FireWire Device Interface Guide

Hardware & Drivers > FireWire

2007-02-08

Contents

3

Figures, Tables, and Listings

Introduction to FireWire Device Interface Guide

> **Note:** This document was previously titled *Working With FireWire Device Interfaces*.

The I/O Kit device interface mechanism provides applications with a means of communicating with hardware from outside the kernel. To communicate with FireWire devices, Mac OS X provides several device interfaces that are specific to different protocols and different types of communication. This document describes the range of device interfaces available in Mac OS X that your application can use to control a FireWire device or unit.

Who Should Read This Document?

You should read this document if you are developing an application that needs to communicate with or control a FireWire device. Although this document describes how Mac OS X supports FireWire devices in the kernel, it does not describe how to develop in-kernel drivers for them.

Before you read this document, you should be familiar with the I/O Kit and the device interface mechanism it provides. To learn more about the I/O Kit in general and device interfaces in particular, see the documents listed in See Also (page 8).

A detailed description of the FireWire specification is beyond the scope of this document—for more information see the 1394 Trade Association website at http://www.1394ta.org.

Organization of This Document

This document contains the following chapters:

- "FireWire on Mac OS X" (page 9) gives a brief overview of FireWire, presents basic information about in-kernel Mac OS X support for FireWire devices, and describes the three device interface libraries the IOFireWire family provides.

- "Accessing FireWire Devices From Applications" (page 17) describes the process of communicating with FireWire devices from applications, from finding the device or unit in the I/O Registry to getting the appropriate device interface.

- "Using the FireWire Device Interface Libraries" (page 23) uses specific code examples to explain how to use some of the IOFireWire family's device interfaces.

- "Document Revision History" (page 43) lists the changes to this document.

The chapter "Using the FireWire Device Interface Libraries" (page 23) uses projects in the FireWire SDK as a code base. The latest version of the SDK is available for download at http://developer.apple.com/hard-waredrivers/download. The complete SDK contains a large number of samples covering many different kinds

of hardware access from applications in a variety of programming languages (Objective-C using the Cocoa framework, C, and C++). In the interests of brevity, this document includes only fragments of some of the projects to illustrate the concepts it describes. Refer to the SDK for the complete versions of the code fragments used, in addition to other projects not described here.

See Also

Apple developer documentation includes several documents that cover device access and the I/O Kit. Some of these documents are listed below.

- *I/O Kit Fundamentals*
- *Accessing Hardware From Applications*
- *I/O Kit Framework Reference*

In addition to these documents, Apple maintains a website devoted to FireWire on Mac OS X, with links to the SDK and related topics, at http://developer.apple.com/hardwaredrivers/firewire.

Apple provides a FireWire mailing list on which you can post questions and discuss issues of interest to the FireWire community. You can also search the archives for helpful information. You can subscribe to the FireWire mailing list at http://lists.apple.com/mailman/listinfo/firewire.

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

FireWire on Mac OS X

FireWire is Apple's implementation of the IEEE 1394 High Performance Serial Bus. Fast, hot-pluggable, and flexible, FireWire is a digital interface that supports daisy-chaining and branching for true peer-to-peer communication.

This chapter gives you a brief overview of FireWire. Then, it explains how the I/O Kit represents FireWire devices and describes the device interface libraries the IOFireWire family provides.

FireWire Overview

The FireWire bus appears as a large, 64-bit memory-mapped address space with each device, represented by a node, occupying a specific address range. The high-order 16 bits of each address identify the node and the remaining 48 bits are for device-specific use.

Because of FireWire's hot-pluggable nature, the FireWire bus is by no means static. A bus reset occurs whenever nodes appear or disappear or when software initiates it. A bus reset signal erases the current topology of the bus and forces the nodes to reidentify and reconnect themselves. Although much of the logic to handle bus resets is at the device firmware level, your application must be ready to manage the affects of a bus reset at any point. In some cases, this may mean setting a flag in a command that requests it to retry after a bus reset. In others, however, you may have to perform significant device reconfiguration tasks in order to continue processing data.

FireWire supports two types of data transfer: Asynchronous and isochronous. Asynchronous transfer provides acknowledged, guaranteed delivery of data and is targeted to a specific node with an explicit address. If the data you're sending is not error-tolerant, such as data written to a disk drive, asynchronous transfer is appropriate.

Isochronous transfer, on the other hand, is broadcast in a one-to-many or one-to-one manner. Before a node begins to send or receive isochronous data, it requests a particular amount of bandwidth and one or more isochronous channels. A channel can have one transmitter of data (or talker) and any number of receivers (or listeners). Isochronous transfers do not allow for error-checking or retransmission but they do deliver data at a constant, real-time rate. If you're sending time-critical, error-tolerant data, such as a video stream, isochronous transfers are preferable.

As a medium for data transmission that defines how to move packets on the FireWire bus, FireWire does not specify any high-level communication protocol to use. Protocols are special sets of communication guidelines that simplify and standardize the data transmission between FireWire devices. Mac OS X supports user-space access for two protocols commonly used on FireWire: SBP-2 and AV/C.

The Serial Bus Protocol 2, or SBP-2, is a storage protocol used to efficiently transfer large amounts of data at high speed. Printers, scanners, and hard drives are typically SBP-2 devices. The SBP-2 protocol defines a data and command transaction entity called an ORB, or Operation Request Block. You use normal ORBs to send data and commands to your device and special management ORBs to perform device login and logout. The SBP-2 protocol sends ORBs using FireWire's asynchronous data transport mode. You can get more information about the SBP-2 specification at http://t10.org.

The Audio/Video Control, or AV/C, protocol defines a command set used to control devices such as video recorders and digital cameras. Like SBP-2, the AV/C protocol uses FireWire's asynchronous data transport mode. An AV/C command or response is first encapsulated in a Function Control Protocol (or FCP) frame. Then, a FireWire write or block write transaction transports the FCP frame to and from the device.

The AV/C protocol uses the concept of a plug to describe an end-point of a connection an AV/C unit implements to receive or transmit data. Plugs can be external, physical plugs on the device itself, internal, virtual plugs the AV/C unit implements, or serial bus plugs that are accessible through the plug control registers (or PCRs). For more information about the AV/C specification, see http://www.1394ta.org.

A Digital Video (or DV) device, such as a DV camcorder, implements a subunit of the AV/C specification. If you want to communicate with a DV camcorder, you should use the QuickTime APIs Mac OS X provides. QuickTime supplies APIs and components to encode and decode DV data, communicate with a DV device, and handle video digitizing. For more information on QuickTime, see Reference Library > QuickTime. In "Using the IOFireWireAVCLib" (page 35), this document describes how to communicate with AV/C devices that are not DV camcorders.

In-Kernel FireWire Device Support

In the kernel, several layers of objects represent each FireWire device. For each FireWire hardware interface on the Macintosh, such as FireWire OHCI (or Open Host Controller Interface), the IOFireWire family publishes an IOFireWireController object in the I/O Registry. The IOFireWireController object provides bus management services for the multiple devices and protocols that can exist on one FireWire hardware interface.

The IOFireWire family then tries to read the configuration ROM of each device on the bus. A device's configuration ROM (or config ROM) contains information such as device identification and addresses of various registers. For each device that responds with its bus information block, the IOFireWire family publishes an IOFireWireDevice object in the I/O Registry. The IOFireWireDevice object keeps track of the device's node ID (which can change with the dynamic reconfiguration of the bus) and it copies properties from the device's configuration ROM, such as the device's globally unique identification (or GUID), into its property list. Most importantly, however, the IOFireWireDevice object scans the configuration ROM for unit directories. For each unit directory it finds, it publishes an IOFireWireUnit object in the I/O Registry. Figure 1-1 shows the stack of objects instantiated for a FireWire unit.

Figure 1-1   In-kernel objects supporting a FireWire unit
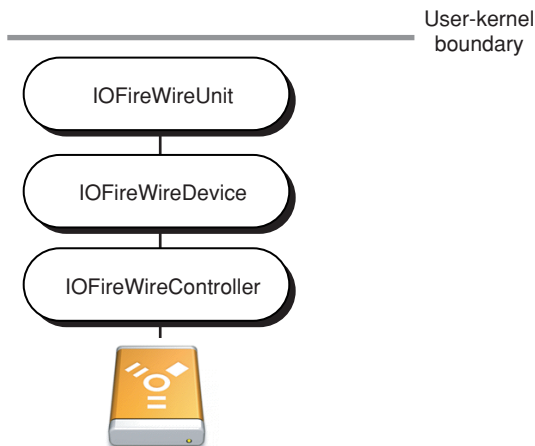
The IOFireWireUnit object copies properties from the device's unit into its property list in the I/O Registry. The I/O Kit uses these properties to enable matching on a specific unit specification ID and unit software version. When an SBP-2 unit appears in the I/O Registry, for example, the IOFireWireSBP2 kernel extension (or KEXT) matches on it and loads. The IOFireWireSBP2 family then publishes an IOFireWireSBP2Target object that scans the configuration ROM and publishes an IOFireWireSBP2LUN object for each logical unit (or LUN) it finds.

For AV/C units, the process is similar: The IOFireWireAVC KEXT matches an AV/C unit and publishes an IOFireWireAVCUnit object in the I/O Registry for it.

IOFireWire Family Device Interface Libraries

The same device and unit properties the I/O Kit uses for in-kernel driver matching are available to you for application-level device matching (described in "Device Matching for FireWire Devices" (page 17)). Because many devices, such as scanners, digital cameras, and printers, are better driven from applications, you can use the device interfaces the IOFireWire family provides to access them. Nearly all services available to in-kernel drivers are also available to an application through one of the IOFireWire family's device interfaces.

To communicate with FireWire devices or units from an application, the IOFireWire family provides three libraries that each include several device interfaces:

■ IOFireWireLib for standard FireWire commands and isochronous communication

■ IOFireWireSBP2Lib for sending ORBs to an SBP-2 unit and managing login status

> **Note:** If your device conforms to the SBP-2 specification and complies with the SCSI Architecture Model SCSI Primary Commands specification (both available at http://t10.org), you should consider using the SCSI Architecture Model family device interfaces, described in *SCSI Architecture Model Device Interface Guide*.

■ IOFireWireAVCLib for sending AV/C commands to an AV/C unit

The header files for all three libraries are part of `IOKit.framework`, located in `/System/Library/Frameworks`. In your application, you link against `IOKit.framework` to get access to these libraries.

IOFireWireLib Device Interfaces

The IOFireWireLib is a library of device interfaces that give applications access to both standard FireWire device functions and isochronous communication. The IOFireWireLib provides the lowest-level FireWire interfaces available in user space, allowing you to browse the configuration ROM of an external device, send commands, and perform FireWire bus operations. IOFireWireLib also gives you control of isochronous communication with a FireWire device by providing an interface that allows you to open a channel object and send and receive isochronous data. Figure 1-2 shows the interfaces available in the IOFireWireLib.

