For the full gamut of symbols, obtain a symboled kernel. In particular, a symboled kernel is required to use the kernel debugging macros (described in "Using the Kernel Debugging Macros" (page 134)).

  See "Setting Up for Two-Machine Debugging" (page 131) for instructions on generating symbol files for kernel extensions. To obtain a symboled kernel, you might have to build the Darwin kernel from the open source. See Building and Debugging Kernels in *Kernel Programming Guide* for details.
- Learn `gdb` thoroughly. Start by reading *Debugging with GDB*, provided as part of Tools Documentation.
- Become familiar with how computer instructions look in assembler (see "Examining Computer Instructions" (page 139)).
- Every now and then, single-step through every line of code to make sure your driver's doing exactly what you want it to.
- Don't overly rely on calls to `IOLog` to provide you with debugging information (see "Using IOLog" (page 147)).
- Make sure your KEXT includes only header files from `Kernel.framework`, in addition to the header files you define. If you include other headers, although their definitions may be in scope at compile time, the functions and services they define will not be available in the kernel environment.

Issues With 64-Bit Architectures

In Mac OS X version 10.3, Apple introduced some changes to support the new 64-bit architectures. Because the changes are implemented in the operating system itself, some drivers may be affected even if they aren't running on a 64-bit computer. To better explain these changes, this section first provides a brief description of PCI address translation.

Hardware devices on the PCI bus can handle 32-bit addresses, which means that a PCI device has a 4-gigabyte window into main memory. When a PCI device performs a data transaction to or from main memory, the device driver prepares this memory for I/O. On systems where both the memory subsystem and the PCI device drivers use 32-bit addressing, there are no difficulties. On systems where the memory subsystem uses 64-bit addressing, however, a PCI device can see only four gigabytes of main memory at a time. To address this issue, Apple chose to implement address translation. In this scheme, blocks of memory are mapped into the 32-bit address space of PCI devices. The PCI device still has a 4-gigabyte window into main memory, but that window may contain noncontiguous blocks of main memory. The address translation is performed by a part of the memory controller called the DART (device address resolution table). The DART keeps 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).

If your driver adheres to documented, Apple-provided APIs, this address-translation process is transparent. For example, when your driver calls IOMemoryDescriptor's `prepare` method, 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 so may not have resulted in undesirable behavior in previous versions of Mac OS X. Beginning in Mac OS X version 10.3, however, failure to follow this procedure may result in random data corruption or panics.

Note: Be aware that the `release` method does not take the place of the IOMemoryDescriptor `complete` method. As always, every invocation of `prepare` must be balanced with an invocation of `complete`.

If your driver experiences difficulty on a Mac OS X version 10.3 or later system, you should check that you are following these guidelines:

- Always call `IOMemoryDescriptor::prepare` to prepare the physical memory for the I/O transfer (in Mac OS X version 10.3 and later, this also places a mapping into the DART).
- Balance each call to `prepare` with a call to `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.

A side effect of the changes in the memory subsystem is that Mac OS X is much more 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 going on 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 page ranges in memory regions might expose latent bugs in drivers that previously did not have to handle physically contiguous pages. If your driver is behaving incorrectly or panicking, be sure to investigate this possibility.

Another result of the memory-subsystem changes concerns the physical addresses some drivers 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 to get such addresses (such as `pmap_extract`) will fail to function on systems with a DART. To prevent the use of these calls, Mac OS X version 10.3 will refuse to load a kernel extension that uses them, even on systems without a DART.

Driver-Debugging Tools

Apple provides a number of tools that you can use to debug I/O Kit device drivers. Table 7-1 (page 123) presents some of the more important tools.

Table 7-1 Debugging aids for kernel extensions

Tool or API	Description
`gdb`	The GNU debugger, used in two-machine kernel debugging. See "Tips on Using gdb" (page 138) for some useful information on `gdb`.
Kernel debugging macros	Powerful macros designed for use in `gdb` while debugging the Darwin kernel. The macros are contained in the file `.gdbinit` in the Open Source directory location `/xnu/osfmk/`. See "Using the Kernel Debugging Macros" (page 134) for more information.

Tool or API	Description
`kextload`	A utility that does a variety of things with kernel extensions. It loads kernel extensions and, prior to loading, validates the extensions, providing diagnostic information related to file-permission and dependency errors. It also enables the debugging of kernel extensions (using `gdb`) during the matching and loading phases. See "Loading Your Driver" (page 125) for more on `kextload` options.
`kextstat`	A utility that prints columns of information about the kernel extensions that are currently loaded in a system. The information most useful to driver debugging is the driver's load address, references held by the driver to other kernel extensions, the number of references other extensions have to it, and the version of a kernel extension. See "Unloading Your Driver" (page 127) for more information.
`ioreg`	Provides a snapshot of the I/O Registry, showing the hierarchical structure of client–provider relationships among current drivers and nubs in the system. With the necessary options, it shows the properties associated with each node in the registry. The I/O Registry Explorer shows the same information in a GUI interface. Both tool and application are particularly useful for debugging problems with matching. See "Debugging Matching Problems" (page 130).
`ioalloccount` `ioclasscount`	The former tool displays a summary of memory allocation by allocator type (instance, container, and `IOMalloc`). The latter tool shows the number of instances allocated for each specified class. Both tools are useful for tracking down memory leaks. See "Unloading Your Driver" (page 127) for more information.
`IOKitDebug.h`	Defines a set of values for the IOKitDebug property. When this property is defined, the system writes the status of various driver events, such as attaching, matching, and probing, to the system log. See "Debugging Matching Problems" (page 130) for further information.

Debugging Matching and Loading Problems

Before you begin debugging your driver's hardware-specific functionality, you should first ensure that your KEXT is valid and that it loads, unloads, and matches properly. This section describes how to use Mac OS X tools, such as `kextload` and `kextstat`, to authenticate and validate your KEXT, resolve its dependencies, and test its matching.

Driver Dependencies

Every driver declares its dependencies on other loadable kernel extensions (such as an I/O Kit family), kernel subcomponents (such as `com.apple.kernel.iokit`), or KPIs (such as `com.apple.kpi.libkern`) in its `Info.plist` file. At the top level of the `Info.plist` file, a driver lists each dependency in the form of a key-value pair in the `OSBundleLibraries` dictionary; the key is the KEXT or kernel subcomponent name and the value is the KEXT's or subcomponent's version number.

In Mac OS X v10.2, declaring a dependency on the I/O Kit kernel subcomponent `com.apple.kernel.iokit` (version 6.x) implicitly allows your driver to access symbols in two other kernel subcomponents: `com.apple.kernel.mach` and `com.apple.kernel.bsd`.

In Mac OS X v10.3, a driver that declares a dependency on `com.apple.kernel.iokit` version 6.x (the 10.2 version) will still enjoy automatic access to Mach and BSD symbols.

Important: A driver that declares a dependency on version 7 of `com.apple.kernel.iokit` (the 10.3 version) will *not* have automatic access to Mach and BSD symbols. If a driver requires access to those symbols, it must explicitly declare dependencies on `com.apple.kernel.mach` and `com.apple.kernel.bsd`.

In Mac OS X v10.4, Apple introduced sustainable kernel programming interfaces, or KPIs. In particular, the KPIs support the development of NKEs (network kernel extensions), file-system KEXTs, and other non-I/O Kit KEXTs. KPIs are also recommended for pure I/O Kit KEXTs (KEXTs that use only I/O Kit-provided APIs) that target Mac OS X v10.4 and later. Be aware, however, that no KEXT can declare dependencies on a combination of both kernel subcomponents and KPIs. To learn more about KPIs, see *Kernel Framework Reference*; to find out how to declare dependencies on them, see Kernel Extension Dependencies.

Using kextload, kextunload, and kextstat

During the development process, it's a good idea to load and unload your driver regularly to catch problems in these processes before they become very difficult to find. The `kextload` tool allows you to load your driver at the command line and performs a wide range of authentication and validation tests. `kextload` also allows you to debug your driver's `start` and `probe` routines before matching begins. The `kextunload` tool terminates and unregisters the I/O Kit objects associated with your KEXT and unloads the code and personalities for that KEXT. For complete information on `kextload`, `kextunload`, and `kextstat`, view the `man` pages for these tools in a Terminal window.

You can use `kextload` to load and start your KEXT with or without starting I/O Kit matching, but `kextload` provides no information about the matching process itself. If your driver isn't matching properly, see "Debugging Matching Problems" (page 130) for more information.

Loading Your Driver

The `kextload` tool checks your driver's `Info.plist` file thoroughly before it attempts to load your driver into the kernel. If required properties are missing or incorrectly identified, if your declared dependencies are missing or incompatible, or if your driver has the wrong permissions or ownership, `kextload` refuses to load your KEXT. To find out why your KEXT doesn't load, run `kextload` with the `-t` or the `-tn` option. This tells `kextload` to perform all its tests on your KEXT and provide you with a dictionary of the errors it finds (the addition of the `-n` option prevents `kextload` from loading the KEXT even if it finds no errors). For example, Listing 7-1 (page 125) shows the `Info.plist` of a KEXT named BadKEXT, that contains three errors.

Listing 7-1 Example `Info.plist` containing errors

```
<key>IOKitPersonalities</key>
<dict>
    <key>BadKEXT</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.MySoftwareCompany.driver.BadKEXT</string>

        <key>IOClass></key>
        <string>com_MySoftwareCompany_driver_BadKEXT</string>

        <!-- No floating point numbers are allowed in the kernel. -->
```

```
        <key>IOKitDebug</key>
        <number>0.0</number>

        <key>IOMatchCategory</key>
        <string>com_MySoftwareCompany_driver_BadKEXT</string>

        <key>IOResourceMatch</key>
        <string>IOKit</string>

        <!-- The personality is missing its IOProviderClass key-value pair. -->
    </dict>
</dict>
<key>Libraries</key>
<dict>
    <key>com.apple.kernel.iokit</key>
    <string>1.1</string>

    <!-- com.apple.kernel.libkern is misspelled. -->
    <key>com.apple.kernel.libker</key>
    <string>1.1</string>

    <key>com.apple.kernel.mach</key>
    <string>1.1</string>
</dict>
```

In addition, BadKEXT has ownership and permissions problems: It is owned by the user `user_name` and the group `staff` and its folders have permissions `rwxrwxr-x` (775 octal). For reasons of security, KEXTs must be owned by the root user and the wheel group. Further, no component of a KEXT may be writable by any user other than root. (For more information on the correct permissions and ownership of KEXTs, see "Testing and Deploying Drivers" (page 151)). Assuming root privileges by using the `sudo` command and running `kextload -t` on BadKEXT provides the following information, shown in Listing 7-2 (page 126).

Listing 7-2 `kextload -t` outputs errors found in KEXT

```
[computer_name:] user_name% sudo kextload -t BadKEXT.kext
Password:
can't add kernel extension BadKEXT.kext (validation error) (run kextload
                            on this kext with -t for diagnostic output)
kernel extension BadKEXT.kext has problems:
Validation failures
{
    "Info dictionary missing required property/value" = {
        "IOKitPersonalities:BadKEXT:IOProviderClass"
    }
    "Info dictionary property value is of illegal type" = {
        "IOKitPersonalities:BadKEXT:IOKitDebug"
    }
}
Authentication failures
{
    "File owner/permissions are incorrect" = {
    "/Users/user_name/Projects/BadKEXT/build/BadKEXT.kext"
    "/Users/user_name/Projects/BadKext/build/BadKEXT.kext/Contents/Info.plist"
    "/Users/user_name/Projects/BadKext/build/BadKEXT.kext/Contents"
   "/Users/user_name/Projects/BadKext/build/BadKEXT.kext/Contents/MacOS/BadKEXT"
        "/Users/user_name/Projects/BadKext/build/BadKEXT.kext/Contents/MacOS"
    }
```

```
}
Missing dependencies
{
    "com.apple.kernel.libker" =
        "No valid version of this dependency can be found"
}
```

The checks `kextload` performs are listed below.

- `Info.plist` dictionary must be present and must be of type `dictionary`.
- `CFBundleIdentifier` property must be of type `string` and no longer than 63 characters.
- `CFBundleVersion` property must be a proper "vers" style `string`.
- If the optional `OSBundleCompatibleVersion` property is present, it must be a proper "vers" style string with a value less than or equal to the `CFBundleVersion` value.
- `CFBundleExecutable` must be of type `string`.
- If the `IOKitPersonalities` property is present (the `IOKitPersonalities` property is required for driver matching but isn't required for the KEXT to be valid), all its values must be of type `dictionary`. `kextload` performs the following checks on the personalities.
    - `IOClass` property must be of type `string`.
    - `IOProviderClass` property must be of type `string`.
    - `CFBundleIdentifier` must be of type `string`.
    - `IOKitDebug` property, if present, must be of type `integer` (and *not* a floating-point number).
    - All property types must be valid for in-kernel use; for example, the `date` type is not allowed in the kernel.
- `OSBundleLibraries` must be present and be of type `dict`. The `OSBundleLibraries` values must exist and have valid versions.
- If all personalities have the `IOKitDebug` property set to a nonzero value, `kextload` marks the KEXT an ineligible for safe boot, even if the `OSBundleRequired` property is present and valid.