Figure 1-2 IOFireWireLib device interfaces

```
┌─ IOFireWireLib ────────────────────────────────────┐
│                                                      │
│   ┌─ IOFireWireDeviceInterface ──────────────────┐  │
│   │                                               │  │
│   │     IOFireWireReadCommandInterface            │  │
│   │     IOFireWireWriteCommandInterface           │  │
│   │     IOFireWireCompareSwapCommandInterface     │  │
│   │     IOFireWirePseudoAddressSpaceInterface     │  │
│   │     IOFireWirePhysicalAddressSpaceInterface   │  │
│   │     IOFireWireLocalUnitDirectoryInterface     │  │
│   │     IOFireWireConfigDirectoryInterface        │  │
│   │     IOFireWireIsochChannelInterface           │  │
│   │     IOFireWireLocalIsochPortInterface         │  │
│   │     IOFireWireRemoteIsochPortInterface        │  │
│   │                                               │  │
│   └───────────────────────────────────────────────┘  │
│                                                      │
└──────────────────────────────────────────────────────┘
```

In order to use the other interfaces in the IOFireWireLib, you must first get an instance of the primary interface, IOFireWireDeviceInterface. The IOFireWireDeviceInterface includes functions that allow you to communicate directly with a device and methods that create the other interfaces listed in Figure 1-2. You can get an IOFireWireDeviceInterface for a FireWire device or even the local node (defined to be the Macintosh itself). Figure 1-3 shows how an application uses an instance of the IOFireWireDeviceInterface to communicate with a device on the FireWire bus.

Figure 1-3 Using the IOFireWireDeviceInterface

```
        ┌───────────────────┐
        │    Application     │
        └───────────────────┘
                  │
        ┌───────────────────────┐
        │ IOFireWireDeviceInterface │
        └───────────────────────┘
                  │                        User-kernel
 ─────────────────┼──────────────────────   boundary
                  │
        ┌───────────────────┐
        │ IOFireWireUserClient │
        └───────────────────┘
                  │
        ┌───────────────────┐
        │   IOFireWireUnit   │
        └───────────────────┘
                  │
        ┌───────────────────┐
        │  IOFireWireDevice  │
        └───────────────────┘
                  │
        ┌───────────────────┐
        │ IOFireWireController │
        └───────────────────┘
```
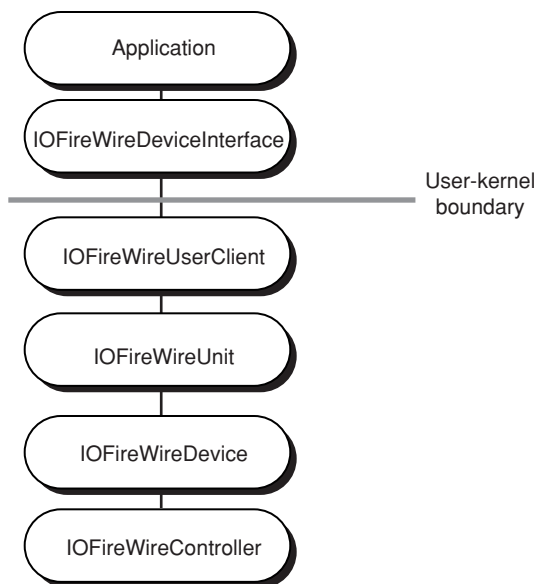
When you have an instance of the IOFireWireDeviceInterface, you can:

■ Perform a FireWire bus reset

■ Create FireWire command object interfaces to perform asynchronous read, write and lock operations

■ Create an isochronous interface that provides isochronous services, such as creating and managing isochronous channels and sending and receiving isochronous data

■ Create other interfaces that provide miscellaneous services, such as managing local unit directories in the Macintosh and accessing and browsing remote device configuration ROMs

You can also use the IOFireWireLib services and interfaces in conjunction with interfaces from the other two FireWire device interface libraries. Figure 1-5 (page 14) and Figure 1-7 (page 16) show how an application can use the IOFireWireDeviceInterface in cooperation with interfaces from other libraries. For more information on how to do this, see "Getting Multiple FireWire Device Interfaces" (page 21). For code samples that illustrate how to use various FireWire device interfaces, see "Using the FireWire Device Interface Libraries" (page 23).

IOFireWireSBP2Lib Device Interfaces

The IOFireWireSBP2Lib is a library of interfaces and functions that allow you to control the SBP-2 functions of your device. Figure 1-4 shows the interfaces of the IOFireWireSBP2Lib.

Figure 1-4   IOFireWireSBP2Lib interfaces
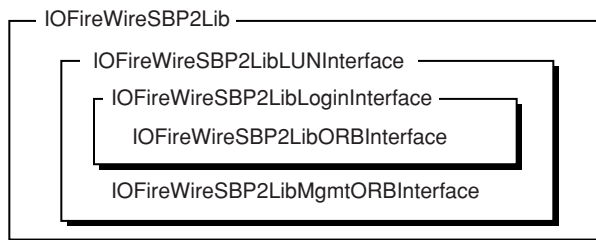
To use the SBP-2 functions the IOFireWireSBP2Lib provides, you first acquire the primary interface, IOFireWireSBP2LibLUNInterface, which supplies the methods that control the operation of the logical unit (or LUN) as a whole. When you have the IOFireWireSBP2LibLUNInterface, you can get two other interfaces:

■ The IOFireWireSBP2LibLoginInterface, which supplies methods that control the behavior and execution of an SBP-2 login session, including the execution of ORBs

■ The IOFireWireSBP2LibMgmtORBInterface, which provides methods that configure and append nonlogin–related management functions

The IOFireWireSBP2LibLoginInterface itself provides another interface, the IOFireWireSBP2LibORBInterface, that supplies the methods for configuring normal command ORBs.

The IOFireWireSBP2Lib handles device communication at the SBP-2 protocol level. In other words, it is concerned mostly with sending and receiving ORBs and managing login sessions. It does not include interfaces that allow you to communicate directly with the device. If you need to send commands to the device itself, for example, to read the configuration ROM or to service additional functionality beyond the scope of SBP-2, you must use the IOFireWireDeviceInterface of the IOFireWireLib (for more information on how to do this, see "Getting Multiple FireWire Device Interfaces" (page 21)).

Figure 1-5 shows an application using the IOFireWireSBP2LibLUNInterface to communicate with the logical unit of a FireWire SBP-2 device and, optionally, using the IOFireWireDeviceInterface to communicate with the device.

Figure 1-5   Using the IOFireWireSBP2LibLUNInterface and IOFireWireDeviceInterface
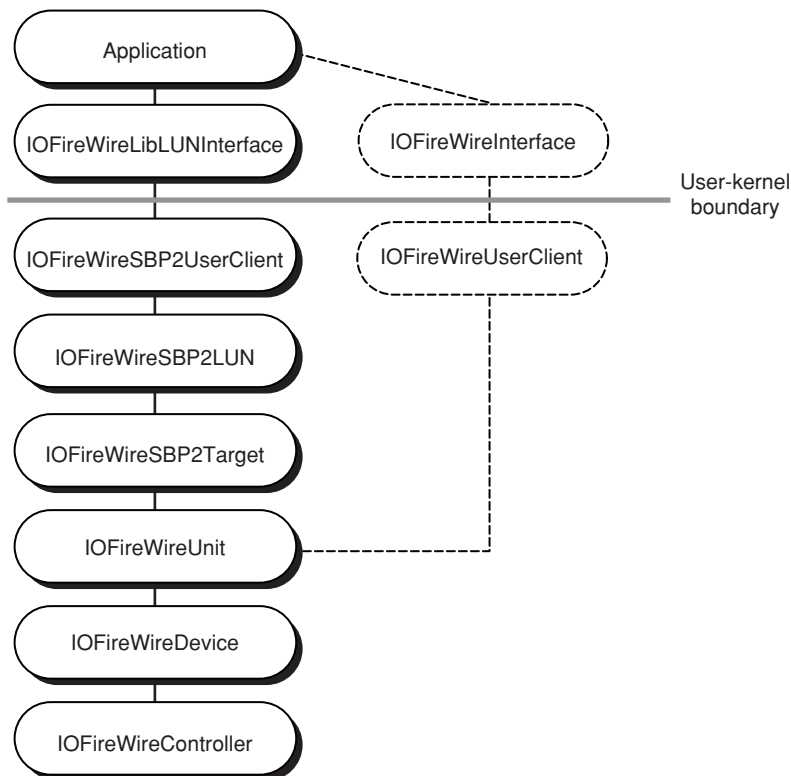
IOFireWireAVCLib Device Interfaces

The IOFireWireAVCLib is a library of interfaces and functions that you can use to send AV/C commands to an AV/C unit. Figure 1-6 shows the interfaces in the IOFireWireAVCLib.

Figure 1-6   IOFireWireAVCLib interfaces
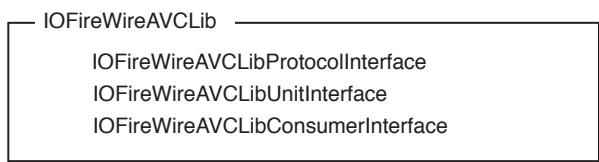
The IOFireWireAVCLib supplies two interfaces to handle AV/C units and commands: the IOFireWireAVCLibProtocolInterface and the IOFireWireAVCLibUnitInterface. Unlike the device interfaces in the IOFireWireLib and IOFireWireSBP2Lib, these device interfaces are not dependent on each other in any way. You can get either the IOFireWireAVCLibProtocolInterface or the IOFireWireAVCLibUnitInterface or both, according to what you need to accomplish. Because they are both primary interfaces, you do not need to get one before the other.

In addition, the IOFireWireAVCLib supplies a limited interface that supports a subset of asynchronous connection functionality. The IOFireWireAVCLibConsumerInterface is based on the assumption that the controller node is built into the consumer node, and that this dual entity is implemented in the Macintosh.

The functions in the IOFireWireAVCLibConsumerInterface support creating an asynchronous connection to a producer, sending commands to the producer, and receiving data from the producer. The IOFireWireAVCLibConsumerInterface does not support any other configuration of the consumer, controller, and producer nodes and does not allow the consumer/controller pair to receive commands.

The IOFireWireAVCLibProtocolInterface allows your application to treat the Macintosh as an AV/C device and access its plug control registers (or PCRs). To access the PCRs of an external FireWire device, you need to use the IOFireWireDeviceInterface. Because the IOFireWireAVCLibProtocolInterface is specifically for accessing the Macintosh as an AV/C device, you can open it only on the local node. With the IOFireWireAVCLibProtocolInterface, you can:

■ Allocate and deallocate input and output plugs

■ Read the current value of input and output plugs

■ Register callbacks for AV/C commands sent to the Macintosh and for bus reset and reconnect messages

■ Update the value of input and output plugs, simulating lock transactions

The IOFireWireAVCLibUnitInterface supplies functions that issue AV/C commands to an AV/C unit. Beginning in Mac OS X v10.4, the IOFireWireAVCLibUnitInterface includes functions that support asynchronous AV/C commands (not to be confused with AV/C asynchronous connections). You can use the asynchronous command functions to, for example, receive notifications when a user adjusts a switch on the device you're communicating with. Because the asynchronous commands are part of the IOFireWireAVCLibUnitInterface, you do not have to acquire a separate interface object to use them.

As with the IOFireWireSBP2LibLUNInterface, you can use both an IOFireWireAVCLibUnitInterface and an IOFireWireDeviceInterface to control the AV/C unit as well as the device object. For more information on how to do this, see "Getting Multiple FireWire Device Interfaces" (page 21).

Figure 1-7 shows an application using the IOFireWireAVCLibUnitInterface, the IOFireWireAVCLibProtocolInterface, and, optionally, the IOFireWireDeviceInterface.

Figure 1-7   Using the IOFireWireAVCLib interfaces and IOFireWireDeviceInterface

```
                              ┌─────────────────┐
                              │   Application   │
                              └─────────────────┘

┌──────────────────────────┐ ┌──────────────────────────┐ ┌──────────────────────┐
│IOFireWireAVCLibProtocol  │ │IOFireWireAVCLibUnit      │ │  IOFireWireInterface │
│Interface                 │ │Interface                 │ └──────────────────────┘
└──────────────────────────┘ └──────────────────────────┘
─────────────────────────────────────────────────────────────     User-kernel
                                                                    boundary
┌──────────────────────────┐ ┌──────────────────────────┐ ┌──────────────────────┐
│IOFireWireAVCProtocol     │ │  IOFireWireAVCUserClient │ │ IOFireWireUserClient │
│UserClient                │ │                          │ └──────────────────────┘
└──────────────────────────┘ └──────────────────────────┘
                             ┌──────────────────────────┐
                             │    IOFireWireAVCUnit     │
                             └──────────────────────────┘
                             ┌──────────────────────────┐
                             │     IOFireWireUnit       │
                             └──────────────────────────┘
┌──────────────────────────┐ ┌──────────────────────────┐
│  IOFireWireLocalNode     │ │    IOFireWireDevice      │
└──────────────────────────┘ └──────────────────────────┘
                             ┌──────────────────────────┐
                             │   IOFireWireController   │
                             └──────────────────────────┘
```
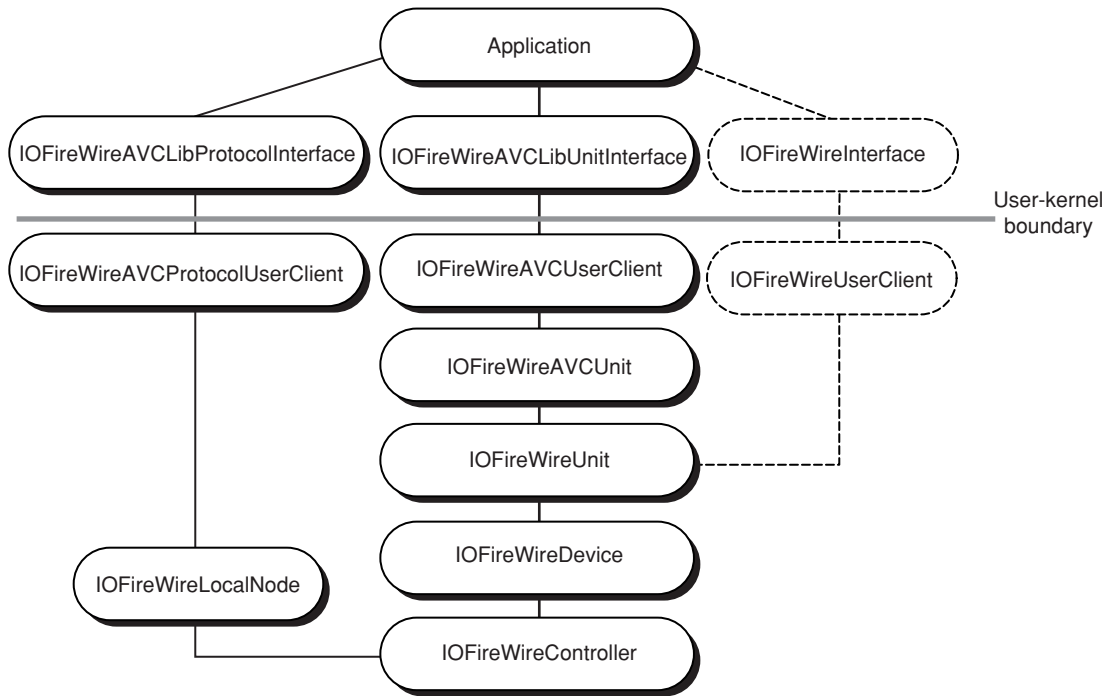
Accessing FireWire Devices From Applications

The device interfaces the FireWire family provides are plug-ins that supply functions your application can call to communicate with or control a device. This chapter describes how to find a FireWire device and get one or more device interfaces to communicate with it.

Device Matching for FireWire Devices

All user-level access of a device begins with a Mach port to communicate with the I/O Kit. You get a Mach port by calling the I/O Kit function `IOMasterPort`, as in the following example:

```
kern_return_t    result;
mach_port_t      masterPort;

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

In order to find a device, you create a matching dictionary that describes the device or device type you're interested in. The I/O Kit function `IOServiceMatching` creates a dictionary that matches on a given class name, as in this example:

```
CFMutableDictionaryRef  matchingDictionary = IOServiceMatching
                                        ( "IOFireWireDevice" );
```

In this example, you can use `matchingDictionary` to find all devices currently represented by an IOFireWireDevice object in the I/O Registry. Table 2-1 (page 17) lists the FireWire class names you can use to create a matching dictionary.

Table 2-1   FireWire class names

Class name	Definition
IOFireWireSBP2Target	A unit in a device that is an SBP-2 target
IOFireWireSBP2LUN	A LUN in an SBP-2 target
IOFireWireAVCUnit	An AV/C unit in a device
IOFireWireUnit	A unit in a device
IOFireWireDevice	A device on the FireWire Bus
IOFireWireLocalNode	The Macintosh itself

If you do not need to perform a more specific search, the dictionary `IOServiceMatching` creates will be sufficient to find all devices of a particular class currently in the I/O Registry.

If you want to refine your search, you can add properties to the dictionary to describe a unique device. As described in "In-Kernel FireWire Device Support" (page 10), the IOFireWire family places properties that describe each device and unit into the objects that represent them in the I/O Registry. You can use any of these properties to narrow down your search for a specific device or unit. Table 2-2 (page 18) shows the properties available in each object.

Table 2-2 FireWire class properties

Class name	Property key	Definition	Type
IOFireWireDevice	`FireWire Node ID`	Current node ID	`CFNumber`
	`Vendor_ID`	Device vendor ID	`CFNumber`
	`FireWire Device ROM`	Currently read contents of config ROM (may not contain entire ROM contents)	`CFData`
	`FireWire Vendor Name`	Device vendor name	`CFString`
	`FireWire Self IDs`	Self IDs received from device	`CFData`
	`FireWire Speed`	Current operating speed of device	`CFNumber`
	`GUID`	Globally unique ID of device	`CFNumber`
IOFireWireUnit	`Vendor_ID`	Device vendor ID	`CFNumber`
	`FireWire Vendor Name`	Device vendor name	`CFString`
	`GUID`	Globally unique ID of unit	`CFNumber`
	`Unit_Spec_ID`	Unit spec ID for this unit	`CFNumber`
	`Unit_SW_Version`	Unit software version of unit	`CFNumber`
	`FireWire Product Name`	Product name of device	`CFString`
IOFireWireSBP2Target	`Command_Set`	SBP2 command set	`CFNumber`
	`Command_Set_Revision`	SBP2 command set revision	`CFNumber`
	`Unit_Spec_ID`	Unit spec ID for FireWire unit provider of this SBP-2 target	`CFNumber`
	`Command_Set_Spec_ID`	SBP-2 command set spec ID	`CFNumber`
	`GUID`	Globally unique ID for this device	`CFNumber`
	`Vendor_ID`	Device vendor ID	`CFNumber`
	`Device_Type`	SBP-2 device type	`CFNumber`
	`Firmware_Revision`	SBP-2 firmware revision	`CFNumber`
	`Unit_SW_Version`	Unit software version for this unit	`CFNumber`

Class name	Property key	Definition	Type
IOFireWireSBP2LUN	Vendor_ID	Device vendor ID	CFNumber
	Command_Set	SBP-2 command set	CFNumber
	Command_Set_Spec_ID	SBP-2 command set spec ID	CFNumber
	GUID	Globally unique ID for this device	CFNumber
	Device_Type	SBP-2 device type	CFNumber
	Firmware_Revision	SBP-2 firmware revision	CFNumber
	IOUnit	SBP-2 LUN number	CFNumber
IOFireWireAVCUnit	Unit_Spec_ID	Unit spec ID for this unit	CFNumber
	Unit_Type	AV/C unit type	CFNumber
	Vendor_ID	Device vendor ID	CFNumber
	GUID	Globally unique ID for this device	CFNumber
	Unit_SW_Version	Unit software version for this unit	CFNumber
	FireWire Product Name	Name of this product	CFString

To use these properties for matching, start with a dictionary from `IOServiceMatching` and use Core Foundation functions to add property key-value pairs to it. Listing 2-1 (page 19) shows a dictionary that matches on an IOFireWireUnit object with specific `Unit_Spec_ID` and `Unit_SW_Version` values:

Listing 2-1   A matching dictionary for an IOFireWireUnit object

```
CFMutableDictionaryRef  matchingDictionary =
                                IOServiceMatching("IOFireWireUnit" );

UInt32      value;
CFNumberRef cfValue;

value = myFireWireUnitSpecID;
cfValue = CFNumberCreate( kCFAllocatorDefault, kCFNumberSInt32Type, &value );
CFDictionaryAddValue( matchingDictionary, CFSTR( "Unit_Spec_ID" ), cfValue );
CFRelease( cfValue );

value = myFireWireUnitSwVersionID;
cfValue = CFNumberCreate( kCFAllocatorDefault, kCFNumberSInt32Type, &value );
CFDictionaryAddValue( matchingDictionary, CFSTR( "Unit_SW_Version" ),
                    cfValue);
CFRelease( cfValue );
```

Finding FireWire Devices

After setting up a matching dictionary that describes the device or device type you're looking for, you pass it to the I/O Kit function `IOServiceGetMatchingServices`. This function searches the I/O Registry for objects that match the properties in the given dictionary and returns an iterator you can use to access each one:

```
kern_return_t    result;
io_iterator_t    iterator;

result = IOServiceGetMatchingServices( masterPort, matchingDictionary,
    &iterator );
```

Next, pass the iterator to the I/O Kit function `IOIteratorNext`, which returns a reference to the first matching object. Each call to `IOIteratorNext` returns the next available object:

```
io_object_t aDevice;

while ( (aDevice = IOIteratorNext( iterator ) ) != 0 ) {
    //get a device interface for the device
}
```

You can also use I/O Kit functions to receive device hot-plug and unplug notifications. To do this, you first set up a notification port and add its run loop event source to your program's run loop. Then, instead of calling `IOServiceGetMatchingServices`, you send the matching dictionary you created earlier to the I/O Kit function `IOServiceAddMatchingNotification` to get an iterator for matching device objects and install a notification request for new device objects that match. The function `myServiceMatchingCallback` is a function you supply to iterate over the matching devices and access each one. shows how to set up a notification.

Listing 2-2   Setting up a device hot-plug notification

```
//global variables
IONotificationPortRef    gNotifyPort;
CFRunLoopSourceRef       gNotifySource;

//local variables
io_iterator_t    iterator;
kern_return_t    result;
mach_port_t      masterPort;
CFMutableDictionaryRef  matchingDictionary;

//The acquisition of the Mach port and the creation of the matching
//dictionary are not shown here.
gNotifyPort = IONotificationPortCreate( masterPort );
gNotifySource = IONotificationPortGetRunLoopSource( gNotifyPort );
CFRunLoopAddSource( CFRunLoopGetCurrent(), gNotifySource,
    kCFRunLoopDefaultMode );

result = IOServiceAddMatchingNotification( gNotifyPort,
    kIOMatchedNotification, matchingDictionary,
    myServiceMatchingCallback, NULL, &iterator );

//IOServiceAddMatchingNotification consumes a reference to the
//matching dictionary so if you need to use the dictionary again
```

```
//after this call, you must first call CFRetain on it.