If the `-t` option doesn't provide you with enough information, you can tell `kextload` to give you a step-by-step account of its actions by using the `-v` option followed by a number from 1 to 6 indicating how much information you want. In this way, you can see at what point `kextload` fails.

Unloading Your Driver

The `kextunload` tool simply tries to unload your driver's code and personalities, terminating and unregistering all I/O Kit objects associated with your driver. Unfortunately, `kextunload` provides very little information about why your driver might fail to unload. It's possible that the system log (which you can view at `/var/log/system.log`) will display some information about a failure to unload, but you'll get the most useful information using the other methods and tools described in this section.

A driver that has a mismatched `retain` or `release`, one that is depended on by other kernel modules, or one that has a failing `stop` method will fail to unload. Whatever the cause, if your driver fails to unload, the debugging process becomes arduous because you are forced to restart the computer each time you need to load and unload your KEXT.

In general, you should make only incremental changes in your driver's code, checking its ability to load and unload after every change, to narrow down the possible culprits if your driver suddenly refuses to unload. In this way, you can revert to a previous, unloadable version of your driver and examine the changes you've made to it.

You can determine if your driver cannot unload because another module is depending on it by using the `kextstat` tool to display information about all currently loaded KEXTs. Listing 7-3 (page 128) shows a few lines of the output from `kextstat`.

Listing 7-3 Partial listing of `kextstat` output

```
Index Refs Address  Size     Wired Name          (Version) <Linked Against>
 1   1   0x0        0x0      0x0  com.apple.kernel (6.0)
 2   8   0x0        0x0      0x0  com.apple.kernel.bsd (6.0)
 3  33   0x0        0x0      0x0  com.apple.kernel.iokit (6.0)
 4  32   0x0        0x0      0x0  com.apple.kernel.libkern (6.0)
// Some lines not shown...
11   7   0xaadd000  0x9000   0x8000 com.apple.iokit.IOPCIFamily (1.2) <4 3>
// Some lines not shown...
26   6   0xb14c000  0x1a000 0x19000 com.apple.iokit.IOUSBFamily (1.2) <4 3 2>
// Some lines not shown...
```

The first column of the `kextstat` output is the load index of the KEXT. `kextstat` uses this index in the last column to show which KEXTs a particular KEXT is linked against. For example, the IOPCIFamily KEXT, index 11 in Listing 7-3 (page 128), depends on both `com.apple.kernel.iokit` (index 3) and `com.apple.kernel.libkern` (index 4).

The second column is the sum of all the references to the selected KEXT by other currently loaded KEXTs. Check this column if your driver fails to unload: If its number of references is nonzero, `kextunload` cannot unload it.

If there are no other KEXTs depending on your driver yet it fails to unload, the problem is often a mismatched `retain` or `release` on an object instantiated from one of your classes. If you have multiple classes defined in your KEXT, you can use `ioclasscount` to find out which class has outstanding instances. In a Terminal window, type

```
ioclasscount MyClassName
```

The `ioclasscount` tool outputs the instance count of the specified object, offset by the number of direct subclasses that have at least one instance allocated. With this information, you then go through your driver's code looking for a `retain` on your object that's not balanced with a `release`. A mismatched `retain` can be difficult to find because often it is not explicitly in your code, but in a container class you've put your object into, in a method you've called, or in a still-running user application that holds an `io_object_t` reference on the object. For example, calling

```
addEventSource ( My_Event_Source_Object )
```

implicitly places a retain on My_Event_Source_Object that remains until you call

```
removeEventSource ( My_Event_Source_Object )
```

Debugging Your Driver's start and probe Methods

When you've determined that your KEXT can load and unload successfully, you may want to debug your driver's `start` and `probe` methods. To do this, you use `kextload`. This is because installing your KEXT in `/System/Library/Extensions` and allowing `kextd` to load it when needed does not present you with the opportunity to attach to the computer in a `gdb` session before your driver has already matched and started.

If you are developing an I/O Kit driver, you can use `kextload` with the `-l` option to load your KEXT binary on the target computer but to refrain from sending any personalities to the kernel. By separating loading from matching, `kextload -l` allows you to defer matching (and the calling of your driver's `start` and `probe` methods) until after you set up a two-machine debugging session (for more information on how to set up such a session, see "Setting Up for Two-Machine Debugging" (page 131)).

To avoid matching before you're ready, you can invoke `kextunload` on your driver before using `kextload -l` to make sure none of your driver's personalities remain in the kernel from an earlier debugging session. Be aware, however, that plugging in a new device after you've used `kextload -l` to load your KEXT binary may cause some other driver to match on the device. If this is not desirable, take care not to change your system's configuration until you are ready.

After you've attached to the target computer and set breakpoints on the `start` and `probe` methods, you use `kextload` with the `-m` option to send the personalities to the kernel, which triggers matching. If your KEXT's `start` and `probe` methods do not execute, a successful match did not occur and you should examine the driver personalities to find out why. See "Debugging Matching Problems" (page 130) for advice on how to debug matching problems.

If you are developing a non–I/O Kit KEXT, such as an NKE (network kernel extension) or a file system KEXT, you implement the `MODULE_START` method instead of the `start` method an I/O Kit driver implements. To debug the `MODULE_START` method, you use `kextload` with the `-i` or `-I` options to interactively load the KEXT and its dependencies, pausing after each step for your permission to continue.

Using the `-i` option tells `kextload` to automatically load your KEXT's dependencies but to ask for permission to continue at each step in the process of loading the KEXT itself. The `-I` option causes `kextload` to pause for input at each step of every stage of loading, including the loading of dependencies. If you want to debug your KEXT's `MODULE_START` method, you use `kextload -i` or `-I` to load your KEXT binary and then set up a two-machine debugging session before you give permission to invoke the `MODULE_START` method. Listing 7-4 (page 129) shows the loading of an example KEXT called MyKEXT using `kextload -i`.

Listing 7-4 Using `kextload -i` for interactive KEXT loading

```
[computer_name:/tmp] user_name% sudo kextload -i MyKEXT.kext
Password:
Load extension MyKEXT.kext and its dependencies [Y/n]? y

Load module /tmp/MyKEXT.kext/Contents/MacOS/MyKEXT [Y/n]? y
kextload: module com.MySoftwareCompany.driver.MyKEXT created as #70 at
                                        address 0xaf5c000, size 8192
kextload: You can now break to the debugger and set breakpoints for this
                                                            extension.

Start module /tmp/MyKEXT.kext/Contents/MacOS/MyKEXT (answering no will abort
                                                    the load) [Y/n]? y
kextload: started module /tmp/MyKEXT.kext/Contents/MacOS/MyKEXT
kextload: MyKEXT.kext loaded successfully
```

```
Send personality "MyKEXT" to the kernel [Y/n]? y
kextload: matching started for MyKEXT.kext
[computer_name:/tmp] user_name%
```

Debugging Matching Problems

Getting your driver to match correctly can be one of the more challenging aspects of I/O Kit driver development. If your driver loads and unloads properly but does not match, the most important thing you can do is become thoroughly familiar with your family's matching language. Although the I/O Kit implements driver matching in a three-phase, subtractive process (described in detail in*I/O Kit Fundamentals*), each I/O Kit family is free to define an arbitrarily complex matching language of its own.

After the I/O Kit discards drivers of the wrong class in the class-matching phase, the I/O Kit examines the remaining drivers for family-specific matching properties in the passive-matching phase. If the family implements family-specific matching, it considers the candidate driver's personality and raises or lowers the driver's `probe` score based on the matches it finds.

Although some families define no family-specific matching, instead preferring drivers to probe the device, many others exercise some control over the passive-matching phase. The USB family, for example, requires its drivers to use specific combinations of properties defined by the USB Common Class Specification. In order for your USB driver to match, you must include all the elements of exactly one property combination. Another example is the PCI family which defines a number of match keys whose values are the contents of various PCI configuration-space registers. A candidate PCI driver can then match on a single register value, a list of values, or even a partial register value indicated by a supplied bit mask.

If your driver does not match, first make sure the property values in your driver's personality match the corresponding properties your device publishes in the I/O Registry. You can use the I/O Registry Explorer application (available at `/Developer/Applications`) or the command-line tool `ioreg` to view the properties your device publishes. If you use I/O Registry Explorer, select Find to search for your device name, follow the path to the object representing your device, and view its properties in the lower window. Alternately, to view the I/O Registry in a Terminal window, type

```
ioreg -bl
```

The `-b` option displays the object names in bold and the `-l` option displays the object properties.

If there are no discrepancies between your device's published properties and your driver's matching properties, next make sure you know how your driver's family implements matching. Families that define their own matching language do so by implementing the `matchPropertyTable` method. By viewing the code for this method, you can determine how your driver's family uses the properties it finds in a driver's personality dictionary to adjust the `probe` score.

In very rare cases, a driver might declare `IOResources` as the value of its `IOProviderClass` key. `IOResouces` is a special nub attached to the root of the I/O Registry that makes resources, such as the BSD kernel, available throughout the system. Traditionally, drivers of virtual devices match on `IOResources` because virtual devices do not publish nubs of their own. Another example of such a driver is the HelloIOKit KEXT (described in Hello I/O Kit: Creating a Device Driver With Xcode) which matches on `IOResources` because it does not control any hardware.

Important: Any driver that declares `IOResources` as the value of its `IOProviderClass` key must also include in its personality the `IOMatchCategory` key and a private match category value. This prevents the driver from matching exclusively on the `IOResources` nub and thereby preventing other drivers from matching on it. It also prevents the driver from having to compete with all other drivers that need to match on `IOResources`. The value of the `IOMatchCategory` property should be identical to the value of the driver's `IOClass` property, which is the driver's class name in reverse-DNS notation with underbars instead of dots, such as `com_MyCompany_driver_MyDriver`.

Finally, you can place the `IOKitDebug` property in your driver's personality dictionary. When you give this property a non-zero value, it places the status of various events, such as attaching, matching, and probing, into the system log (available at `/var/log/system.log`). The value of the `IOKitDebug` property defines which events are logged; see `IOKitDebug.h` (available in `/System/Library/Frameworks/Kernel.framework/Headers/IOKit`) for the I/O Kit–defined values. Unless you are interested in only a single event or a particular set of events, set the value to `65535` to get information about all the events `IOKitDebug` covers. With this information, you can see which events occurred and whether they succeeded or failed, but not why they failed.

Although the `IOKitDebug` property can be useful during development, be sure to set its value to `0` (or remove the property altogether) before shipping your driver because a non-zero value will prevent your driver from loading during a safe-boot. For more information on safe booting, see the Loading Kernel Extensions at Boot Time section of *Kernel Extension Programming Topics*.

Two-Machine Debugging

For debugging a device driver or indeed any code that resides in the kernel, two computers are a necessity. Buggy kernel code has a nasty tendency to crash or hang a system, and so directly debugging that system is often impossible.

Two-machine debugging with `gdb` is the main pathway to finding bugs in driver code. This section takes you through the procedure for setting up two computers for debugging, offers a few tips on using `gdb` in kernel code, introduces you to the kernel debugging macros, and discusses techniques for finding bugs causing kernel panics and hangs.

Setting Up for Two-Machine Debugging

This section summarizes the steps required to set up two computers for kernel debugging. It draws heavily on the following documents, which you should refer to for more detailed information:

- The tutorial Hello Debugger: Debugging a Device Driver with GDB. (This tutorial is part of *Kernel Extension Programming Topics*.)
- The "When Things Go Wrong" section of Building and Debugging Kernels in *Kernel Programming Guide*. (This document is available on-line in Darwin Documentation.)

In two-machine debugging, one system is called the *target* computer and the other the *host* (or *development*) computer. The host computer is the computer that actually runs `gdb`. It is typically the computer on which your driver is developed—hence, it's also referred to as the development computer. The target computer is the system on which the driver to be debugged is run. Your host and target computers should be running

the same version of the Darwin kernel, or as close as possible to same version. (Of course, if you're debugging a panic-prone version of the kernel, you'll want the host computer to run the most recent stable version of Darwin.) For optimal source-level debugging, the host computer should have the source code of the driver, any kernel extensions related to your driver (such as its client or provider), and perhaps even the kernel itself (`/xnu`).

In order for two-machine debugging to be feasible, the following must be true:

- For versions of Mac OS X before Mac OS X version 10.2, both computers must be on the same subnet.
- You must have login access to both computers as an administrator (group `admin`), because you'll need root privileges to load your KEXT (you can use the `sudo` command).
- You must be able to copy files between the computers using FTP, `scp` (SSH), `rsync`, AFP, or similar protocol or tool.

Note: The following steps include instructions on setting up a permanent network connection via ARP (step 3). This is unnecessary if you are running Mac OS X v. 10.2 or later on both machines and if you set the NVRAM `debug` variable to 0x144 (as in step 1). This configuration allows you to set up two-machine debugging on two computers that are not necessarily on the same subnet. If you are running an earlier version of Mac OS X, however, you do need to follow step 3 and both computers must be on the same subnet.