//Execute your callback function to iterate over the set of matching
//devices already present and to arm the notification for future devices.
myServiceMatchingCallback( NULL, iterator );
```

Getting FireWire Device Interfaces

After you've gotten an iterator and used it to get a reference to each matching device, you need to get a device interface for it. Regardless of which FireWire device interface library you want to use, you first get a plug-in interface of type IOCFPlugInInterface. You use the device's `io_object_t` reference you got from `IOIteratorNext` and the constant `kIOFireWireLibTypeID`, as in this example:

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

result = IOCreatePlugInInterfaceForService( aDevice, kIOFireWireLibTypeID,
    kIOCFPlugInInterfaceID, &cfPlugInInterface, &theScore );
```

Then, you use the IOCFPlugInInterface `QueryInterface` function to get the specific device interface you need. You pass it the identifier of the primary device interface you want to use, for example, `kIOFireWireDeviceInterfaceID` or `kIOFireWireSBP2LibLUNInterfaceID`:

```
IOFireWireLibDeviceRef  fwDeviceInterface = 0;

(*cfPlugInInterface)->QueryInterface( cfPlugInInterface, CFUUIDGetUUIDBytes(
    kIOFireWireDeviceInterfaceID ), (void **) fwDeviceInterface );
```

There are three versions of the IOFireWireDeviceInterface you can use. Each version adds new services to the functionality of the previous version. Be sure to check the header files to make sure you're running the version of Mac OS X that corresponds to the IOFireWireDeviceInterface version you want. If you try to get a later version of IOFireWireDeviceInterface than your version of Mac OS X supports, the `QueryInterface` function will fail. You can, however, get an earlier version of the interface than is currently supported by the version of Mac OS X you're running.

You can also get instances of IOFireWireUnitInterface and IOFireWireNubInterface. These are synonymous with IOFireWireDeviceInterface and refer to the same object. Their only function is to provide you with a more descriptive name for IOFireWireDeviceInterface if you open it on a unit or the local node.

When you have the primary device interface in the library you want to use, you do not have to use the `IOCreatePlugInForService` and `QueryInterface` functions to get the additional interfaces that library provides. Each library provides its own functions to get its other interfaces.

Getting Multiple FireWire Device Interfaces

When you open a device or unit using one of the device interface `open` functions, you have an exclusive connection to the object representing that device or unit in the kernel. Because the `open` function opens not only the object you call it on but also the objects that precede it in the stack, no other application can

then open that device or unit. For example, if you open an AV/C unit with the IOFireWireAVCLibUnitInterface's `open` function, you are automatically opening the IOFireWireUnit and IOFireWireDevice objects as well as the IOFireWireAVCUnit object.

Thus, if you want to use either the IOFireWireSBP2Lib or the IOFireWireAVCLib in cooperation with the IOFireWireLib, you must, in effect, open a device or unit object that is already open. In order to allow this, the IOFireWire family uses the concept of a session reference to refer to a particular connection to a device or unit object.

A session reference is like a key that allows you to override the exclusivity of the `open` function. If your application used a device interface `open` function to open an SBP-2 or AV/C unit, you can use the `GetSessionRef` function (available in both IOFireWireAVCLib and IOFireWireSBP2Lib) to get an `IOFireWireSessionRef` to open the IOFireWireUnit or IOFireWireDevice objects. The `GetSessionRef` function requires a pointer to the current device interface you hold so only your application can successfully use it.

You pass the `IOFireWireSessionRef` to the IOFireWireDeviceInterface function `openWithSessionRef`, opening the in-kernel device or unit object. You can then use the interfaces and functions of both libraries to communicate with your device or unit. Listing 2-3 (page 22) shows how to use an IOFireWireAVCLibUnitInterface instance to open an AV/C unit and then get the session reference to open the device for an instance of IOFireWireDeviceInterface.

Listing 2-3   Getting and using the session reference

```
IOFireWireSessionRef    session;

result = (*avcInterface)->open( avcInterface );
session = (*avcInterface)->getSessionRef( avcInterface );
result = (*fwInterface)->openWithSessionRef( fwInterface, session );
```

Before you can use these device interfaces, however, you must first find the device you're interested in. Device matching is the process of using I/O Kit functions to search the I/O Registry for particular devices (or units) or device types. This section provides general information on this process as it applies to FireWire devices. It supplies code fragments for common tasks such as searching the I/O Registry, setting up notifications for new devices, and getting the primary device interfaces in each FireWire device interface library. In addition, in "Getting Multiple FireWire Device Interfaces" (page 21), it describes how to use the IOFireWireDeviceInterface in conjunction with either the IOFireWireSBP2LibLUNInterface or the IOFireWireAVCLibUnitInterface.

Using the FireWire Device Interface Libraries

This chapter uses sample code from the FireWire SDK to illustrate how to use the FireWire device interface libraries. (The latest version of the SDK is available for download at http://developer.apple.com/hard-waredrivers/download.) The first section, "Using the IOFireWireLib" (page 23), describes two sample projects. The first sets up isochronous communication with a device and the second sets up a pseudo-address space in the Macintosh to handle FireWire packets. The second section, "Using the IOFireWireSBP2Lib" (page 28), presents a sample project that creates a user-space driver to access an SBP-2 device. Finally, "Using the IOFireWireAVCLib" (page 35) describes a project that accesses an AV/C unit with the IOFireWireAVCLib interfaces.

The code samples in this chapter omit most of the steps required to create matching dictionaries, search the I/O Registry, and get device interfaces. For code samples that show how to perform these tasks, see "Accessing FireWire Devices From Applications" (page 17) or the code projects in the FireWire SDK.

Using the IOFireWireLib

The IOFireWireLib provides the lowest level interfaces available to communicate directly with a FireWire device from an application. It also provides interfaces to set up isochronous communication with a FireWire device.

Setting Up Isochronous Communication

Isochronous communication takes place on channels that can each have at most one talker and any number of listeners. The IOFireWire family abstracts these concepts into channel objects and port objects. Channel objects correspond to FireWire bus channels and port objects are either remote or local and correspond to either talkers or listeners. Remote ports correspond to external devices and have methods you can override to support your device. The local port corresponds to the Macintosh and does not have methods you can override. You use these objects in your application to map out the flow of isochronous communication before any actual data transfer begins.

To manage the isochronous data your device sends and receives, you create a datastream control language (or DCL) program. The DCL program is a linked list of DCL commands that specify where to place each received packet and where to find the data to send. A DCL program can run both linearly, from beginning to end, and nonlinearly, by jumping from one command to another. You can even set up a DCL program to modify its jumps during execution, creating a dynamically changeable program. The IOFireWire family makes writing DCL programs easier by providing a function that allows you to create a pool of DCL command objects ready for use and functions that fill in DCL command objects with your parameters.

The IOFireWireLibIsochTest project creates a remote port and a local port, allocates an isochronous channel, sets up the remote port as a listener and the local port as the talker, and then starts the channel. The program includes a simple DCL program that manages the packets. IOFireWireLibIsochTest matches on the local node (the Macintosh itself) so you can run it even if no external FireWire devices are currently plugged in.

To begin with, the IOFireWireLibIsochTest project acquires a Mach port, creates a matching dictionary for the local node, and gets an IOFireWireDeviceInterface object for it. These steps closely follow those described in "Finding FireWire Devices" (page 20) so this section does not repeat them.

The code fragments and listings in this section use the variables in Listing 3-1 (page 24).

Listing 3-1 Variable definitions for IOFireWireLibIsochTest

```
//global declarations
IOFireWireLibDCLCommandPoolRef  gCommandPool;
UInt8                           gBuf[1024];
UInt8                           gBuf2[1024];

//local declarations in the main function
IOReturn                        result;
IOFireWireLibNubRef             localNode;
IOFireWireLibLocalIsochPortRef  localIsochPort;
IOFireWireLibRemoteIsochPortRef remoteIsochPort;
IOFireWireLibIsochChannelRef    isochChannel;
```

To exclusively open the device (in this case, the Macintosh), IOFireWireLibIsochTest `main` function calls the IOFireWireLibDeviceInterface function `open`:

```
result = (*localNode)->Open( localNode );
```

Next, it creates a pool of DCL command structures. The function `CreateDCLCommandPool` calls the IOFireWireLibDeviceInterface function of the same name, creating a DCL command pool object and returning an interface to it that the program can use to build a DCL program. Listing 3-2 (page 24) shows the function call followed by the function definition, excluding error checking.

Listing 3-2 Creating a DCL command pool

```
//Call to CreateDCLCommandPool from main.
result = CreateDCLCommandPool( localNode, &gCommandPool );
//...
//CreateDCLCommandPool function.
IOReturn CreateDCLCommandPool( IOFireWireLibNubRef inNub,
    IOFireWireLibDCLCommandPoolRef* outCommandPool ) {

    //Create a DCL command pool.
    *outCommandPool = (*inNub)->CreateDCLCommandPool( inNub, 0x1000,
        CFUUIDGetUUIDBytes( kIOFireWireDCLCommandPoolInterfaceID ) );
}
```

Before it can start the isochronous channel, the IOFireWireLibIsochTest `main` function first creates the remote and local port objects and an isochronous channel object. The IOFireWireLib contains interfaces for each of these objects that provide functions to manage them.

Listing 3-3 (page 24) shows the `CreateRemoteIsochPort` function (along with its call from `main`) which includes a number of callback functions you can override to provide device-specific functions, such as how to start and stop the port and which channels your device supports.

Listing 3-3 Creating a remote isochronous port

```
//Call to CreateRemoteIsochPort from main.
result = CreateRemoteIsochPort( localNode, &remoteIsochPort );
//...
```

```
//CreateRemoteIsochPort function.
IOReturn CreateRemoteIsochPort( IOFireWireLibNubRef inNub,
    IOFireWireLibRemoteIsochPortRef* outPort ) {
    IOFireWireLibRemoteIsochPortRef port;

    //Call the IOFireWireDeviceInterface function CreateRemoteIsochPort to
    //create a remote port object and return an interface to it. The "false"
    //parameter indicates that this port will not be a talker.

    port = (*inNub)->CreateRemoteIsochPort( inNub, false, CFUUIDGetUUIDBytes
                                (kIOFireWireRemoteIsochPortInterfaceID ) );
    (*port)->SetGetSupportedHandler( port, &RemotePort_GetSupported );
    (*port)->SetAllocatePortHandler( port, &RemotePort_AllocatePort );
    (*port)->SetReleasePortHandler( port, &RemotePort_ReleasePort );
    (*port)->SetStartHandler( port, &RemotePort_Start );
    (*port)->SetStopHandler( port, &RemotePort_Stop );
    *outPort = port;
}
```

In IOFireWireLibIsochTest, the `RemotePort_GetSupported` function indicates that it supports all channels and the remaining callback functions simply print messages. You should add your own code to these callback functions to support your device.

To create the local isochronous port, the IOFireWireLibIsochTest `main` function calls the function `CreateLocalIsochPort`. This function uses the IOFireWireDeviceInterface function of the same name to create a local isochronous port object with a particular DCL program and return an interface to it. Listing 3-4 (page 25) shows the `CreateLocalIsochPort` function, along with its call from `main`.

Listing 3-4   Creating a local isochronous port

```
//Call to CreateLocalIsochPort from main.
result = CreateLocalIsochPort( localNode, &localIsochPort );
//...
//CreateLocalIsochPort function.
IOReturn CreateLocalIsochPort( IOFireWireLibNubRef inNub,
    IOFireWireLibLocalIsochPortRef* outPort ) {

    //Set the pointer dclProgram to a simple DCL program created in
    //the IOFireWireLibIsochTest function WriteTalkingDCLProgram (not shown).
    DCLCommandStruct*   dclProgram = WriteTalkingDCLProgram();

    //Use the IOFireWireDeviceInterface function PrintDCLProgram
    //to print the contents of the DCL program.
    (*inNub)->PrintDCLProgram( inNub, dclProgram, 6 );

    //Create a local isochronous port object and return an interface
    //for it. The "true" parameter indicates that this port will be a talker.
    *outPort = (*inNub)->CreateLocalIsochPort( inNub, true, dclProgram, 0, 0,
        0, nil, 0, nil, 0, CFUUIDGetUUIDBytes(
        kIOFireWireLocalIsochPortInterfaceID ) );
    return *outPort;
}
```

Now the IOFireWireLibIsochTest `main` function uses channel interface functions to set up isochronous communication. First, it designates the remote port as a listener and the local port as the talker:

```
result = (*isochChannel)->AddListener( isochChannel,
```

```
    (IOFireWireLibIsochPortRef) remoteIsochPort );
//...
result = (*isochChannel)->SetTalker( isochChannel,
    (IOFireWireLibIsochPortRef) localIsochPort );
```

Then, it uses another channel interface function to allocate the channel:

```
result = (*isochChannel)->AllocateChannel( isochChannel );
```

The channel interface function `AllocateChannel` calls the `GetSupported` methods on all the ports to find out which channels they support, reconciles the answers, and requests a particular channel and bandwidth from the FireWire bus.

After the channel is allocated, the IOFireWireLibIsochTest `main` function calls the channel interface `Start` function which calls the `Start` functions on all the ports:

```
result = (*isochChannel)->Start( isochChannel );
```

To make sure the callbacks get called, the `main` function then calls an IOFireWireDeviceInterface function to add the isochronous callback dispatcher to the program's run loop:

```
(*localNode)->AddIsochCallbackDispatcherToRunLoop( localNode,
    CFRunLoopGetCurrent() );
```

For the purposes of testing, the `main` function calls the Core Foundation function `CFRunLoopRunInMode` to run the run loop and trigger the callback functions for 15 seconds. After that time, the `main` function then calls the channel interface `Stop` function which calls the `_Stop` functions on all the ports:

```
result = (*isochChannel)->Stop( isochChannel );
```

Finally, the IOFireWireLibIsochTest `main` function releases the interfaces it acquired. It first releases the isochronous channel object with the channel interface function `ReleaseChannel` and then it uses the IOCFPlugInInterface function `Release` to release the interface itself:

```
result = (*isochChannel)->ReleaseChannel( isochChannel );
//...
(*isochChannel)->Release( isochChannel );
```

The `main` function similarly releases the local and remote port interfaces, then calls the IOFireWireDeviceInterface function `Close` on the local node before releasing its interface, and releases the original CFPlugInInterface:

```
IODestroyPlugInInterface( gCFPlugInInterface );
```

Setting Up a Packet-Handling Project

The IOFireWire family defines two types of address space on the Macintosh: Physical address space and pseudo-address space. The IOFireWireDeviceInterface provides IOFireWirePhysicalAddressSpace objects, which are allocated in a range of hardware-backed addresses and IOFireWirePseudoAddressSpace objects, which are allocated in a range of addresses outside the physical range and are accessible only through software.

As mentioned in "FireWire Overview" (page 9), FireWire defines 64-bit addresses of which the lower 48 bits are for device-specific use. The physical address space range is the lower 32 bits of the device-specific range mapped onto the 32 -bit Macintosh RAM. Thus, in terms of FireWire addresses, an

IOFireWirePhysicalAddressSpace object is an allocated subrange in the range $0000.00000000 to $0000.FFFFFFFF. An IOFireWirePseudoAddressSpace object is an allocated subrange of FireWire addresses of $1.00000000 and higher.

The FireWire controller chip can write directly to and read directly from a physical address space without any software intervention. For this reason, you can transfer large amounts of data to and from this address space efficiently. Because there is no software involvement in such a transfer, however, there is no indication of when the transfer is complete or if it failed. To handle this, you must also allocate a pseudo-address space to serve as a messaging area. Your remote device then transfers its data to a physical address space and sends a message of success or failure to a pseudo-address space that alerts your software.

The IOFireWirePacketQueueTest project uses functions of the IOFireWireDeviceInterface to create an IOFireWirePseudoAddressSpace interface object that represents a pseudo-address space on the local node. The project then uses functions of the IOFireWirePseudoAddressSpace interface to get the FireWire address of the address space and set up callback routines for writing and reading to the address space. It also sets up a callback routine for handling packets that must be dropped when the queue of packets coming into the pseudo-address space fills up too quickly.

> **Note:** The IOFireWirePacketQueueTest project only *receives* asynchronous data; it does not send any data. To use IOFireWirePacketQueueTest, you should also run a program that sends data, such as FWUtil (also available in the FireWire SDK).

The `main` function of the `IOFireWirePacketQueueTest.cpp` first calls the `GetIOFireWireDevices` function to put all IOFireWireLocalNode device objects into an array. Then, it uses standard Core Foundation and I/O Kit functions to get an IOFireWireDeviceInterface for the first object in the array (see "Getting FireWire Device Interfaces" (page 21) for sample code illustrating this process). After using IOFireWireDeviceInterface functions to open the device and set up a callback dispatcher, the `main` function then creates a pseudo-address space object and gets an interface to it, as Listing 3-5 (page 27) shows.

Listing 3-5   Getting an IOFireWirePseudoAddressSpaceInterface

```
Ptr                     gBuf = 0; //Buffer for backing store.
//...
IOFireWireLibDeviceRef  interface;
//...
IOFireWireLibPseudoAddressSpaceRef  addressSpace = 0;
IOFireWireLibPhysicalAddressSpaceRefphysAddressSpace = 0;

//Allocate backing store and put some text into gBuf for testing.
gBuf = (Ptr) new char[40960];
sprintf( gBuf, "Testing..." );

//Use IOFireWireDeviceInterface function CreatePseudoAddressSpace to
//create a pseudo-address space object and get an interface to it.
//In the call to CreatePseudoAddressSpace, 40960 is the size of the
//address space, addressSpace is the reference value passed to all
//callback functions, and 4096 is the size of the queue that
//receives packets from the bus.

addressSpace = (*interface)->CreatePseudoAddressSpace( interface,
    40960, (void*) addressSpace, 4096, gBuf, kFWAddressSpaceAutoCopyOnWrite,
    CFUUIDGetUUIDBytes( kIOFireWirePseudoAddressSpaceInterfaceID ) );
```

Next, the `main` function uses IOFireWirePseudoAddressSpaceInterface functions to set up callback handler routines for reading and writing to the pseudo-address space and handling skipped packets. When a `write` to the pseudo-address space occurs, the `PacketWriteHandler` routine prints out information about the pseudo-address space and the packets it received. Similarly, when a `read` occurs, the `PacketReadHandler` routine handles the `read` and then prints out information about the `read` request. The `SkippedPacketHandler` routine simply prints out the number of skipped packets. All three handler routines end with a call to the IOFireWirePseudoAddressSpaceInterface function `ClientCommandIsComplete`, which notifies the pseudo-address space that a packet notification handler has completed its work, as in this example from `SkippedPacketHandler`:

```
//In the following function call, commandID is the same ID that was
//passed to the packet notification handler and kIOReturnSuccess is
//the completion status of the packet handler.

(*addressSpace)->ClientCommandIsComplete( addressSpace,
    commandID, kIOReturnSuccess );
```

The `main` function then sets everything in motion by calling `CFRunLoopRun`. Finally, it releases all the interfaces it acquired and ends.

Using the IOFireWireSBP2Lib

The SBP2SampleProject in the FireWire SDK adds another layer of complexity to the standard application-level access of a FireWire device. Instead of acquiring interfaces from the IOFireWireSBP2Lib and using them to communicate with a device, the SBP2SampleProject provides a user-space driver plug-in that exists between the application and the IOFireWireSBP2Lib. The application in the SBP2SampleProject creates a matching dictionary and gets an interface, but it is an interface to the user-space driver, not to the device object itself. The user-space driver then gets an interface from the IOFireWireSBP2Lib and uses it to access the in-kernel SBP-2 services. This design allows multiple applications to use the same user-space driver plug-in to access a device.

The SBP2SampleProject is a multi-threaded project. For an example of the same project in a single-threaded environment, see the SBP2SampleProject-SingleThread project.

The SBP2SampleProject driver abstracts device access into two logical layers. The lower layer is a transport layer that handles such tasks as SBP-2 ORB chaining and login maintenance. The upper layer is a protocol layer that receives commands from the application and creates and submits the appropriate ORBs to the transport layer to execute them. This architecture allows you to modify the upper, protocol layer to meet your device's needs without having to change much of the transport layer of the driver.

This section introduces the main features of the SBP2SampleProject and points out where you can change the code to use it as a foundation for your own project. In order to be universally applicable, the SBP2SampleProject accesses the computer's hard disk. To test the unmodified project on your Macintosh, follow the instructions included in the FireWire SDK to disable the built-in hard disk driver and allow the sample driver to attach.

Although disabling a built-in driver to allow your user-space driver to attach is an acceptable testing procedure, you should provide your own kernel extension (or KEXT) that can successfully compete with other drivers. This not only ensures that the I/O Kit finds and loads your KEXT for your device, but also provides the CFPlugIn information that identifies your user-space driver plug-in. In general, you should provide your own KEXT when your driver is a user-space plug-in that your application loads using the function

`IOCreatePlugInInterfaceForService` or when there may be other drivers competing for your device. Examine the bundle settings of the `SBP2SampleDriver.kext` to see how the driver plug-in is identified. For more information on developing a KEXT, see *I/O Kit Device Driver Design Guidelines*.