When all this is in place, complete the following steps:

1. **Target** Set the NVRAM `debug` variable to `0x144`, which lets you drop into the debugger upon a non-maskable interrupt (NMI) and, if you're running Mac OS X v. 10.2 or later, lets you debug two computers not on the same subnet. You can use `setenv` to set the flag within Open Firmware itself (on PowerPC-based Macintosh computers), or you can use the `nvram` utility. For the latter, enter the following as root at the command line:

   a. `nvram boot-args="debug=0x144"`

   It's a good idea to enter `nvram boot-args` (no argument) first to get any current NVRAM variables in effect; then include these variables along with the debug flag when you give the `nvram boot-args` command a second time. Reboot the system.

   Note: If your target machine contains a PMU (for example, a PowerBook G4 or an early G5 desktop computer), you may find that it shuts down when you exit kernel debugging mode. One reason for this is that if a breakpoint causes kernel debugger entry in the middle of a PMU transaction, the PMU watchdog may trigger on a timeout and cause the machine to shut down. If you experience this, you may find that forcing the PMU driver to operate in polled mode fixes the problem. To do this, set the NVRAM variable `pmuflags` to `1`, as shown below:

   ```
   nvram boot-args="pmuflags=1"
   ```

   You can set the `pmuflags` variable separately, as shown above, or you can set it at the same time you set the `debug` variable, as shown below:

   ```
   boot-args="debug=0x144 pmuflags=1"
   ```

2. **Host** or **Target** Copy the driver (or any other kernel extension) to a working directory on the target computer.

3. **Host** Set up a permanent network connection to the target computer via ARP. The following example assumes that your test computer is `target.goober.com`:

```
$ ping -c 1 target.goober.com
ping results: ....
$ arp -an
target.goober.com (10.0.0.69): 00:a0:13:12:65:31
$ arp -s target.goober.com 00:a0:13:12:65:31
$ arp -an
target.goober.com (10.0.0.69) at00:a0:13:12:65:31 permanent
```

   This sequence of commands establishes a connection to the target computer (via `ping`), displays the information on recent connections ARP knows about (`arp -an`), makes the connection to the target computer permanent by specifying the Ethernet hardware address (`arp -s`), and issues the `arp -an` command a second time to verify this.

4. **Target** Create symbol files for the driver and any other kernel extensions it depends on. First create a directory to hold the symbols; then run the `kextload` command-line tool, specifying the directory as the argument of the `-s` option:

```
$ kextload -l -s /tmp/symbols /tmp/MyDriver.kext
```

   This command loads `MyDriver.kext` but, because of the `-l` option, doesn't start the matching process yet (that happens in a later step). If you don't want the driver to load just yet, specify the `-n` option along with the `-s` option. See "Using kextload, kextunload, and kextstat" (page 125) for the `kextload` procedure for debugging a driver's start-up code.

5. **Target** or **Host** Copy the symbol files to the host computer.

6. **Host** Optionally, if you want to debug your driver with access to all the symbols in the kernel, obtain or build a symboled kernel. For further information, contact Apple Developer Technical support. You can find the instructions for building the Darwin kernel from the Open Source code in the Building and Debugging Kernels in *Kernel Programming Guide*.

7. **Host** Run `gdb` on the kernel.

```
$ gdb /mach_kernel
```

   If you have a symboled kernel, specify the path to it rather than `/mach_kernel`. It is important that you run `gdb` on a kernel of the same version and build as the one that runs on the target computer. If the versions are different, you should obtain a symboled copy of the target's kernel and use that.

8. **Host** In `gdb`, add the symbol file of your driver.

```
(gdb) add-symbol-file /tmp/symbols/com.acme.driver.MyDriver.sym
```

   Add the symbol files of the other kernel extensions in your driver's dependency chain.

9. **Host** Tell `gdb` that you will be debugging remotely.

```
(gdb) target remote-kdp
```

10. **Target** Break into kernel debugging mode. Depending on the model of your target system, either issue the appropriate keyboard command or press the programmer's button. On USB keyboards, hold down the Command key and the Power button; on ADB keyboards, hold down the Control key and the Power button. If you're running Mac OS X version 10.4 or later, hold down the following five keys: Command, Option, Control, Shift, and Escape.

You may have to hold down the keys or buttons for several seconds until you see the "Waiting for remote debugger connection" message.

11. **Host** Attach to the target computer and set breakpoints.

```
(gdb) attach target.goober.com
(gdb) break 'MyDriverClass::WriteData(* char)'
(gdb) continue
```

Be sure you give the `continue` command; otherwise the target computer is unresponsive.

12. **Target** Start the driver running.

```
$ kextload -m -t /tmp/MyDriver.kext
```

The `-m` option starts the matching process for the driver. The `-t` option, which tells `kextload` to conduct extensive validation checks, is really optional here; ideally, your driver should have passed these checks during an earlier stage of debugging (see "Using kextload, kextunload, and kextstat" (page 125)). After starting the driver, perform the actions necessary to trigger the breakpoint.

13. **Host** When the breakpoint you set is triggered, you can begin debugging your driver using `gdb` commands. If you "source" the kernel debugging macros (see the following section, "Using the Kernel Debugging Macros" (page 134)), you can use those as well.

Using the Kernel Debugging Macros

Apple includes a set of kernel debugging macros as part of Darwin. They have been written by engineers with an intimate knowledge of how the Darwin kernel works. Although it is possible to debug driver code without these macros, they will make the task much easier.

The kernel debugging macros probe the internal structures of a running Mac OS X system in considerable depth. With them you can get summary and detailed snapshots of tasks and their threads in the kernel, including such information as thread priority, executable names, and invoked functions. The kernel debugging macros also yield information on the kernel stacks for all or selected thread activations, on IPC spaces and port rights, on virtual-memory maps and map entries, and on allocation zones. See Table 7-2 (page 135) for a summary of the kernel debugging macros.

Important: The kernel debugging macros described in this section will not work unless you have a symboled kernel. You can either build the Darwin kernel from the open source (see the Building and Debugging Kernels in *Kernel Programming Guide* for details) or you can refer to the Kernel Debug Kit, available at http://developer.apple.com/sdk, which includes a copy of the kernel debug macros.

You can obtain the kernel debugging macros from the Darwin Open Source repository. They are in the `.gdbinit` file in the `/xnu/osfmk` branch of the source tree. Because `.gdbinit` is the standard name of the initialization file for `gdb`, you might already have your own `.gdbinit` file to set up your debugging sessions. If this is the case, you can combine the contents of the files or have a "source" statement in one `.gdbinit` file that references the other file. To include the macros in a `.gdbinit` file for a debugging session, specify the following `gdb` command shortly after running `gdb` on `mach_kernel`:

```
(gdb) source /tmp/.gdbinit
```

(In this example, `/tmp` represents any directory that holds the copy of the `.gdbinit` file you obtained from the Open Source repository.) Because the kernel debugging macros can change between versions of the kernel, make sure that you use the macros that match as closely as possible the version of the kernel you're debugging.

Table 7-2 Kernel debugging macros

Macro	Description
`showalltasks`	Displays a summary listing of tasks
`showallacts`	Displays a summary listing of all activations
`showallstacks`	Displays the kernel stacks for all activations
`showallvm`	Displays a summary listing of all the VM maps
`showallvme`	Displays a summary listing of all the VM map entries
`showallipc`	Displays a summary listing of all the IPC spaces
`showallrights`	Displays a summary listing of all the IPC rights
`showallkmods`	Displays a summary listing of all the kernel extension binaries
`showtask`	Displays status of the specified task
`showtaskacts`	Displays the status of all activations in the task
`showtaskstacks`	Displays all kernel stacks for all activations in the task
`showtaskvm`	Displays status of the specified task's VM map
`showtaskvme`	Displays a summary list of the task's VM map entries
`showtaskipc`	Displays status of the specified task's IPC space
`showtaskrights`	Displays a summary list of the task's IPC space entries
`showact`	Displays status of the specified thread activation
`showactstack`	Displays the kernel stack for the specified activation
`showmap`	Displays the status of the specified VM map
`showmapvme`	Displays a summary list of the specified VM map's entries
`showipc`	Displays the status of the specified IPC space
`showrights`	Displays a summary list of all the rights in an IPC space
`showpid`	Displays the status of the process identified by PID
`showproc`	Displays the status of the process identified by a proc pointer
`showkmod`	Displays information about a kernel extension binary

Macro	Description
showkmodaddr	Given an address, displays the kernel extension binary and offset
zprint	Displays zone information
paniclog	Displays the panic log information
switchtoact	Switch thread context
switchtoctx	Switch context
resetctx	Reset context

A subset of the kernel debugging macros are particularly useful for driver writers: showallstacks, switchtoact, showkmodaddr, showallkmods, and switchtoctx. The output of showallstacks lists all tasks in the system and, for each task, the threads and the stacks associated with each thread. Listing 7-5 (page 136) shows the information on a couple tasks as emitted by showallstacks.

Listing 7-5 Example thread stacks shown by showallstacks

```
(gdb) showallstacks
...
task        vm_map      ipc_space  #acts   pid  proc        command
0x00c1e620  0x00a79a2c  0x00c10ce0    2     51  0x00d60760  kextd
            activation  thread      pri  state  wait_queue  wait_event
            0x00c2a1f8  0x00ccab0c   31  W      0x00c9fee8  0x30a10c <ipc_mqueue_rcv>
                        continuation=0x1ef44 <ipc_mqueue_receive_continue>
            activation  thread      pri  state  wait_queue  wait_event
            0x00c29a48  0x00cca194   31  W      0x00310570  0x30a3a0 <kmod_cmd_queue>
                kernel_stack=0x04d48000
                stacktop=0x04d4bbe0
                0x04d4bbe0  0xccab0c
                0x04d4bc40  0x342d8 <thread_invoke+1104>
                0x04d4bca0  0x344b4 <thread_block_reason+212>
                0x04d4bd00  0x334e0 <thread_sleep_fast_usimple_lock+56>
                0x04d4bd50  0x81ee0 <kmod_control+248>
                0x04d4bdb0  0x45f1c <_Xkmod_control+192>
                0x04d4be00  0x2aa70 <ipc_kobject_server+276>
                0x04d4be50  0x253e4 <mach_msg_overwrite_trap+2848>
                0x04d4bf20  0x257e0 <mach_msg_trap+28>
                0x04d4bf70  0x92078 <.L_syscall_return>
                0x04d4bfc0  0x10000000
                stackbottom=0x04d4bfc0

task        vm_map      ipc_space  #acts   pid  proc        command
0x00c1e4c0  0x00a79930  0x00c10c88    1     65  0x00d608c8  update
            activation  thread      pri  state  wait_queue  wait_event
           0x00ddaa50  0x00ddbe34   31  W      0x00310780  0xd608c8 <rld_env+10471956>
                        continuation=0x1da528 <_sleep_continue>
```

The typical number of stacks revealed by showallstacks runs into the dozens. Most of the threads associated with these stacks are asleep, blocked on continuation (as is that for the second task shown in the above example). Stacks such as these you can usually ignore. The remaining stacks are significant because they reflect the activity going on in the system at a particular moment and context (as happens when an NMI or kernel panic occurs).

Thread activations and stacks in the kernel—including those of drivers—belong to the task named `kernel_task` (under the `command` column). When you're debugging a driver, you look in the active stacks in `kernel_task` for any indication of your driver or its provider, client, or any other object it communicates with. If you add the symbol files for these driver objects before you begin the debugging session, the indication will be much clearer. Listing 7-6 (page 137) shows an active driver-related thread in `kernel_task` in the context of adjacent threads.

Listing 7-6 Kernel thread stacks as shown by `showallstacks`

```
activation  thread      pri  state  wait_queue  wait_event
0x0101ac38  0x010957e4   80  UW     0x00311510  0x10b371c <rld_env+13953096>
          continuation=0x2227d0 <_ZN10IOWorkLoop22threadMainContinuationEv>
activation  thread      pri  state  wait_queue  wait_event
0x0101aaf0  0x01095650   80  R
      stack_privilege=0x07950000
      kernel_stack=0x07950000
      stacktop=0x07953b90
      0x07953b90  0xdf239e4 <com.apple.driver.AppleUSBProKeyboard + 0x19e4>
      0x07953be0  0xe546694 <com.apple.iokit.IOUSBFamily + 0x2694>
      0x07953c40  0xe5a84b4 <com.apple.driver.AppleUSBOHCI + 0x34b4>
      0x07953d00  0xe5a8640 <com.apple.driver.AppleUSBOHCI + 0x3640>
      0x07953d60  0xe5a93bc <com.apple.driver.AppleUSBOHCI + 0x43bc>
      0x07953df0  0x2239a8 <_ZN22IOInterruptEventSource12checkForWorkEv+18>
      0x07953e40  0x222864 <_ZN10IOWorkLoop10threadMainEv+104>
      0x07953e90  0x2227d0 <_ZN10IOWorkLoop22threadMainContinuationEv>
      stackbottom=0x07953e90
activation  thread      pri  state  wait_queue  wait_event
0x0101b530  0x0101c328   80  UW     0x00311500  0x10b605c <rld_env+13963656>
          continuation=0x2227d0 <_ZN10IOWorkLoop22threadMainContinuationEv>
```

You can use `showallstacks` in debugging panics, hangs, and wedges. For instance, it might reveal a pair of threads that are deadlocked against each other or it might help to identify a thread that is not handling interrupts properly, thus causing a system hang.