Setting Up the Application

The application portion of the SBP2SampleProject is written in Objective-C using the Cocoa framework and contains two code files: `SelectorController.m` and `LUNController.m`. In addition to GUI-related tasks, the implementation of the SelectorController class is responsible for:

- Getting the Mach port

- Creating a matching dictionary for IOFireWireSBP2LUN objects

- Getting an iterator over the set of matching objects

- Instantiating a LUNController object to create a name string for each matching IOFireWireSBP2LUN object

- Populating an array with matching IOFireWireSBP2LUN objects

- Using the LUNController object to get an interface to the driver plug-in for each IOFireWireSBP2LUN object the user selects

- Releasing the Mach port

The main task of the implementation of the LUNController class is to get the driver plug-in interface and use it to communicate with the device. An instance of the LUNController class:

- Uses the device object reference from an instance of SelectorController to get device properties from the I/O Registry

- Gets the driver plug-in interface for the device object and sets up callbacks on the current run loop

- Performs device `login` and `logout`

- Performs a `read` of the first four blocks of the device

- Releases the driver plug-in interface

The implementation of the LUNController class uses the same Core Foundation functions to get the interface to the driver plug-in as described in "Getting FireWire Device Interfaces" (page 21). For example,

```
//Get CFPlugIn interface.
status = IOCreatePlugInInterfaceForService( fLUNReference,
    kSBP2SampleDriverTypeID, kIOCFPlugInInterfaceID, &fCFPlugInInterface,
    &score );
//...
//Get driver plug-in interface.
result = (*fCFPlugInInterface)->QueryInterface( fCFPlugInInterface,
    CFUUIDGetUUIDBytes( kSBP2SampleDriverInterfaceID ), (LPVOID *)
    &fDriverInterface );
```

The only difference is in the parameters identifying the type of interface. Instead of using `kIOFireWireSBP2LibTypeID`, it uses `kSBP2SampleDriverTypeID` to get the IOCFPlugInInterface. Then, it calls `QueryInterface` with `kSBP2SampleDriverInterfaceID` to get the interface to the driver plug-in. Both `kSBP2SampleDriverTypeID` and `kSBP2SampleDriverInterfaceID` are defined in `SBP2SampleDriverInterface.h`.

An instance of the LUNController class performs device functions (`login`, `logout`, and `read`) using functions of the driver plug-in interface. The `login` and `logout` functions are straightforward calls to the driver plug-in interface:

```
(*fDriverInterface)->loginToDevice( fDriverInterface );
//...
(*fDriverInterface)->logoutOfDevice( fDriverInterface );
```

The LUNController instance uses a separate thread to perform the `read` function:

```
[NSThread detachNewThreadSelector:@selector(runReadTransaction:)
    toTarget:self withObject:nil];
```

Then, in the `runReadWorker` method, the LUNController instance uses the driver plug-in interface to send the `read` command:

```
(*fDriverInterface)->readBlock( fDriverInterface, transactionID, 1, &block );
```

Setting Up the Protocol Layer

The SBP2SampleProject classes SBP2SampleDriver and SBP2SampleDriverPlugInGlue define objects that comprise the protocol layer. The protocol layer first creates the transport layer, then creates the ORBs, fills and queues them, and communicates with the transport layer to submit them. In turn, the transport layer relays ORB-status and login-status messages to the protocol layer so the SBP2SampleDriver object can react accordingly.

The "glue" in SBP2SampleDriverPlugInGlue mainly refers to code that supplies the SBP2SampleDriver's methods as CFPlugIn functions (other code handles C++ idiosyncrasies, such as supplying static methods that can call virtual methods on objects). Recall that the application uses standard CFPlugIn functions, such as `QueryInterface`, to get and use the driver plug-in interface. If your application is getting IOFireWireLib or IOFireWireSBP2Lib interfaces directly, the libraries provide this "glue" behind the scenes. The SBP2SampleProject, however, must explicitly provide this code so the application can use the driver plug-in interface as if it were an IOFireWire family device interface.

The SBP2SampleDriver class implements many of the same methods an in-kernel driver does, such as `start`, `stop`, and `probe`. In its `start` method, the driver calls a method of the SBP2SampleORBTransport class to create the transport layer and uses the returned reference to tell the layer to start.

Next, it uses the transport layer's `createORB` method to create a `TEST_UNIT_READY` ORB. The `TEST_UNIT_READY` ORB is part of the configuration process for SBP-2 hard drives (see "Handling Device Configuration and Reconfiguration" (page 34) for more information on the device configuration process). Your device may have different configuration needs. Finally, it uses the `createORB` method again to create a pool of free ORBs to draw from later and the Core Foundation function `CFArrayCreateMutable` to create an array to hold in-progress ORBs.

The `stop` method of the SBP2SampleDriver class releases the ORB pool and in-progress ORB array and calls the `stop` method of the SBP2SampleORBTransport class to initiate the shutdown of the transport layer. The `probe` method merely checks to be sure that the referenced device object is of type IOFireWireSBP2LUN.

The SBP2SampleDriver class implementation uses Core Foundation array-handling functions to create and manage its queue of in-progress ORBs, as in this example:

```
CFMutableArrayRef    fInProgressORBQueue;
//...
//kSBP2SampleFreeORBCount is defined in SBP2SampleDriver.h
fInProgressORBQueue = CFArrayCreateMutable( kCFAllocatorDefault,
    kSBP2SampleFreeORBCount, SBP2SampleORB::getCFArrayCallbacks() );
```

Because the SBP2SampleDriver project is multithreaded, it's possible that one thread might remove an ORB from the free ORB pool at the same time another thread appends one. To protect against this, the SBP2SampleDriver methods that handle the free ORB pool get a lock on a mutex before removing or appending ORBs and release the lock afterwards, as Listing 3-6 (page 31) shows.

Listing 3-6   The getORBFromFreePool method of the SBP2SampleDriver class

```
SBP2SampleORB * SBP2SampleDriver::getORBFromFreePool( void )
{
    CFIndex freeORBCount;

    pthread_mutex_lock( &fFreePoolLock );
    while ( ( freeORBCount = CFArrayGetCount( fFreeORBPool ) ) == 0 ) {
        //If there are no free ORBs, wait for one.
        pthread_cond_wait( &fFreePoolCondition, &fFreePoolLock );
    }
    //Remove the ORB from the pool.
    SBP2SampleORB * orb = (SBP2SampleORB*) CFArrayGetValueAtIndex(
        fFreeORBPool, 0 );
    orb->retain();
    CFArrayRemoveValueAtIndex( fFreeORBPool, 0 );

    pthread_mutex_unlock( &fFreePoolLock );
    return orb;
}
```

If you need to customize the sample driver's queueing mechanisms, examine the following methods in which the SBP2SampleDriver class does most of its ORB-handling work:

- `getORBFromFreePool` gets a lock on the `fFreePoolLock` mutex and waits for a free ORB which it removes from the pool and returns.

- `addORBToFreePool` gets a lock on the `fFreePoolLock` mutex, appends the passed-in ORB to the free pool, and sends a signal that wakes up threads that are waiting for free ORBs.

- `appendORBToInProgressQueue` uses a Core Foundation function to add the passed-in ORB to the queue of in-progress ORBs.

- `removeORBFromInProgressQueue` uses Core Foundation functions to find the index of the passed-in ORB in the queue of in-progress ORBs and removes it.

The status of the ORBs is tracked by methods that send reports between the protocol and transport layers. The sample driver uses the transport layer's `submitORB` method to send an ORB to the device. The transport layer responds by calling one of the following SBP2SampleDriver class methods:

- `completeORB` if the device received the ORB

- `suspendORBs` when a bus reset occurs or if the device is unplugged

- `resumeORBs` if the device is again logged in
- `loginLost` if the connection to the device is lost

Setting Up the Transport Layer

The transport layer of the SBP2SampleProject is responsible for acquiring the appropriate IOFireWireSBP2Lib interfaces that allow direct communication with the device and using those interfaces to send ORBs and manage the login status of the device.

The transport layer comprises four classes, listed in order of proximity to the device object:

- SBP2SampleSBP2LibGlue
- SBP2SampleLoginController
- SBP2SampleORB
- SBP2SampleORBTransport

When the protocol layer starts the transport layer, the `start` method of the SBP2SampleSBP2LibGlue class executes, acquiring an IOFireWireSBP2LibLUNInterface. Using this interface, the SBP2SampleSBP2LibGlue instance opens the LUN and gets the IOFireWireSBP2LibLoginInterface. The IOFireWireSBP2LibLoginInterface supplies APIs for login maintenance and command execution.

> **Note:** The SBP2SampleSBP2LibGlue class implementation also shows how to get the IOFireWireLibInterface using the session reference the IOFireWireSBP2LibLUNInterface supplies, but this is conditionally bypassed in the code.

The SBP2SampleSBP2LibGlue class uses the IOFireWireSBP2LibLoginInterface to register status callbacks that the in-kernel SBP-2 services use to notify the driver, such as `setLoginCallback` and `setStatusNotify`. In addition, the class uses this interface to create an IOFireWireSBP2LibMgmtORBInterface object that gives you access to an SBP-2 management ORB. This object allows you to execute commands such as `QueryLogin`, `AbortTask`, and `LogicalUnitReset`. In this project, the SBP2SampleSBP2LibGlue class uses the IOFireWireSBP2LibMgmtORBInterface object functions to set the command to be managed by the management ORB (with `setManageeLogin`), set the function of the management ORB (with `setCommandFunction`), and set the ORB completion routine (with `setORBCompleteCallback`).

The SBP2SampleLoginController class uses the IOFireWireSBP2LibLoginInterface to send `login` and `logout` commands to the device. The `loginCompletion` method checks status parameters and tries to log in again if necessary. You can add code here to check for status values specific to your device. The SBP2SampleLoginController class also handles callback messages from the device describing the status of the device's connection. You can subclass the methods that handle the login states (lost, suspended, and resumed) to perform device-specific tasks.

The SBP2SampleORB class implementation is responsible for the ORB object itself. Methods in this class initialize a new SBP2SampleORB object that represents the ORB and provide access to various fields in the IOFireWireSBP2LibORBInterface objects, such as the transaction identification and the maximum ORB payload size.

The SBP2SampleORBTransport class abstracts the transport of ORBs from the protocol layer to the lowest levels of the transport layer. When the SBP2SampleDriver class initiates the transport layer, it calls the factory method of the SBP2SampleORBTransport class. In its `start` method, the SBP2SampleORBTransport class instantiates the SBP2SampleSBP2LibGlue class which acquires the IOFireWireSBP2Lib interfaces it needs to communicate with the device. The SBP2SampleORBTransport class also implements the `submitORB` method for the protocol layer, using the IOFireWireSBP2LibLoginInterface function `submitORB`.

The SBP2SampleORBTransport class notifies the protocol layer of ORB status with a method called `statusNotify`. Listing 3-7 (page 33) shows the `statusNotify` method. It does not show the method `parseStatus` that interprets the ORB's status and returns one of the values enumerated at the beginning of the SBP2SampleORBTransport class implementation, or the messages `statusNotify` sends to `FWLog` (a macro defined in `FWDebugging.h`).

Listing 3-7   The statusNotify method of the SBP2SampleORBTransport class

```
void SBP2SampleORBTransport::statusNotify( FWSBP2NotifyParams * params )
{
    SBP2SampleORB * orb = ( SBP2SampleORB *)params->refCon;
    UInt32 event = parseStatus( params );

    switch( event )
    {
        case kStatusDeadBitSet:
            //Suspend protocol layer.
            fDriverLayer->suspendORBs();
            //Complete this ORB.
            completeORB( orb, kIOReturnError );
            //Reset the fetch agent.
            (*fSBP2LoginInterface)->submitFetchAgentReset(
                fSBP2LoginInterface );
            break;
        case kStatusDummyORBComplete:
            if ( orb == fDummyORB ) {
                // All is initialized so resume protocol layer.
                fDriverLayer->resumeORBs();
            }
            else {
                //This must be an aborted ORB.
                completeORB( orb, kIOReturnError );
            }
            break;
            case kStatusORBComplete:
                // Complete the protocol layer's ORBs.
                completeORB( orb, kIOReturnSuccess );
                break;
            case kStatusORBError:
                completeORB( orb, kIOReturnIOError );
                break;
            case kStatusORBReset:
                //This is a command reset so tell the protocol layer (it
                //should already be suspended at this point).
                completeORB( orb, kIOReturnNotReady );
                break;
            case kStatusORBTimeout:
                //Suspend the protocol layer.
                fDriverLayer->suspendORBs();
                //Complete this ORB.
```

```
            completeORB( orb, kIOReturnError );
            //Reset the LUN.
            (*fLUNResetORBInterface)->submitORB( fLUNResetORBInterface
                );
            break;
        }
}
```

In Listing 3-7 (page 33), the `completeORB` method refers to a method of the SBP2SampleORBTransport class only if the ORB in question is the dummy ORB (described in "Handling Device Configuration and Reconfiguration" (page 34)); otherwise, it refers to a method of the SBP2SampleDriver class.

Handling Device Configuration and Reconfiguration

Both the transport and protocol layers must perform some configuration before they can accept commands from the application. This occurs when the application starts up and after bus resets. The transport layer appends a dummy ORB (an ORB with no command) to a special register on the SBP-2 device called the fetch agent which holds the address of the ORB the device is currently working on. At login or after a bus reset, you must write the address of the first ORB directly to the fetch agent register. You can then chain all subsequent ORBs onto that first ORB. In this case, the dummy ORB's function is to always be the first ORB executed at login or after a bus reset.

Because the project is written to access a mass storage device, the protocol layer reconfigures the device when the application starts up and after every bus reset. The protocol layer sends a `TEST_UNIT_READY` ORB because the device is a hard disk—your device might require another type of ORB. The protocol layer must wait until the transport layer has executed the dummy ORB before it can send the `TEST_UNIT_READY` ORB. The application must wait until both these ORBs have executed before it can start sending its own ORBs.

When the project first starts up, an instance of the LUNController class calls the `loginToDevice` method of the SBP2SampleDriver class in response to user input. The protocol and transport layers then perform the following steps to initialize the device and ready it to accept ORBs from the application:

■ The SBP2SampleDriver instance calls the `loginToDevice` method of the SBP2SampleLoginController class.

■ The SBP2SampleLoginController instance uses the IOFireWireSBP2LibLoginInterface function `submitLogin` to log in to the device.

■ When the login is complete, it triggers the SBP2SampleLoginController callback method `loginCompletion` which calls the SBP2SampleORBTransport class's `loginResumed` method.

■ The SBP2SampleORBTransport instance submits a dummy ORB.

■ When the dummy ORB completes, the `statusNotify` method of the SBP2SampleORBTransport calls the SBP2SampleDriver's `resumeORBs` method which sets the `fSuspended` flag to `false` and submits a `TEST_UNIT_READY` ORB to the transport layer.

■ The SBP2SampleORBTransport instance submits the `TEST_UNIT_READY` ORB.

■ When the `TEST_UNIT_READY` ORB completes, the SBP2SampleORBTransport `statusNotify` method calls the `completeORB` method of the SBP2SampleDriver class.

■ The SBP2SampleDriver `completeORB` method checks if the ORB just completed is a `TEST_UNIT_READY` ORB and, if it is, it sets the flag `fDeviceReady` to `true` and proceeds to submit ORBs from the application.

After bus resets, the steps are similar:

- The SBP2SampleLoginController instance receives a `kIOMessageServiceIsSuspended` message and, if it is currently logged in to the device, it sets the `fLoggedIn` flag to `false`.

- The SBP2SampleDriver instance executes its `suspendORBs` method which sets the `fSuspended` flag to `true` and the `fDeviceReady` flag to `false`.

- When the device successfully reconnects, the SBP2SampleLoginController instance receives a `kIOMessageFWSBP2ReconnectComplete` message and sets the `fLoggedIn` flag to `true`.

- The SBP2SampleORBTransport instance then calls the SBP2SampleLoginController class's `loginResumed` method (which you can modify to perform device-specific operations) and submits a dummy ORB.

- When the dummy ORB completes, the `statusNotify` method of the SBP2SampleORBTransport calls the SBP2SampleDriver's `resumeORBs` method which sets the `fSuspended` flag to `false` and submits a `TEST_UNIT_READY` ORB to the transport layer.

- The SBP2SampleORBTransport instance submits the `TEST_UNIT_READY` ORB.

- When the `TEST_UNIT_READY` ORB completes, the SBP2SampleORBTransport `statusNotify` method calls the `completeORB` method of the SBP2SampleDriver class.

- The SBP2SampleDriver `completeORB` method checks if the ORB just completed is a `TEST_UNIT_READY` ORB and, if it is, it sets the flag `fDeviceReady` to `true` and proceeds to submit ORBs from the application.

Using the IOFireWireAVCLib

The IOFireWireAVCLib provides two interfaces that you can use independently of each other. The IOFireWireAVCLibProtocolInterface allows you to treat the Macintosh as an AV/C unit, setting up local plugs and receiving AV/C requests. The IOFireWireAVCLibUnitInterface supplies methods that send AV/C commands to an external AV/C unit. The IOFireWireAVCLibUnitInterface also provides a method to get the session reference so you can simultaneously get the IOFireWireLibDeviceInterface to access an external device and read its plug control registers.

The AVCBrowser project shows how to use both interfaces. First, it uses the IOFireWireAVCLibProtocolInterface to allocate and read input and output plugs on the Macintosh and receive AV/C requests. Then, it finds all AV/C units currently in the I/O Registry and gets an IOFireWireAVCLibUnitInterface for each. With this interface, the project opens the unit and sends it commands. Finally, the project gets an IOFireWireDeviceInterface for the device object underlying the IOFireWireAVCUnit object and reads its plug control registers.

AVCBrowser is written in Objective-C using the Cocoa framework and, as such, contains several methods that handle the user interface. For the sake of brevity, this document focuses on only those methods that acquire and use the IOFireWireAVCLib interfaces.

AVCBrowser contains four classes:

- DevicesController implements a DevicesController object that contains an array of AV/C units it finds in the I/O Registry and includes an instance method to open the device and display information about its input and output plugs.

- AVCDevice implements an AVCDevice object that contains information about an IOFireWireAVCUnit object, including a handle to its IOFireWireAVCLibUnitInterface and an IOFireWireDeviceInterface for its underlying IOFireWireDevice object.

- PlugBrowserController implements a PlugBrowserController object that responds to user requests for information about a particular AV/C unit's input and output plugs.

■ Plug implements a Plug object for each plug on an AV/C unit that includes information on its broadcast status and channel.

The implementation of the DevicesController `init` method uses standard I/O Kit functions to get a Mach port and to search the I/O Registry for IOFireWireLocalNode objects (if there is more than one FireWire bus, there is more than one IOFireWireLocalNode object representing the Macintosh). For each object it finds, the `init` method gets an IOFireWireAVCLibProtocolInterface and it uses the interface function `setMessageCallback` to receive bus reset and reconnect messages. It also sets up a routine to handle requests sent to the Macintosh from an AV/C device (in this sample, a video camera), using the interface function `setAVCRequestCallback`.

The DevicesController `init` method then sets up input and output plugs on the local node and reads their values using IOFireWireAVCLibProtocolInterface functions, as Listing 3-8 (page 36) shows.

Listing 3-8   Setting up plugs on the local node

```
IOFireWireAVCLibProtocolInterface ** avcInterface;
UInt32 inputPlug, outputPlug;
kern_return_t result;

//Not shown in this listing: acquisition of 'avcInterface' and error
//checking using the value of 'result'.
if ( (*avcInterface)->allocateInputPlug )
    //In the call to allocateInputPlug, writePlug is the callback function
    //called when a successful lock transition plug has been performed
    //and inputPlug is set to the plug number upon successful allocation.
    result = (*avcInterface)->allocateInputPlug( avcInterface, self,
        writePlug, &inputPlug )

if ( (*avcInterface)->readInputPlug ) {
    UInt32 val;
    val = (*avcInterface)->readInputPlug( avcInterface, inputPlug );
    if ( (*avcInterface)->updateInputPlug )
        (*avcInterface)->updateInputPlug( avcInterface, inputPlug, val,
            val+1 );
    val = (*avcInterface)->readInputPlug( avcInterface, inputPlug );
}

if ( (*avcInterface)->freeInputPlug )
    (*avcInterface)->freeInputPlug( avcInterface, inputPlug );

if ( (*avcInterface)->allocateOutputPlug )
    (*avcInterface)->allocateOutputPlug( avcInterface, self, writePlug,
        &outputPlug );

if ( (*avcInterface)->readOutputPlug )
    (*avcInterface)->readOutputPlug( avcInterface, outputPlug );
```

To set up a dispatcher for kernel messages to the program, the `init` method uses the IOFireWireAVCLibProtocolInterface function `addCallbackDispatcherToRunLoop`, as in this example:

```
(*avcInterface)->addCallbackDispatcherToRunLoop( avcInterface,
    CFRunLoopGetCurrent() );
```

After setting up the IOFireWireAVCLibProtocolInterface for the local node, the `init` method then registers for notification of new IOFireWireAVCUnit objects in the I/O Registry, using the functions described in "Finding FireWire Devices" (page 20). To find all IOFireWireAVCUnit objects already present in the I/O Registry and arm the notification, the DevicesController class calls its function `serviceMatchingCallback`. This function instantiates an AVCDevice object (declared in `AVCDevice.h`) for each IOFireWireAVCUnit object it finds.

The implementation of the AVCDevice class method `withIOService:` uses I/O Kit and Core Foundation functions to extract the IOFireWireAVCUnit object's `GUID` (globally unique identifier), `Unit_Type`, and `FireWire Product Name` properties. You can use these properties (or any of the properties of the IOFireWireAVCUnit object listed in Table 2-2 (page 18)) to create a more specific matching dictionary. The `withIOService:` method calls the private function `findPluginForDevice` to create a dictionary that matches on IOFireWireAVCUnit objects with the passed-in GUID and to create an interface of type IOCFPlugInInterface for it.

The `withIOService:` method queries the IOCFPlugInInterface from `findPluginForDevice` to get the IOFireWireAVCLibUnitInterface. It then uses IOFireWireAVCLibUnitInterface functions to open the unit, set up a callback routine and dispatcher, and send a command to the unit, as Listing 3-9 (page 37) shows.