Another common technique using the kernel debugging macros is to run the `showallstacks` macro and find the stack or stacks that are most of interest. Then run the `switchtoact` macro, giving it the address of a thread activation, to switch to the context of that thread and its stack. From there you can get a backtrace, inspect frames and variables, and so on. Listing 7-7 (page 137) shows this technique.

Listing 7-7 Switching to thread activation and examining it

```
(gdb) switchtoact 0x00c29a48
(gdb) bt
#0  0x00090448 in cswnovect ()
#1  0x0008f84c in switch_context (old=0xcca194, continuation=0, new=0xccab0c) at
/SourceCache/xnu/xnu-327/osfmk/ppc/pcb.c:235
#2  0x000344b4 in thread_block_reason (continuation=0, reason=0) at
/SourceCache/xnu/xnu-327/osfmk/kern/sched_prim.c:1629
#3  0x000334e0 in thread_sleep_fast_usimple_lock (event=0xeec500, lock=0x30a3ac,
interruptible=213844) at /SourceCache/xnu/xnu-327/osfmk/kern/sched_prim.c:626
#4  0x00081ee0 in kmod_control (host_priv=0xeec500, id=4144, flavor=213844,
data=0xc1202c, dataCount=0xc12048) at /SourceCache/xnu/xnu-327/osfmk/kern/kmod.c:602
#5  0x00045f1c in _Xkmod_control (InHeadP=0xc12010, OutHeadP=0xc12110) at
mach/host_priv_server.c:958
#6  0x0002aa70 in ipc_kobject_server (request=0xc12000) at
/SourceCache/xnu/xnu-327/osfmk/kern/ipc_kobject.c:309
#7  0x000253e4 in mach_msg_overwrite_trap (msg=0xf0080dd0, option=3, send_size=60,
```

```
rcv_size=60, rcv_name=3843, timeout=12685100, notify=172953600, rcv_msg=0x0,
scatter_list_size=0) at /SourceCache/xnu/xnu-327/osfmk/ipc/mach_msg.c:1601
#8  0x000257e0 in mach_msg_trap (msg=0xeec500, option=13410708, send_size=213844,
rcv_size=4144, rcv_name=172953600, timeout=178377984, notify=256) at
/SourceCache/xnu/xnu-327/osfmk/ipc/mach_msg.c:1853
#9  0x00092078 in .L_syscall_return ()
#10 0x10000000 in ?? ()
Cannot access memory at address 0xf0080d10
(gdb) f 4
#4  0x00081ee0 in kmod_control (host_priv=0xeec500, id=4144, flavor=213844,
data=0xc1202c, dataCount=0xc12048) at /SourceCache/xnu/xnu-327/osfmk/kern/kmod.c:602
602                     res = thread_sleep_simple_lock((event_t)&kmod_cmd_queue,
(gdb) l
597                 simple_lock(&kmod_queue_lock);
598
599                 if (queue_empty(&kmod_cmd_queue)) {
600                     wait_result_t res;
601
602                     res = thread_sleep_simple_lock((event_t)&kmod_cmd_queue,
603                                         &kmod_queue_lock,
604                                         THREAD_ABORTSAFE);
605                     if (queue_empty(&kmod_cmd_queue)) {
606                         // we must have been interrupted!
```

Remember that when use the `switchtoact` that you've actually changed the value of the stack pointer. You are in a different context than before. If you want to return to the former context, use the `resetctx` macro.

The `showallkmods` and `showkmodaddr` macros are also useful in driver debugging. The former macro lists all loaded kernel extensions in a format similar to the `kextstat` command-line utility (Listing 7-8 (page 138) shows a few lines of output). If you give the `showkmodaddr` macro the address of an "anonymous" frame in a stack, and if the frame belongs to a driver (or other kernel extension), the macro prints information about the kernel extension.

Listing 7-8 Sample output from the `showallkmods` macro

```
(gdb) showallkmods
kmod        address     size        id      refs    version  name
0x0ebc39f4  0x0eb7d000  0x00048000  71      0       3.2  com.apple.filesystems.afpfs
0x0ea09480  0x0ea03000  0x00007000  70      0       2.1  com.apple.nke.asp_atp
0x0e9e0c60  0x0e9d9000  0x00008000  69      0       3.0  com.apple.nke.asp_tcp
0x0e22b13c  0x0e226000  0x00006000  68      0       1.2  com.apple.nke.IPFirewall
0x0e225600  0x0e220000  0x00006000  67      0       1.2  com.apple.nke.SharedIP
0x0df5d868  0x0df37000  0x00028000  62      0       1.2  com.apple.ATIRage128
0x0de96454  0x0de79000  0x0001e000  55      3       1.3 com.apple.iokit.IOAudioFamily
...
```

Tips on Using gdb

If you hope to become proficient at I/O Kit driver debugging, you'll have to become proficient in the use of `gdb`. There's no getting around this requirement. But even if you are already familiar with `gdb`, you can always benefit from insights garnered by other driver writers from their experience.

Examining Computer Instructions

If you don't have symbols for a driver binary—and even if you do—you should try examining the computer instructions in memory to get a detailed view of what is going on in that binary. You use the `gdb` command `x` to examine memory in the current context; usually, `x` is followed by a slash ("/") and one to three parameters, one of which is `i`. The examine-memory parameters are:

- A repeat count
- The display format: `s` (string), `x` (hexadecimal), or `i` (computer instruction)
- The unit size: `b` (byte), `h` (halfword), `w` (word—four bytes), `g` (giant word—eight bytes)

For example, if you want to examine 10 instructions before and 10 instructions after the current context (as described in "Tips on Debugging Panics" (page 144)), you could issue a command such as:

```
(gdb) x/20i $pc -40
```

This command says "show me 20 instructions, but starting 40 bytes" (4 bytes per instruction) "before the current address in the program counter" (the `$pc` variable). Of course, you could be less elaborate and give a simple command such as:

```
(gdb) x/10i 0x001c220c
```

which shows you 10 computer instructions starting at a specified address. Listing 7-9 (page 139) shows you a typical block of instructions.

Listing 7-9 Typical output of the `gdb` "examine memory" command

```
(gdb) x/20i $pc-40
0x8257c <kmod_control+124>:     addi    r3,r27,-19540
0x82580 <kmod_control+128>:     bl      0x8d980 <get_cpu_data>
0x82584 <kmod_control+132>:     addi    r0,r30,-19552
0x82588 <kmod_control+136>:     lwz     r31,-19552(r30)
0x8258c <kmod_control+140>:     cmpw    r31,r0
0x82590 <kmod_control+144>:     bne+    0x825c0 <kmod_control+192>
0x82594 <kmod_control+148>:     mr      r3,r31
0x82598 <kmod_control+152>:     addi    r4,r27,-19540
0x8259c <kmod_control+156>:     li      r5,2
0x825a0 <kmod_control+160>:     bl      0x338a8
                                     <thread_sleep_fast_usimple_lock>
0x825a4 <kmod_control+164>:     lwz     r0,-19552(r30)
0x825a8 <kmod_control+168>:     cmpw    r0,r31
0x825ac <kmod_control+172>:     bne+    0x825c0 <kmod_control+192>
0x825b0 <kmod_control+176>:     addi    r3,r27,-19540
0x825b4 <kmod_control+180>:     bl      0x8da00 <fast_usimple_lock+32>
0x825b8 <kmod_control+184>:     li      r3,14
0x825bc <kmod_control+188>:     b       0x82678 <kmod_control+376>
0x825c0 <kmod_control+192>:     lis     r26,49
0x825c4 <kmod_control+196>:     li      r30,0
0x825c8 <kmod_control+200>:     lwz     r0,-19552(r26)
```

Needless to say, you need to know some assembler in order to make sense of the output of the examine-memory command. You don't need to be an expert in assembler, just knowledgeable enough to recognize patterns. For example, it would be beneficial to know how pointer indirection with an object looks in computer instructions. With an object, there are two indirections really, one to get the data (and that could be null) and one to an object's virtual table (the first field inside the object). If that field doesn't point to either

your code or kernel code, then there's something that might be causing a null-pointer exception. If your assembler knowledge is rusty or non-existent, you can examine the computer instructions for your driver's code that you know to be sound. By knowing how "healthy" code looks in assembler, you'll be better prepared to spot divergences from the pattern.

Breakpoints

Using breakpoints to debug code inside the kernel can be a frustrating experience. Often kernel functions are called so frequently that, if you put a breakpoint on a function, it's difficult to determine which particular case is the one with the problem. There are a few things you can do with breakpoints to ameliorate this.

- **Conditional breakpoints**. A conditional breakpoint tells `gdb` to trigger a breakpoint only if a certain expression is true. The syntax is `cond<breakpoint index> <expression>`. An example is the following:

  ```
  (gdb) cond 1 (num > 0)
  ```

  However, conditional breakpoints are very slow in two-machine debugging. Unless you're expecting the breakpoint expression to be evaluated only a couple dozen times or so, they are probably too tedious to rely on.

- **Cooperative breakpoints**. To speed things up you can use two breakpoints that cooperate with each other. One breakpoint is a conditional breakpoint set at the critical but frequently invoked function. The other breakpoint, which has a command list attached to it, is set at a point in the code which is only arrived at after a series of events has occurred. You initially disable the conditional breakpoint and the second breakpoint enables it at some point later where the context is more pertinent to the problem you're investigating (and so you don't mind the slowness of expression evaluation). The following series of `gdb` commands sets up both breakpoints:

  ```
  (gdb) cond 1 (num > 0)
  (gdb) disable 1
  (gdb) com 2
      enable 1
      continue
  end
  ```

  If this debugging is something you do frequently, you can put breakpoint-setup commands into a macro and put that macro in your `.gdbinit` file.

- **Breakpoints at dummy functions**. You can use the previous two techniques on code that you do not own. However, if it's your code that you're debugging, the fastest way to trigger a breakpoint exactly when and where you want is to use dummy functions. Consider the follow stripped code snippet:

  ```
  void dummy() {
      ;
  }

  void myFunction() {
      // ...
      if (num > 0)
          dummy();
      // ...
  }
  ```

The expression in `myFunction` is the same exact expression you would have in a conditional breakpoint. Just set a breakpoint on the dummy function. When the breakpoint is triggered, get a backtrace, switch frames, and you're in the desired context.

Single-Stepping

Single-stepping through source code does not necessarily take you from one line to the next. You can bounce around in the source code quite a bit because the compiler does various things with the symbols to optimize them.

There are two things you can do to get around this. If it's your code you're stepping through, you can turn off optimizations. Or you can single-step through the computer instructions in assembler because one line of source code typically generates several consecutive lines of assembler. So, if you find it hard to figure things out by single-stepping through source, try single-stepping through assembler.

To single-step in `gdb`, use the `stepi` command (`si` for short). You can get a better view of your progress if you also use the `display` command, as in this example:

```
(gdb) display/4i $pc
```

This displays the program counter and the next three instructions as you step.

Debugging Kernel Panics

You might be familiar with kernel panics: those unexpected events that cripple a system, leaving it completely unresponsive. When a panic occurs on Mac OS X, the kernel prints information about the panic that you can analyze to find the cause of the panic. On pre-Jaguar systems, this information appears on the screen as a black and white text dump. Starting with the Jaguar release, a kernel panic causes the display of a message informing you that a problem occurred and requesting that you restart your computer. After rebooting, you can find the debug information on the panic in the file `panic.log` at `/Library/Logs/`.

If you've never seen it before, the information in `panic.log` might seem cryptic. Listing 7-10 (page 141) shows a typical entry in the panic log.

Listing 7-10 Sample log entry for a kernel panic

```
Unresolved kernel trap(cpu 0): 0x300 - Data access DAR=0x00000058 PC=0x0b4255b4
Latest crash info for cpu 0:
   Exception state (sv=0x0AD86A00)
      PC=0x0B4255B4; MSR=0x00009030; DAR=0x00000058; DSISR=0x40000000; LR=0x0B4255A0;
      R1=0x04DE3B50; XCP=0x0000000C (0x300 - Data access)
      Backtrace:
         0x0B4255A0 0x000BA9F8 0x001D41F8 0x001D411C 0x001D6B90 0x0003ACCC
         0x0008EC84 0x0003D69C 0x0003D4FC 0x000276E0 0x0009108C 0xFFFFFFFF
      Kernel loadable modules in backtrace (with dependencies):
         com.acme.driver.MyDriver(1.6)@0xb409000
Proceeding back via exception chain:
   Exception state (sv=0x0AD86A00)
      previously dumped as "Latest" state. skipping...
   Exception state (sv=0x0B2BBA00)
      PC=0x90015BC8; MSR=0x0200F030; DAR=0x012DA94C; DSISR=0x40000000; LR=0x902498DC;
      R1=0xBFFFE140; XCP=0x00000030 (0xC00 - System call)
```

```
Kernel version:
Darwin Kernel Version 6.0:
Wed May  1 01:04:14 PDT 2002; root:xnu/xnu-282.obj~4/RELEASE_PPC
```

This block of information has several different parts, each with its own significance for debugging the problem.

- **The first line**. The single most important bit of information about a panic is the first line, which briefly describes the nature of the panic. In this case, the panic has something to do with a data access exception. The registers that appear on the same line as the message are the ones with the most pertinent data; in this case they are the DAR (Data Access Register) and the PC (Program Counter) registers. The registers shown on the first line vary according to the type of kernel trap. The hexadecimal code before the description, which is defined in `/xnu/osfmk/ppc_init.c`, indicates the exception type. Table 7-3 (page 142) describes the possible types of exceptions.

Table 7-3 Types of kernel exceptions

Trap Value	Type of Kernel Trap
`0x100`	System reset
`0x200`	Computer check
`0x300`	Data access
`0x400`	Instruction access
`0x500`	External interrupt
`0x600`	Alignment exception
`0x700`	Illegal instruction

- **The registers**. Under the first "Exception state" is a snapshot of the contents of the CPU registers when the panic occurred.
- **The backtrace**. Each hexadecimal address in the backtrace indicates the state of program execution at a particular point leading up to the panic. Because each of these addresses is actually that of the function return pointer, you need to subtract four from this address to see the instruction that was executed.
- **The kernel extensions**. Under "Kernel loadable modules in backtrace" are the bundle identifiers (`CFBundleIdentifier` property) of all kernel extensions referenced in the backtrace and all other kernel extensions on which these extensions have dependencies. These are the kernel extensions for which you'll probably want to generate symbol files prior to debugging the panic.
- **The other exception states**. Under "Proceeding back via exception chain:" are the previous exception states the kernel experienced, separated by snapshots of the contents of the CPU registers at the time the exceptions occurred. Most of the time, the first exception state (immediately following the first line of the panic log) gives you enough information to determine what caused the panic. Sometimes, however, the panic is the result of an earlier exception and you can examine the chain of exceptions for more information.
- **The kernel version**. The version of the Darwin kernel and, more importantly, the build version of the `xnu` project (the core part of the Darwin kernel). If you are debugging with a symboled kernel (as is recommended), you need to get or build the symboled kernel from this version of `xnu`.

General Procedure

There are many possible ways to debug a kernel panic, but the following course of action has proven fruitful in practice.

1. **Get as many binaries with debugging symbols as possible.**

   Make a note of all the kernel extensions listed under "Kernel loadable modules in backtrace". If you don't have debugging symbols for some of them, try to obtained a symboled version of them or get the source and build one with debugging symbols. This would include `mach_kernel`, the I/O Kit families, and other KEXTs that are part of the default install. You need to have the same version of the kernel and KEXT binaries that the panicked computer does, or the symbols won't line up correctly.

2. **Generate and add symbol files for each kernel extension in the backtrace.**

   Once you've got the kernel extension binaries with (or without) debugging symbols, generate relocated symbol files for each KEXT in the backtrace. Use `kextload` with the `-s` and `-n` options to do this; kextload prompts you for the load address of each kernel extension, which you can get from the backtrace. Alternatively, you can specify the `-a` option with `-s` when using `kextload` to specify KEXTs and their load addresses. Although you don't need to relocate symbol files for all kernel extensions, you can only decode stack frames in the kernel or in KEXTs that you have done this for. After you run `gdb` on `mach_kernel` (preferably symboled), use `gdb`'s `add-symbol-file` command for each relocatable symbol files you've generated; see "Setting Up for Two-Machine Debugging" (page 131) for details.

3. **Decode the addresses in the panic log.**

   Start with the PC register and possibly the LR (Link Register). (The contents of the LR should look like a valid text address, usually a little smaller than the PC-register address.) Then process each address in the backtrace, remembering to subtract four from each of the stack addresses to get the last instruction executed in that frame. One possible way to go about it is to use a pair of `gdb` commands for each address:

   ```
   (gdb) x/i <address>-4
   ...
   (gdb) info line *<address>-4
   ```

   You need the asterisk in front of the address in the `info` command because you are passing a raw address rather than the symbol `gdb` expects. The `x` command, on the other hand, expects a raw address so no asterisk is necessary.

   Listing 7-11 (page 143) gives an example of a symbolic backtrace generated from `x/i <address>-4`. You'll know you've succeeded when all the stack frames decode to some sort of branch instruction in assembler.

4. **Interpret the results.**

   Interpreting the results of the previous step is the hardest phase of debugging panics because it isn't mechanical in nature. See the following section, "Tips on Debugging Panics" (page 144), for some suggestions.

Listing 7-11 Example of symbolic backtrace

```
(gdb) x/i 0x001c2200-4
0x1c21fc <IOService::PMstop(void)+320>: bctrl
0xa538260 <IODisplay::stop(IOService *)+36>:    bctrl
```

```
0x1bbc34 <IOService::actionStop(IOService *, IOService *)+160>: bctrl
0x1ccda4 <runAction__10IOWorkLoopPFP8OSObjectPvn3_iPB2Pvn3+92>: bctrl
0x1bc434 <IOService::terminateWorker(unsigned long)+1824>:      bctrl
0x1bb1f0 <IOService::terminatePhase1(unsigned long)+928>:       bl
0x1bb20c <IOService::scheduleTerminatePhase2(unsigned long)>
0x1edfcc <IOADBController::powerStateWillChangeTo(unsigned long, unsigned long, IOService
 *)+88>:       bctrl
0x1c54e8 <IOService::inform(IOPMinformee *, bool)+204>: bctrl
0x1c5118 <IOService::notifyAll(bool)+84>:       bl
0x1c541c <IOService::inform(IOPMinformee *, bool)>
0x1c58b8 <IOService::parent_down_05(void)+36>:  bl
0x1c50c4 <IOService::notifyAll(bool)>
0x1c8364 <IOService::allowCancelCommon(void)+356>:      bl
0x1c5894 <IOService::parent_down_05(void)>
0x1c80a0 <IOService::serializedAllowPowerChange2(unsigned long)+84>:    bl
0x1c8200 <IOService::allowCancelCommon(void)>
0x1ce198 <IOCommandGate::runAction(int (*)(OSObject *, void *, void *, void *, void *),
 void *, void *, void *, void *)+184>: bctrl
0x1c802c <IOService::allowPowerChange(unsigned long)+72>:       bctrl
0xa52e6c ????
0x3dfe0 <_call_thread_continue+440>:    bctrl
0x333fc <thread_continue+144>:  bctrl
```

Tips on Debugging Panics

The following tips might make debugging a panic easier for you.

- As noted previously, always pay attention to the first line of the panic message and the list of kernel extensions involved in the panic. The panic message provides the major clue to the problem. With the kernel extensions, generate relocated symbol files for the debugging session.
- Don't assume that the panic is not your driver's fault just because it doesn't show up in the backtrace. Passing a null pointer to an I/O Kit family or any other body of kernel code *will* cause a panic in that code. Because the kernel doesn't have the resources to protect itself from null pointers, drivers must be extremely vigilant against passing them in.
- The `showallstacks` kernel debugging macro is very useful for debugging panics. See "Using the Kernel Debugging Macros" (page 134) for more information.
- Use `gdb`'s `$pc` variable when debugging panics with `gdb`. The `$pc` variable holds the value of the program counter, which identifies the place where the panic exception was taken. If you want to examine the context of the panic, you could issue a command such as:

  ```
  (gdb) x/20i $pc -40
  ```

  This displays 10 instructions in assembler before and after the point where the panic occurred. If you have the appropriate source code and symbols, you can enter:

  ```
  (gdb) l *$pc
  ```

  This shows the particular line of code that took the panic.
- If you have a panic caused by an Instruction Access Exception (`0x400`) and the PC register is zero, it means that something in the kernel branched to zero. The top frame in the stack is usually a jump through some function pointer that wasn't initialized (or that somehow got "stomped").

- Panics caused by a Data Access Exception `0x300` are quite common. These types of panics typically involve indirection through zero (in other words, a dereferenced null pointer). Any time a null pointer is dereferenced, a panic results. When you get a Data Access Exception, first check the DAR register; if the value is less than about 1000 the panic is probably the result of indirection through a null pointer. This is because most classes are no larger than about 1000 bytes and when an offset is added to a null pointer, the result is about 1000 or less. If the result is much larger, it's probable that the location of a pointer has been trashed (as opposed to its contents) and the contents of an unknown location is being used as a pointer in your code.

  A null pointer implies the possibility of a race condition with a shutdown or completion value. When called, a completion routine starts to free resources and if your driver is referencing these resources after the routine is called, it will probably get null pointers back.

- If you get a panic in `kalloc`, `kmem`, or `IOMalloc`, it suggests that you're using a freed pointer. Even if your driver code doesn't show up in the backtrace, your driver could be the culprit. Using a freed pointer is likely to break the kernel's internal data structures for allocation.

- Panics can also be caused by accidentally scribbling on someone else's memory. These "land mines" can be notoriously hard to debug; a backtrace doesn't show you much, except that something bad has happened. If the panic is reproducible, however, you have a chance to track down the offending code by using a "probe" macro. A probe macro helps to bracket exactly *where* in the code the scribbling happened. By definition, a scribble is a byte that does not have the expected contents (because they were altered by some other code). By knowing where the panic occurred and where the byte last held its expected value, you know where to look for the scribbling code.

  Often it's the case that your driver is the offending scribbler. To find your offending code (if any), define a probe macro that tests whether a memory address (`A` in the example below) has the expected value (`N`). (Note that `A` is the address in the DAR register of the original scribble panic.) If `A` doesn't have the expected value, then cause a panic right then and there:

```
Uint32 N = 123;
Uint32 *A;
A = &N;
// ...
#define PROBE() do {
    if (A && *A != N)
        *(Uint32)0 = 0;
    } while (0)
```

  By putting the probe macro in every function where `A` and `N` appear, you can narrow down the location of the scribble. If your driver is not the one doing the scribbling, it still might be indirectly responsible because it could be causing other code to scribble. For example, your driver might be asking some other code to write in your address space; if it's passed them the wrong address, it might result in data being scribbled on inside it.

Debugging System Hangs

System hangs are, after kernel panics, the most serious condition caused by badly behaved kernel code. A hung system may not be completely unresponsive, but it is unusable because you cannot effectively click the mouse button or type a key. You can categorize system hangs, and their probable cause, by the behavior of the mouse cursor.

- **Cursor doesn't spin and won't move**. This symptom indicates that a primary interrupt is not being delivered. The mouse doesn't even spin because that behavior is on a primary interrupt; its not spinning indicates that the system is in a very tight loop. What has probably happened is that a driver object has disabled an interrupt, causing code somewhere in the driver stack to go into an infinite loop. In other words, a piece of hardware has raised an interrupt but the driver that should handle it is not handling it and so the hardware keeps raising it. The driver has probably caused this "ghost" interrupt, but is unaware it has and so is not clearing it.
- **Cursor spins but won't move**. This symptom indicates that a high-priority thread such as a timer is spinning.
- **Cursor spins and moves, but nothing else**. This symptom suggests that the USB thread is still scheduled, thus indicating a deadlock in some driver object that is not related to USB or HI.

For system hangs with the first symptom—the cursor doesn't spin and won't move—your first aim should be to find out what caused the interrupt. Why is the hardware controlled by your driver raising the interrupt? If your driver is using a filter interrupt event source (IOFilterInterruptEventSource), you might want to investigate that, too. With a filter event source a driver can ignore interrupts that it thinks aren't its responsibility.

With any system hang, you should launch `gdb` on the kernel, attach to the hung system and run the `showallstacks` macro. Scan the output for threads that are deadlocked against each other. Or, if it's an unhandled primary interrupt that you suspect, find the running thread; if it is the one that took the interrupt, it is probably the thread that's gone into an infinite loop. If the driver is in an infinite loop, you can set a breakpoint in a frame of the thread's stack that is the possible culprit; when you continue and hit the breakpoint almost immediately, you know you're in an infinite loop. You can single-step from there to find the problem.

Debugging Boot Drivers

The Mac OS X BootX booter copies drivers for hardware required in the boot process into memory for the kernel's boot-time loading code to load. Because boot drivers are already loaded by the time the system comes up, you do not have as much control over them as you do over non-boot drivers. In addition, a badly behaving boot driver can cause your system to become unusable until you are able to unload it. For these reasons, debugging techniques for boot drivers vary somewhat from those for other drivers.

The most important step you can take is to treat your boot driver as a non-boot driver while you are in the development phase. Remove the `OSBundleRequired` property from your driver's `Info.plist` file and use the techniques described in this chapter to make sure the driver is performing all its functions correctly before you declare it to be a boot driver.

After you've thoroughly tested your driver, add the `OSBundleRequired` property to its `Info.plist` (see the document Loading Kernel Extensions at Boot Time to determine which value your driver should declare). This will cause the BootX booter to load your driver into memory during the boot process.