Listing 3-9   Communicating with an external AV/C device

```
IOReturn    result;
IOFireWireAVCLibUnitInterface    **avcInterface;
AVCDevice   *theDevice;
UInt32      size;
UInt8       cmd[8], response[8];

cmd[kAVCCommandResponse] = kAVCStatusInquiryCommand;
cmd[kAVCAddress] = kAVCUnitAddress;
cmd[kAVCOpcode] = kAVCUnitInfoOpcode;
cmd[3] = cmd[4] = cmd[5] = cmd[6] = cmd[7] = 0xff;
size = 8;

//Error-checking by examining 'result' is not shown here.

result = (*avcInterface)->open( avcInterface );

//In the call to setMessageCallback, avcMessage is the completion routine
//that forwards AV/C bus status messages from the I/O Kit.
(*avcInterface)->setMessageCallback( avcInterface, theDevice, avcMessage );
(*avcInterface)->addCallbackDispatcherToRunLoop( avcInterface,
    CFRunLoopGetCurrent() );
result = (*avcInterface)->AVCCommand( avcInterface, cmd, 8, response, &size
    );
```

Next, the `withIOService:` method passes the IOFireWireAVCUnit object's GUID property and the IOFireWireDevice service type to the `findPluginForDevice` function to get an IOCFPlugInInterface for the corresponding device object. It then gets an IOFireWireDeviceInterface for the device object and uses interface functions to add a callback dispatcher to the current run loop and open the device, as in this example:

```
IOFireWireLibDeviceRef  resultInterface;
(*resultInterface)->AddCallbackDispatcherToRunLoop( resultInterface,
    CFRunLoopGetCurrent() );
(*resultInterface)->Open( resultInterface );
```

Finally, the `withIOService:` method uses the instance method `readQuad:` to call the IOFireWireDeviceInterface function `ReadQuadlet` with the addresses of the master output and input plugs, as Listing 3-10 (page 38) shows.

Listing 3-10   Reading the plug control registers of an external AV/C device

```
//...
AVCDevice *theDevice = [[self alloc] init];
//...
FWAddress addr;
UInt32 master;
addr.addressHi = 0xffff;
addr.addressLo = 0xf0000900;

//Read the master output plug.
master = [theDevice readQuad:addr];

//Place results in the AVCDevice object 'theDevice'.
[theDevice setOutMax: [NSNumber numberWithUnsignedInt:(1 << (master >> 30))
    * 100]];
[theDevice setOutBase: [NSNumber numberWithUnsignedInt:(master >> 24 &
    0x3f)]];
[theDevice setOutNum: [NSNumber numberWithUnsignedInt:master & 0x1f]];

addr.addressLo = 0xf0000980;

//Read the master input plug.
master = [theDevice readQuad:addr];

//Place results in the AVCDevice object 'theDevice'.
[theDevice setInMax: [NSNumber numberWithUnsignedInt:(1 << (master >> 30)) *
    100]];
[theDevice setInMax: [NSNumber numberWithUnsignedInt:(master >> 24 & 0x3f)]];
[theDevice setInMax: [NSNumber numberWithUnsignedInt:master & 0x1f]];

//...

//The readQuad: instance method.
-(UInt32)readQuad:(FWAddress)addr
{
    UInt32 val = 0xdeadbeef;
    IOReturn status;
    status = (*_interface)->ReadQuadlet( _interface, _device, &addr, &val,
        kFWDontFailOnReset, 0 );
    return val;
}
```

When the AVCBrowser project runs, the user can choose to open a particular AV/C device and view its type, packet speed, and number of input and output plugs. For each plug, the user can view information about it, such as if it is online and broadcasting.

FireWire Device Access in an Intel-Based Macintosh

This chapter provides an overview of some of the issues related to developing a universal binary version of an application that accesses a FireWire device. 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. The guidelines in that document apply to all types of applications, including those that access hardware.

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

■ You port your project to GCC 4 (Xcode uses GCC 4 to target Intel-based Macintosh computers)

■ You install the Mac OS X v10.4 universal SDK

■ You develop your project in Xcode 2.1 or later

Byte Ordering on the FireWire Bus

The FireWire bus is a big-endian bus. Structured data (such as a FireWire address or configuration ROM) appears on the bus in big-endian format, regardless of the native endian format of the computer an application is running in. Data that has no FireWire-specific structure, such as disk block (or sector) data, appears on the bus in its original format, regardless of the native endian format of the computer sending or receiving that data.

Although the IOFireWire family does not need to swap data buffers for transmission on the FireWire bus, your application must be careful with the arguments passed to and received from FireWire APIs and services. This is because those arguments may need to be byte swapped if they express payload data in numerical ways, such as with `UInt32` values. For example, if an application uses the `Write` function of the IOFireWireDeviceInterface, it must pass the following parameters (among others):

■ The target address for the command

■ A pointer to a buffer containing the data to write

■ The number of bytes to write

■ The FireWire bus generation during which the command should be executed

The target address is passed in a `FWAddress` structure. Because the IOFireWire family knows how to interpret a `FWAddress` structure as numeric values, the family expects those values to be in host-native format. This means that you don't have to do any byte-swapping when you populate the structure with, for instance, the target address. Continuing the `Write` function example, Listing 4-1 shows an example of how to create a target address that will produce correct results whether the application is running in a PowerPC-based or Intel-based Macintosh.

Listing 4-1   Universal way to create a FireWire target address

```
FWAddress a;
a.nodeID = 0xffc0;
a.addressHi = 0xffff;
a.addressLo = 0xf000040c;
```

Internally, the IOFireWire family may need to swap these values when it programs the hardware, but it will not modify the contents of the structure you prepare.

Similarly, the `Write` function arguments for byte count and bus generation should also be in host-native format, as should the pointer to the buffer. The IOFireWire family does not interpret the data in the buffer, so it is sent on the bus unmodified.

Because the data in the buffer is not modified by the IOFireWire family, your application must use endian-safe ways to express the data when filling a buffer for transmission. To see why this is so, imagine your application puts a FireWire address in a buffer to be sent as payload (as you would in an SBP-2 ORB). If your application uses code similar to that shown in Listing 4-2 to fill the buffer, the application would work only in a PowerPC-based Macintosh, not an Intel-based Macintosh:

Listing 4-2   PowerPC-only way to fill a buffer for transmission on the FireWire bus

```
// Fill a buffer with the FireWire address 0xffc2ee33.00f01234
// to send as payload (works only in a PowerPC-based Macintosh).
UInt32 myBuffer[2];
myBuffer[0] = 0xffc2ee33;
myBuffer[1] = 0x00f01234;
```

If you pass the version of `myBuffer` defined in Listing 4-2 to the `Write` function in an application running in an Intel-based Macintosh, the address will be received as `0x33eec2ff.3412f000`. This is because an Intel-based Macintosh stores numerical values such as `0xffc2ee33` in little-endian format and the IOFireWire family sends the buffer unmodified. To avoid this problem, use endian-neutral code that will produce correctly ordered bytes on both types of computer, as shown in Listing 4-3.

Listing 4-3   Universal way to fill a buffer for transmission on the FireWire bus

```
// Fill a buffer with the FireWire address 0xffc2ee33.00f01234
// to send as payload (works in both PowerPC-based and Intel-based
// Macintosh computers).
UInt32 myBuffer[2];
myBuffer[0] = OSSwapHostToBigInt32 (0xffc2ee33);
myBuffer[1] = OSSwapHostToBigInt32 (0x00f01234);
```

In general, wherever you use a multibyte constant (such as `0xffc2ee33`) to represent bytes in a buffer instead using of a numeric value, you will probably need to swap that constant the ensure the bytes appear in the correct order in memory and on the FireWire bus.

For a few specific types of data, the IOFireWire family does perform automatic byte-swapping to the host computer's native endian format. These exceptions are listed below.

- Configuration ROM data. A FireWire device's configuration ROM (or config ROM) contains information such as vendor ID, special register addresses, and unit directories. The information in the config ROM is assumed to be in the format defined by the IEEE 1394 specification and is organized as a hierarchy of key-value pairs. Most of these values represent key-specific data (called immediate values), but some values are pointers to blocks of data (called data leaves). When your application uses IOFireWire family

APIs to get or set immediate values, the values are swapped to the host format because the IOFireWire family can correctly interpret the value. The IOFireWire family cannot interpret data leaves, however, so these blocks of data are left in big-endian format.

Note that this differs from the byte order of a device's config ROM data as published in the I/O Registry in the value of the FireWire Device ROM key. The value of the FireWire Device ROM key is all the information the IOFireWire family has read from the device's config ROM. In this case, the data is not parsed into specific values, but is placed in the I/O Registry as it came from the FireWire bus—in big-endian format.

- AV/C plug control register values. The plug control registers (or PCRs) allow an application to create point-to-point or broadcast connections between AV/C devices or between the Macintosh and an external AV/C device. The format of the 32-bit PCR values are well defined and the built-in drivers automatically swap these values to host-native format for your convenience.

- Isochronous packet headers. Although all data is transmitted on the FireWire bus in big-endian format, the FireWire DMA processes isochronous packet headers in little-endian format. The IOFireWire family swaps these headers to host format. On a PowerPC-based Macintosh, this results in both header and packet being stored in big-endian format. On an Intel-based Macintosh, however, the header is stored in little-endian format whereas the packet is stored in big-endian format. Therefore, you can expect an isochronous header to be in host-native format, even though it represents data transmitted on the FireWire bus.

Code Modification Hints

This section lists some specific hints to help you prepare your application to run in an Intel-based Macintosh. When you've determined that conditional byte swapping is necessary, use the functions and macros defined in the `libkern/OSByteOrder.h` header file in the Kernel framework. For guidelines on various byte-swapping strategies, see "Swapping Bytes" in *Universal Binary Programming Guidelines, Second Edition*.

In general, if you developed your application on a PowerPC-based Macintosh, you may have taken advantage of the fact that the PowerPC processor and the FireWire bus both use the big-endian format. You should search for places where you assume that your data is automatically in the correct byte order and insert conditional byte-swapping code if necessary.

Formulating Structures

Pay particular attention to how you formulate structures, such as the `FWAddress` structure, when you plan to pass them as pointers to blocks of data in buffers. Depending on how you fill such structures, you may have to perform byte swapping. For example, the IOFireWire family defines the `FWAddress` structure as:

```
typedef struct FWAddressStruct
{
    UInt16  nodeID;     // bus/node
    UInt16  addressHi;  // Top 16 bits of node address.
    UInt32  addressLo;  // Bottom 32 bits of node address
} FWAddress, *FWAddressPtr;
```

You might choose to create this structure by formulating a single `UInt64` value or two `UInt32` values. If your application is running in an Intel-based Macintosh, you'll have to perform the appropriate byte swap on these values before you place them in a buffer for transmission on the FireWire bus.

Accessing Values in the I/O Registry

Because an Intel-based Macintosh does not use Open Firmware, some information provided by Open Firmware on a PowerPC-based Macintosh (such as the complete device tree) is not available in the I/O Registry of an Intel-based Macintosh. However, many parts of the I/O Registry are present, including the information provided by the IOFireWire family.

If your application accesses values in the I/O Registry, you should be aware that values of type number, such as the value of the GUID key, are in host-endian format. Values of type data, however, are in big-endian format. As described in "Byte Ordering on the FireWire Bus" (page 39), the IOFireWire family places the unparsed bytes of a device's config ROM into the value of the FireWire Device ROM key in big-endian format.

Byte Swapping Inside Buffers

Sometimes, the buffers you use in your application are created by some other entity, such as a member of the IOFireWire family. If you byte swap the values in such a buffer itself, you may find that these same values get swapped again at some later point. For this reason, it's best to create your own copy of the buffer so you can swap its values without affecting the original buffer.

Document Revision History

This table describes the changes to *FireWire Device Interface Guide*.

Date	Notes
2007-02-08	Made minor corrections.
2006-05-23	Made minor corrections.
2005-11-09	Added information about additional AV/C device interface functions.
2005-09-08	Added information about endian issues. Changed title from "Working With FireWire Device Interfaces".

Document Revision History