If your boot driver does have bugs you were unable to find before, you cannot use `gdb` to debug it because it is not possible to attach to a computer while it is booting. Instead, you must rely on `IOLog` output to find out what is happening. `IOLog` is synchronous when you perform a verbose boot so you can use `IOLog` statements throughout your boot driver's code to track down the bugs. See "Using IOLog" (page 147) for more information on this function.

To perform a verbose boot, reboot holding down both the `Command` and `V` keys. To get even more detail from the I/O Kit, you can set a `boot-args` flag before rebooting. Assuming `root` privileges with the `sudo` command, type the following on the command line

```
%sudo nvram boot-args="io=0xffff"
Password:
%shutdown -r now
```

Although this technique produces voluminous output, it can be difficult to examine because it scrolls off the screen during the boot process. If your boot driver does not prevent the system from completing the boot process, you can view the information in its entirety in the system log at `/var/log/system.log`.

Logging

Logging, the capture and display of information at certain points in the code, is useful in some situations as a debugging tool because it provides a linear record of what happens during the execution of driver code. This section describes the I/O Kit's primary logging function, `IOLog`, and discusses ways you can go about creating your own logging facility.

Using IOLog

It is natural to rely on `printf` statements to display what's going on in your application code during the debugging phase. For in-kernel drivers, the I/O Kit equivalent to the `printf` statement is the `IOLog` function, defined in `/System/Library/Frameworks/Kernel.framework/Headers/IOKit/IOLib.h`. Unfortunately, `IOLog` is only marginally useful for debugging kernel-resident drivers.

Because it executes inside the kernel, `IOLog` is necessarily resource-frugal. The message buffer is not large and if you use `IOLog` to log I/O activity, for example, the older messages are likely to be overwritten by newer ones, erasing the information you need for debugging. In general, you cannot successfully use `IOLog` in any tight loop in your driver's code, whether they are I/O loops or unintentional loops caused by errors in the code. The only way to ensure you get all the information you need in such a situation is to allocate your own buffer and create your own logging facility to write to it (for more information on this technique, see "Custom Event Logging " (page 148)).

If your driver is causing a panic, `IOLog` will not help much in determining the cause. Because the `IOLog` function is asynchronous, when the computer panics, the last `IOLog` message is still waiting to be scheduled for output so you will miss the message containing information about the cause of the panic.

You cannot call `IOLog` from an interrupt context. When you create an IOFilterInterruptEventSource object, you send it a filter function and an action function. You cannot use `IOLog` in your filter routine because it runs at primary (hardware) interrupt time when very little of the system is available. You can, however, call `IOLog` in your action routine because it runs on the work loop the event source is attached to. Note that when you create an IOInterruptEventSource object, you send it an action function that runs on the event source's work loop, but no filter function. As with the action function for an IOFilterInterruptEventSource object, you can call `IOLog` in the action function for an IOInterruptEventSource object.

Although the `IOLog` function is not well-suited for use in the debugging process, it is useful for logging normal status and error messages. For example, the PhantomAudioDevice class (contained in the PhantomAudioDriver project at `/Developer/Examples/Kernel/IOKit/Audio/PhantomAudioDriver`) uses `IOLog` to log status in several of its methods. Listing 7-12 (page 148) shows partial listings of PhantomAudioDevice's `createAudioEngines` and `volumeChanged` methods.

Listing 7-12 Using `IOLog` in PhantomAudioDevice methods

```
bool PhantomAudioDevice::createAudioEngines()
{
    ...
    IOLog("PhantomAudioDevice[%p]::createAudioEngines()\n", this);
    audioEngineArray = OSDynamicCast(OSArray,
                                getProperty(AUDIO_ENGINES_KEY));
    if (audioEngineArray) {
        /* Create new audio engine with AudioEngine array in personality. */
        ...
    }
    else {
        IOLog("PhantomAudioDevice[%p]::createAudioEngines() - Error: no
                            AudioEngine array in personality.\n", this);
        goto Done;
    }
    ...
}


IOReturn PhantomAudioDevice::volumeChanged(IOAudioControl *volumeControl,
                                        SInt32 oldValue, SInt32 newValue)
{
    IOLog("PhantomAudioDevice[%p]::volumeChanged(%p, %ld, %ld)\n", this,
                                    volumeControl, oldValue, newValue);
...
}
```

Custom Event Logging

You can pepper your code with `IOLog` or `printf` calls as a debugging technique, but, as noted in the previous section ("Using IOLog" (page 147)), this approach has its limitations. Some drivers are especially sensitive to any impact on performance, and require more reliable and finer-grained logging data. If your driver falls into this category, you can create your own logging facility.

There are various ways to implement a custom debug-logging solution that avoids the drawbacks of `IOLog`. You would need to allocate your own buffer for you log entries, implement functions and macros for writing those entries to the buffer, and devise some means for examining the contents of the buffer.

For purposes of illustration, this section looks at the logging feature of the AppleGMACEthernet driver. You can find the source code for this driver (including the logging tools) in the Darwin source code for Mac OS X version 10.2.7 and above, available at http://www.opensource.apple.com/darwinsource/index.html. To find the source code, select a version of Mac OS X equal to or greater than v10.2.7 and click Source (choose the source for the PPC version, if there's a choice). This displays a new page, which lists the open source projects available for the version of Mac OS X you've chosen. Scroll down to AppleGMACEthernet and click it to view the source. Be prepared to supply your ADC member name and password.

The AppleGMACEthernet driver implements a kernel-resident logging infrastructure as well as a user-space tool to print the formatted contents of a log buffer. The tool used by this driver prints a report to the standard output that begins with header information followed by time-stamped log entries. You can redirect the output to a file and then add comments to the entries in the file. Listing 7-13 (page 149) shows the first several seconds of the driver's life (along with comments).

Listing 7-13 Sample output of a custom event log

```
 0  1376a020  13779fe0  1376a020    [ffffffe0]
  27   38e9000         0         1


  20:   39  291159         0         0  eNet
  30:   39  291266   19aec90   181c200  Strt
  40:   39  293857   19aec90   18a4000  KeyL
  50:   39  293899       100       100  =txq
  60:   39  293903       100       100  =txe
  70:   39  293907        40        40  =rxe
  80:   39  293908       100   1000040  parm
  90:   39  307103      1000  135f0000  =TxR
  a0:   39  307111       400   198b000  =RxR
  b0:   39  307119   17f4400   19c8f00  arys
  c0:   39  307120   19aec90         0  AttI
  d0:   39  307232   19aec90         0  RwPM
  e0:   39  307247   19aec90         2  mx4d
  f0:   39  307249   19aec90         2  ip4d
 100:   39  307316   19aec90         1  Pwr!
 110:   39  307436   19aec90   198e400  cfig
 120:   39  307465   19aec90         0  AttD
 130:   39  307598   19aec90         0  powr
 140:   39  307599   19aec90         0  RegS
 150:   39  310366   19aec90         0  Exit    end of start method
 160:   46  944895   191d400   198e400  NetE
 170:   46  944899   191d400         0  Wake    wake up ethernet cell
 180:   46  944938         0         0  +Clk
 190:   46  945093   199a5c0  1ee21000  Adrs
 1a0:   46  945096   19c8f00   198b000  IRxR
 1b0:   46  946111   191d400  135f0000  ITxR
 1c0:   46  946127         3      1010  wReg
 1d0:   46  946135         0        ff  ChpI    chip initialization
 1e0:   46  946136         4      9050
```

The report header consists of two lines. The first line identifies any flag in effect and gives the start, ending, and current addresses of the log buffer. The second line indicates the period (in seconds) spanned by the log entries, shows the physical address of the buffer, and tallies the numbers of alerts and buffer wrap-arounds since the driver started up.

Each log entry is identified by an offset (in hexadecimal) from the start of the buffer and a two-part time-stamp. The first part is the time in seconds since the driver started executing and the second is the time in microseconds (actually nanoseconds right-shifted by 10 to closely approximate microseconds). The final column contains four-character codes with special significance to the AppleGMACEthernet driver. The intermediate two columns show parameters whose value depends on the context indicated by the code.

The AppleGMACEthernet driver (in the UniEnet class) implements logging in a way that minimizes its effect on performance. The major steps it undertakes are the following:

- It allocates a 64 kilobyte wrap-around buffer to hold the logging data and defines global pointers to that buffer.
- It implements two low-level functions to log events to the buffer, one for general events and the other for alerts; alerts are supersets of events which first perform the log, then the `IOLog`, and then whatever you customize it to do.
- For convenience, it also defines macros built on these functions.
- It inserts macro calls at critical points in the code.
- It implements a user-space tool to format and print the contents of the log buffer.
- It implements a user client to send the buffer data to the tool.

Listing 7-14 (page 150) shows the definitions of the event-log and alert functions and macros.

Listing 7-14 Definition of logging macros and functions

```
#if USE_ELG /* (( */
#define ELG(A,B,ASCI,STRING)    EvLog( (UInt32)(A), (UInt32)(B),
                                       (UInt32)(ASCI), STRING )
#define ALERT(A,B,ASCI,STRING)  Alert( (UInt32)(A), (UInt32)(B),
                                       (UInt32)(ASCI), STRING )
    void EvLog( UInt32 a, UInt32 b, UInt32 ascii, char* str );
    void Alert( UInt32 a, UInt32 b, UInt32 ascii, char* str );
#else /* ) not USE_ELG: (   */
#define ELG(A,B,ASCI,S)
#define ALERT(A,B,ASCI,STRING)  IOLog( "UniNEnet: %8x %8x " STRING "\n",
                                (unsigned int)(A), (unsigned int)(B) )
#endif /* USE_ELG )) */
```

If you're curious, see the UniEnet class source for the implementations of `EvLog` and `Alert`.

The AppleGMACEthernet driver calls the `ELG` and `ALERT` macros at appropriate points in the source code. In the example in Listing 7-15 (page 150), the `ELG` macro is called just after the `configureInterface` member function is invoked; the event code is `'cfig'` and the parameters are the addresses of the current thread and the IONetworkInterface object passed in. (The final string parameter is printed in alerts and via `IOLog` if the custom debugging feature is disabled.)

Listing 7-15 Calling the logging macro

```
bool UniNEnet::configureInterface( IONetworkInterface *netif ) {
                                   IONetworkData *nd;
    ELG( IOThreadSelf(), netif, 'cfig', "configureInterface" );
 // ...
```

The report shown in Listing 7-13 (page 149) includes a log entry from this macro call, illustrating how it might appear (it's the sixteenth one).

Although it might be convenient to have a tool that nicely prints the contents of the log buffer, there is considerable time and effort involved in developing this tool and the user-client object that must feed it the logging data. Instead of a user-space tool, you can examine the contents of the log buffer in `gdb`. The contents of the buffer, however, will require more effort on your part to interpret.

Testing and Deploying Drivers

You've created your driver and thoroughly debugged it. Now you're ready to test it and prepare it for installation on your customers's computers. This chapter outlines testing strategies and describes how to use the Mac OS X Package Maker application to create an installable package.

Testing Strategies

Now that you've debugged your driver and you know it's working perfectly, you should test it on as many differently configured computers and as many different operating system releases as you plan to support (or expect your customers to have).

Basic Quality Control

The first step in testing your driver is to perform a basic quality-control pass. Ideally, many of these checks should be part of your day-to-day development methodology:

- Are the permissions and ownership of the kernel extension correct? For security reasons, no component of a KEXT should be writable by any user other than the superuser. What this entails is that:
  - All files and folders in the KEXT, including the KEXT itself, must be owned by the root user (UID 0).
  - All files and folders in the KEXT, including the KEXT itself, must be owned by the wheel group (GID 0).
  - All folders in the KEXT, including the KEXT itself, must have permissions 0755 (octal) or `rwxr-xr-x` (as shown by `ls -l`).
  - All files in the KEXT must have permissions 0644 (octal) or `rw-r--r--` (as shown by `ls -l`). A KEXT is not the place to store a user-space executable.

  In your post-build and post-install scripts, you can set the correct ownership and permissions of kernel extensions using these shell script commands:

  ```
  /usr/sbin/chown -R root:wheel MyDriver.kext
  find MyDriver.kext -type d -exec /bin/chmod 0755 {} \;
  find MyDriver.kext -type f -exec /bin/chmod 0644 {} \;
  ```

  You can also copy the driver as root to a temporary location to give the kernel extension the proper ownership and permissions.
- Have you tested the driver for memory leaks? Does it unload properly?
  - Load the driver and then try to unload it (using `kextunload`); if it does not successfully unload, then the driver has live objects that have not been released. You'll need to track down the mismatched retain and release.

- Load the driver and periodically run the `ioclasscount` and `ioalloccount` tools on it. If you see growth in the count of instances or in a type of allocated memory, then your driver is leaking memory.
- Make a list of every class your driver directly defines and every class your driver directly manipulates (such as OSString). Create a command-line script that calls `ioclasscount` on all these classes (you don't have to call `ioclasscount` for each class, you can list all the classes on one line), set your system on a short sleep-wake cycle, and run the script every time your system sleeps and wakes. If you detect a leak (an increase in the count for a class), unload and reload your driver and perform the test again to make sure it's your driver that's leaking. In this way, you can find leaks that might otherwise take thousands of normal sleep-wake cycles to detect.

- Have you recently performed validation checks using `kextload -nt`? (See "Debugging Drivers" (page 121) for details.) These checks may reveal undeclared dependencies and other errors.
- Have you correctly set or incremented the version number for the kernel extension? This is especially important if you want an improved version of your driver to be selected during the matching process.

When you are ready to deploy the driver, build it using the Development build style (in Project Builder) to get the full range of debugging symbols. Save a copy of the driver in case you or another developer might ever need a symboled version of it. Then, to get the final product, strip the symbols from the driver using the `strip` tool. Note that you use the Development build style instead of the Deployment build style for the final build of the driver. The Deployment style introduces additional optimizations that do little to enhance driver performance and yet make it harder to match up symbols in the stripped and non-stripped versions of the binary.

Configuration Testing

Configuration testing aims to test the operation of your driver within a range of factors that stress-test its performance under various conditions, possibly revealing a latent bug. These factors include the following:

- **Multiprocessor systems**. With multiprocessor systems becoming more and more common, it is important to test your driver on such systems. Your driver code could be running on any processor at any moment. Consequently, testing your driver on a multiprocessor system—where preemption and rescheduling occur simultaneously on each processor—might reveal a race condition or a locking bug.
- **Memory configurations**. You should test your driver on systems ranging from those with 128 megabyte installed (the current minimum) up to those with 1 gigabyte or more installed (typical, for example, in video-production systems).
- **Computer models**. You should test your drivers on a wide variety of Power PC computers: G3 and G4, desktop and portable.
- **Other hardware**. You should also test your driver with varied and even extreme assortments of hardware attached. For example, if you're testing a driver for a USB device, you might test it with a dozen or so other USB devices attached.
- **Mac OS X releases**. Obviously, you should also test your driver on a range of Mac OS X versions that are currently in use.

Power-Management Testing

Many drivers implement some form of power management, responding to or initiating requests to sleep, power down, or power up. You should ensure this capability is thoroughly tested. One way to test power management is power cycling. Put a system with your driver installed on it asleep, then wake it up. Repeat this sequence many times (the more the better) to see if a bug in your power-management code surfaces.

If your driver is for a device that is "hot-swappable" (FireWire, USB, and PC Card), test the code in your driver that is responsible for handling device connection and (especially) device removal. You want to ensure that the driver stack is gracefully torn down when a device is disconnected. As with power-management testing, disconnect and reconnect the device many times to see if the sheer repetition of the event reveals a bug.

Other Testing Strategies and Goals

Here are a few more suggestions to help you test your driver:

- If your driver controls a Human Interface (HI) device or accepts input of any kind (such as an audio driver), be sure to exercise all possible types of input, including fringe cases. For example, for a keyboard driver you might want to test various combinations involving the Command, Control, Option, Escape, Function, and other keys. For audio drivers (that control audio input), you might want to test how the driver performs across a range of sound frequencies.
- Isolate and exercise layers of your driver. With a mass storage driver, for example, test how the driver performs at the file-system level with blocks of data that are above and below virtual-memory page size (4 kilobytes). Issue random sized read and write commands, and see how the driver behaves when a file-system check (`fsck`) is conducted.
- Last, but definitely not least, always consider your driver from the viewpoint of security. If your driver relies upon a user-space daemon that is `setuid`, make sure that the executable isn't world-writable. Verify that the installation process does not present opportunities for security infringements and, generally, that there is no way for a non-administrative user to gain control of the device.

If possible, you can let your Apple developer technical support contact know you're about to ship a driver. If you supply Apple with the `CFBundleIdentifier` and version that uniquely identifies your driver, Apple will know who to contact if a problem report appears that seems to point to your driver.

Packaging Drivers for Installation

Before you send your driver out the door, you should package it so it's easy for your customers to install. Mac OS X provides the Package Maker application (available in `/Developer/Applications` and a command-line version available in `/Developer/Applications/PackageMaker.app/Contents/MacOS/PackageMaker`) to create your package. The Package Maker application guides you through the creation of a package and creates the required files from information you enter in its editing windows, making the process almost automatic. The command-line tool requires you to supply the necessary files you've created yourself. For more detailed information on how to use Package Maker to create a package, see the Package Maker Help menu. For more information on using the command-line version, type `./PackageMaker -help` from the `/Developer/Applications/PackageMaker.app/Contents/MacOS` directory.

The following sections provide an overview of the components of a Package Maker package, describe what comprises a valid package, and outline how your package gets installed. If you use another package-making tool to create your package, make sure the resulting package contains all the required elements discussed in these sections.

Note: This chapter describes the Package Maker application available in Mac OS X version 10.2 and above. To create packages that can also be installed on earlier versions of Mac OS X, you should use an earlier version of Package Maker.

Package Contents

A Package Maker package is an `Info.plist`-based NSBundle object that contains the information the Installer requires to install software on a Mac OS X system. Package contents are separated into required global resources and optional localizable resources. Listing 8-1 (page 154) shows the format of a package named MyPackage.

Listing 8-1 PackageMaker format for MyPackage

```
MyPackage.pkg
    Contents
        Info.plist
        Archive.pax.gz
        Archive.bom
        Resources
            /* Optional, localizable package-description information */
            Description.plist

            /* Optional, localizable documents */
            License document
            ReadMe document
            Welcome document

            /* Optional scripts */
            preinstall script
            postinstall script
            preupgrade script
            postupgrade script
            preflight script
            postflight script

            /* Optional check tools */
            InstallationCheck tool
            VolumeCheck tool

            /* Optional, localizable background image */
            background

            /* Optional language project folders. If present, these */
            /* folders contain localized versions of Description.plist, */
            /* License, ReadMe, Welcome, and background files. */
            English.lproj
            Japanese.lproj
```

Package Maker creates the package's `Info.plist` file using the information you enter in the Info window. Information such as the default installation location and level of user authorization required allow you to customize the installation of your package's contents.

Note: If your files must be installed in a non-user location, such as `/System/Library/Extensions` for KEXTs, you must create your package with admin or root authorization.

The `Archive.bom` and `Archive.pax.gz` files are required files Package Maker creates for you using information you enter in the Files window. The `Archive.bom` file is a bill-of-materials file that describes the contents of the package to the Installer. The `Archive.pax.gz` file is a compressed archive containing the contents of a package. (In earlier versions of Package Maker, these filenames incorporated the package name, as in `MyPackage.bom` and `MyPackage.pax.gz`.)

The Resources folder contains documents, scripts, and tools you provide for the Installer to use during installation. With the exception of the `Description.plist` file (which contains the information you enter in the Description window), Package Maker does not create these files for you. You have complete control over the contents of these files, but you do have to follow Package Maker's naming conventions, or the Installer will not find and execute them.

You must name your localizable documents `License`.*extension*, `ReadMe`.*extension*, and `Welcome`.*extension* where *extension* is `html`, `rtf`, `rtfd`, `txt`, or no extension at all. In Mac OS X version 10.2, you can customize the background image the Installer displays when it opens the package. You must name your background picture file `background`.*extension*, where *extension* is `jpg`, `tif`, `tiff`, `gif`, `pict`, `eps`, or `pdf`.

The optional scripts must also follow Package Maker's naming conventions. You can perform a wide range of tasks in these scripts but you should take care not to overburden them or the installation speed will suffer. For example, you could create folders in a preflight script, but it would be better to set up the necessary folders in the archive. As a general rule, you should try to create your package so that it provides everything your software needs, using the scripts sparingly to perform dynamic tasks during the install process. If you do include scripts to run during installation, be sure to make them executable.

If you need to examine a computer's configuration prior to installation to determine, for example, if there is sufficient memory to run the Installer, or if some other software is present, you can provide an InstallationCheck tool. The Installer runs this tool (if it is present) when it first opens a package. As with the optional scripts, you must follow Package Maker's naming convention and you must make the tool executable. In addition, the InstallationCheck tool must return an integer error code. The error code is used as a bitfield that encodes information about the installation status that the Installer can interpret. The Installer uses some of the bits as an index into files of error messages, either predefined by the Installer or custom. If you choose to define your own set of error messages, you create a localizable `InstallationCheck.strings` file and place it in your localized `.lproj` folder.

The Installer generally gives the user a choice of volumes on which to install your package. If you need to disable one or more volume choices, you can provide a VolumeCheck tool (you must name the tool VolumeCheck and make it executable). The Installer runs this tool (if it is present) before it displays the Target Select window. Like InstallationCheck, VolumeCheck returns an integer result code that is treated as a bitfield. The Installer uses some bits to identify several properties about the status of VolumeCheck and others as an index into either a predefined or a custom error message file named `VolumeCheck.strings`.

A metapackage is simply a package of prebuilt packages. If, for example, you offer a suite of drivers, you can create a metapackage to contain them all and allow your customers to install the ones they choose. To create a metapackage, select New Meta Package from Package Maker's File menu.

A metapackage has the same components as a single package, with the addition of a list of subpackages (each of which must be a valid package). For specific differences in the `Info.plist` keys, see Package Maker's Package Format Notes.

Package Validation and Installation

Package Maker provides a validation tool you can run on your package to flag potential problems (to run it, select Validate Package in the Tools menu). You can run the tool on packages built with any version of Package Maker. The tool checks for the presence of required files and that the scripts are executable. A valid package must contain

- a *`.bom`* file
- a `.pax.gz` file unless the package is a receipt or has the Alternate Source Location key or the Proxy Package Location key set (these keys are defined in Package Maker's Package Format Notes)
- an `Info.plist` file (for packages built with the newest version of Package Maker) or `.info` file (for packages built with older versions of Package Maker)
- a `.sizes` file, if the package is built with an older version of Package Maker
- executable scripts and tools, if scripts and tools are present

A valid metapackage must contain

- a `.list` file, if the metapackage is built with an older version of Package Maker
- a package list array in its `Info.plist` file, if the metapackage is built with the newer version of Package Maker
- the subpackages listed in the package list array, if the metapackage is built with the newer version of Package Maker

When the Installer installs a single package, it performs the following steps:

1. Run the InstallationCheck script (if present).
2. Display the Welcome screen.
3. Display the ReadMe screen.
4. Display the License screen.
5. Run the VolumeCheck script (if present) for each available volume.
6. Display the volume selection screen.
7. Wait for the user to select the target volume and click Install.
8. Run the preflight script, if present.
9. Run the preinstall script (or preupgrade script, if this is an upgrade), if present.
10. Copy the files to the target drive (if this is an upgrade, some files may be copied to an intermediate directory first).

11. Run the postinstall script (or postupgrade script, if this is an upgrade), if present.
12. If this is an upgrade, copy the files in the intermediate directory to their final destination.
13. Copy the receipt to the target drive's Receipts folder.
14. Run the postflight script, if present.
15. Reboot or quit, depending on the package's flags.

The Installer performs a similar set of steps when installing a metapackage:

1. Run the InstallationCheck script, if present.
2. Display the Welcome screen.
3. Display the ReadMe screen.
4. Display the License screen.
5. Run the VolumeCheck script (if present) for each available volume.
6. Display the volume selection screen.
7. Wait for the user to select the target volume and click Install.
8. Perform the following steps for each subpackage
    a. Run preflight script, if present.
    b. Run the preinstall script (or preupgrade script, if this is an upgrade), if present.
    c. Copy the files to the target drive (if this is an upgrade, some files may be copied to an intermediate directory first).
    d. Run the postinstall script (or postupgrade script, if this is an upgrade), if present.
    e. If this is an upgrade, copy the files in the intermediate directory to their final destination.
    f. Run the postflight script, if present.
9. Reboot or quit, depending on the metapackage's flags.

If you've used another tool to create your package, you should examine the installed files carefully to make sure everything is where you want it to be.

Developing a Device Driver to Run on an Intel-Based Macintosh

This chapter provides an overview of some of the issues related to developing a universal binary version of an I/O Kit device driver. Before you read this chapter, be sure to read *Universal Binary Programming Guidelines, Second Edition*. That document covers architectural differences and byte-ordering formats and provides comprehensive guidelines for code modification and building universal binaries. Most of the guidelines in that document apply to kernel extensions as well as to applications.

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

- All targets in your Xcode project are native
- You develop your project in Xcode 2.2.1 or later

The Intel side of your universal I/O Kit driver must be built using GCC 4.0 and the Mac OS X 10.4u (Universal) SDK available in Xcode 2.2.1 or later. This allows your driver to load and run on Intel-based Macintosh computers running Mac OS X 10.4.4 or later.

To determine the compiler version and SDK to use when building the PowerPC side of your universal I/O Kit driver, decide which versions of Mac OS X you plan to target:

- If you're targeting versions of Mac OS X earlier than 10.4, use GCC 3.3 and the Mac OS X SDK representing the oldest version of Mac OS X you need to support.
- If you're targeting Mac OS X v10.4.x and you're using Xcode 2.2.x or Xcode 2.3, use GCC 4.0 and the Mac OS X 10.4.0 SDK (*not* the Mac OS X 10.4u (Universal) SDK). Note that this differs from universal applications, which can be built using the Mac OS X 10.4u (Universal) SDK for both architectures.
- If you're targeting Mac OS X v10.4.x and you're using Xcode 2.4 or later, you can define the `KPI_10_4_0_PPC_COMPAT` preprocessor symbol when building for the PowerPC architecture. Defining this symbol allows you to use the Mac OS X 10.4u (Universal) SDK for both PowerPC and Intel architectures.

For more details on building a universal binary version of your driver, including Xcode project settings, see Technical Note TN2163: Building Universal I/O Kit Drivers.

Note: For general guidance on debugging drivers, including the universal binary version of a device driver running on an Intel-based Macintosh computer, see "Debugging Drivers" (page 121) and Hello Debugger: Debugging a Device Driver With GDB.

Byte Swapping

As a device driver developer, you are keenly aware of the native byte-ordering format of the device with which you're working. This familiarity, coupled with the fact that the PowerPC processor uses big-endian byte ordering, may mean that your code makes byte-order assumptions that are not appropriate for a universal

binary. If, for example, a device driver communicates with a natively little-endian device (such as a USB device), it may perform little-to-big byte swaps because it was developed to run in a PowerPC-based Macintosh. Such hard-coded byte swaps will cause problems when the device driver is run in an Intel-based Macintosh.

To avoid these problems, search for hard-coded byte swaps in your device driver and replace them with the conditional byte-swapping macros defined in `libkern/OSByteOrder.h`. These macros are designed to perform the most efficient byte swap for the given situation. For example, using the `OSSwapConst` macros (and the `OSSwap*Const` variants) in `OSByteOrder.h` to swap constant values prevents byte-swapping at runtime.

Handling Architectural Differences

As described in *Universal Binary Programming Guidelines, Second Edition*, there are differences in the PowerPC and x86 architectures that can prevent code developed for one architecture from running properly in the other architecture. This section highlights two architectural differences that have consequences device driver developers need to be aware of.

In an Intel-based Macintosh, an integer divide-by-zero operation is fatal, whereas in a PowerPC-based Macintosh (in which the operation returns 0), it is not. Although it is not likely your device driver purposely performs a divide-by-zero operation, it may occur as a consequence of other operations. Examine your code for places where this might happen and add code to prevent it.

Device drivers often define software data structures that mirror hardware data structures. It's important to realize that some data types have one size in one architecture and a different size in the other. For example, the `bool` data type is one byte in the x86 architecture and four bytes in the PowerPC architecture. If your device driver uses a data structure that contains a `bool` value, consider replacing it with a fixed-size data type to avoid alignment problems.

Viewing Values in the Device Tree Plane

In a PowerPC-based Macintosh, Open Firmware populates the device tree plane of the I/O Registry with device hierarchy information. In an Intel-based Macintosh, the device tree plane may contain entirely different values. In fact, the device tree planes of different PowerPC-based Macintosh computers may contain different values.

Because you cannot predict what values an individual computer's device tree plane will contain, your device driver should not rely on the existence of specific values or locations. Instead, it should use APIs provided by I/O Kit classes, such as IORegistryEntry, to access values in the I/O Registry.

Interrupt Sharing in an Intel-Based Macintosh

It's important for your device driver to pass along any interrupts it might receive that aren't directed at it. In a PowerPC-based Macintosh each internal PCI slot has its own dedicated interrupt line, which means that a driver might be able to get away with ignoring interrupts that aren't destined for it, even though this is not

recommended. In an Intel-based Macintosh, however, it is likely that a driver will share an interrupt line with other drivers. Under these circumstances, ignoring interrupts for other drivers can cause problems for these drivers and for the user.

It's possible that your driver might receive a very large number of interrupts destined for other drivers. Besides being sure to pass them on, your driver should not consider the arrival of such interrupts as an event that requires some sort of error logging. If, for example, your driver uses `IOLog` to log every spurious interrupt it receives, it will quickly overflow the console log and may degrade system performance. (For more information on using `IOLog`, see "Using IOLog" (page 147).)

Using the OSSynchronizeIO Function

Your PowerPC device driver may use the `OSSynchronizeIO` function to, for example, enforce in-order execution of a write operation. In a PowerPC-based Macintosh, this function issues an `eieio` instruction. In an Intel-based Macintosh, however, this function does nothing because the x86 architecture supports strong memory ordering by default.

A universal binary version of a device driver that uses the `OSSynchronizeIO` function should continue to do so to ensure that it still runs correctly on a PowerPC-based Macintosh.

Accessing I/O Space

In a PowerPC-based Macintosh, PCI I/O space is mapped into memory. A PCI device driver accesses the I/O space using the `ioRead*` functions of the IOPCIDevice class (for more on these functions, see `IOPCIDevice` documentation in *Kernel Framework Reference*). In a PowerPC-based Macintosh, these functions translate into memory accesses and in an Intel-based Macintosh they translate into the appropriate x86 instructions. If you're developing a universal binary version of a PCI driver, therefore, you can use the `ioRead*` functions to access I/O space and your driver will run in both PowerPC-based and Intel-based Macintosh computers.

For other devices that use I/O space, however, such as an ATA controller, you may have to add x86-specific instructions that execute when the driver is running on an Intel-based Macintosh.

Debugging on an Intel-Based Macintosh

For the most part, debugging a device driver running on an Intel-based Macintosh computer is the same as debugging a driver running on a PowerPC-based Macintosh computer. Even though an Intel-based Macintosh computer uses EFI (extensible firmware interface) instead of the Open Firmware used in a PowerPC-based Macintosh computer, you still need to set the appropriate NVRAM (nonvolatile RAM) variables to enable two-machine debugging (for more information on two-machine debugging, see "Two-Machine Debugging" (page 131)). To change NVRAM variables on any Macintosh computer running Mac OS X, you use the `nvram` utility as `root` on the command line. For example, you can set the following kernel debug flag:

```
nvram boot-args="debug=0x14e"
```

Intel-based Macintosh computers also provide a way to store a persistent set of boot arguments on a particular machine. The set of boot arguments is *not* required for kernel debugging, but it can make it easier for you if you often perform netbooting or if you boot from different partitions and you'd like to have different boot arguments associated with each partition.

The boot arguments are contained in a property list file named `com.apple.Boot.plist`, which is located in `/Library/Preferences/SystemConfiguration`. The `com.apple.Boot.plist` file contains the following keys:

- Boot Graphics. This obsolete property was used to determine whether the Apple logo is drawn during booting. Instead, use the `-v` boot-argument flag to disable boot-time graphics.
- Kernel. The value of the Kernel property identifies the kernel to boot. The default value is `mach_kernel`, but you can use this property to select an alternate kernel, such as an experimental or debug kernel.
- Kernel Flags. The Kernel Flags property may contain any set of valid flags, such as `debug=0x144`, separated by spaces. By default, the value of this property is the empty string.

Note: The contents of the `com.apple.Boot.plist` file is auxiliary to the NVRAM variables; in fact, if you use the `nvram` command to set NVRAM variables, it will override the values of those variables in the Kernel Flags property.

If you perform netbooting, be sure to make a copy of the `com.apple.Boot.plist` file and place it in the same folder that contains the kernel on the netboot server. A typical development scenario is to have a kernel, an mkext file (a previously cached set of device drivers for hardware involved in the boot process), and a `com.apple.Boot.plist` file for each netboot configuration you work with.

Note: If you use the Startup Disk System Preference or the `bless` command from the command line, the NVRAM variables will be reduced to a list of boot arguments that are considered safe. This is because it is assumed that NVRAM values are global to all installations of Mac OS X on a computer, and some values might have unexpected results in different major releases of the operating system. If you regularly boot different versions of Mac OS X from different partitions, you should place in each partition a separate copy of the `com.apple.Boot.plist` file, containing the appropriate values.

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 folder. See also kernel extension

CFM Code Fragment Manager, the library manager and code loader for processes based on PEF (Preferred Executable Format) object files (Carbon).

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.

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.

cross-boundary I/O The transport of data across the boundary between the kernel and user space. In Mac OS X, cross-boundary I/O can be performed using I/O Kit family device interfaces, POSIX APIs, I/O Registry properties, or custom user clients.

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.

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 I/O Kit, an 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 I/O Kit, a process by which an application finds an appropriate device 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.

driver matching In I/O Kit, a process in which a nub, after discovering a specific hardware device, searches for the driver or drivers most suited to 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.

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).

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

gdb gdb is the GNU Debugger, a powerful source-level debugger with a command-line interface. Used in conjunction with kdp to perform two-machine debugging. See also kdp.

host computer Also called the development computer, the host computer runs the debugger in a two-machine debugging scenario. Most often, the host computer is also the computer on which the KEXT is being developed. See also two-machine debugging; target computer.

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.

internationalization The design or modification of a software product, including online help and documentation, to facilitate localization. Internationalization of software typically involves writing or modifying code to make use of locale-aware operating-system services for appropriate localized text input, display, formatting, and manipulation. See also localization.

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 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.

kdp (Kernel Debugger Protocol) The kernel shim used for communication with a remote debugger. See also gdb.

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 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.

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.

localization The adaptation of a software product, including online help and documentation, for use in one or more regions of the world, in addition to the region for which the original product was created. Localization of software can include translation of user-interface text, resizing of text-related graphical elements, and replacement or modification of user-interface images and sound. See also internationalization.

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.

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.

Mach-O The Mach object file format. Mach-O is the preferred object file format for Mac OS X. See also PEF.

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 and driver matching.

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.

memory leak A bug in a program that prevents it from freeing discarded memory and causes it to use increasing amounts of memory. A KEXT that experiences a memory leak may not be able to unload.

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.

NMI (Nonmaskable interrupt) An interrupt produced by a particular keyboard sequence or button. It can be used to interrupt a hung system and, in two-machine debugging, drop into the debugger.

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; driver.

nvram (Nonvolatile RAM) RAM storage that retains its state even when the power is off.

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 virtual memory.

panic An unrecoverable system failure detected by the kernel.

PEF Preferred Executable Format. An executable format understood by the Code Fragment Manager (CFM). See also Mach-O.

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`) 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.

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.

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 An module that can be dynamically added to a 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 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.

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.

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.

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.

process A BSD abstraction for a running program. A process's 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.

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.

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.

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; thread.

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.

target computer In two-machine debugging, the computer on which the KEXT being debugged is run. The target computer must have a copy of the KEXT being debugged. See also two-machine debugging; host computer.

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; thread.

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.

timer A kernel resource that triggers an event at a specified interval. The event can occur only once or can be recurring. A timer is an example of an event source for a work loop.

two-machine debugging A debugging process in which one computer (the host) runs the debugger (often gdb) and another computer (the target) runs the program being debugged. Debugging KEXTs requires two-machine debugging because a buggy KEXT will often panic the system, making it impossible to run the debugger on that computer. See also host computer; target computer.

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.

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 manger in Mac OS X

provides 32-bit (minimum) protected address space for each task and facilitates efficient sharing of that address space.

virtual table Often called a vtable, a virtual table is a structure in the header of every class object that contains pointers to the methods the class implements.

vram (Video RAM) A special form of RAM used to store image data for a computer display. Vram can be accessed for screen updates at the same time the video processor is providing new data.

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. 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.

XML (Extensible Markup Language) A simplified dialect of SGML (Standard Generalized Markup Language) that provides a metalanguage containing rules for constructing specialized markup languages.

Document Revision History

This table describes the changes to *I/O Kit Device Driver Design Guidelines*.

Date	Notes
2009-08-14	Changed links from KPI Reference to Kernel Framework Reference.
2007-03-06	Made minor corrections.
2006-11-07	Updated information on building a universal I/O Kit device driver.
2006-10-04	Added information on debugging on an Intel-based Macintosh and noted that KUNC APIs are unavailable to KEXTs that depend on KPIs.
2006-06-28	Made minor corrections.
2006-05-23	Made minor corrections.
2006-04-04	Clarified the location of the AppleGMACEthernet driver source code.
2005-12-06	Made minor corrections.
2005-10-04	Made minor bug fixes.
2005-09-08	Added a chapter on creating a universal binary version of an I/O Kit device driver.
2005-08-11	Made minor bug fix. Changed title from "Writing an I/O Kit Device Driver".
2005-04-29	Added description of new way to break into kernel debugging mode in Mac OS X v. 10.4.
2005-04-08	Fixed typos. Reorganized Introduction chapter and added note that Objective-C does not supply I/O Kit interfaces.
2004-05-27	Changed outdated links, removed references to `OSMetaClass::failModLoad()` method.
2003-10-10	Added information about changes in memory subsystem to support 64-bit architectures. Added a link to the AppleGMACEthernet driver source code.
2003-09-18	Added definition of `kAny` constant in code listing 4-20, corrected property-key name `device_type` to `device-type`, added description of the default implementation of `newUserClient` method.
2003-05-15	Added note that a symboled kernel is required for the kernel debugging macros to work. Added link to Kernel Debug Kit.
2002-11-01	First publication.

Document Revision History

Index

Numerals

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X