Contents

FireWire Tasks.pdf 3/8/2002

NOTE: This is a short document to help those just getting started with the FireWire SDK. It is not complete/official; it is just a crash-course style introduction to some concepts related to writing user-space FireWire drivers. The accompanying PDF files contain illustrations that may help further illustrate the contents of this document.

1 General - IOKitLib

1.1 Creating a matching dictionary to locate and match devices in the registry

Use the methods of IOKitLib (in IOKitLib.h) to find connected FireWire devices and units. You specify which devices you are interested in using a matching dictionary (type CFDictionrayRef). Read the CoreFoundation documentation for more information about CFDictionary.

When devices are connected to the FireWire bus, driver objects are created and published in the I/O Registry. In most cases you will be matching devices and units based on the name of these driver class instances in combination with additional properties, such as device vendor ID. You can use IOKit matching to match on any class name; here are some important FireWire-related class names:

- **IOFireWireDevice** - created for every device on the bus. Match on this object to control an entire device or a device with no units. The user interface to this object is provided through **IOFireWireDeviceInterface**, an interface of IOFireWireLib**.**
- **IOFireWireUnit** - created for every unit in a device. Match on this object to control a unit in a device. The user interface to this object is provided through **IOFireWireDeviceInterface**, an interface of IOFireWireLib**.**
- **IOFireWireSBP2Target** - An instance of IOFireWireSBP2Target is created for each unit in a device that is an SBP2 target.
- **IOFireWireSBP2LUN** - An instance of IOFireWireSBP2LUN is created for every LUN discovered in anSBP2 target. Match on this object to control SBP2 LUNs. The user interface to this object is provided by IOFireWireSBP2LibLunInterface which is an interface of IOFireWireSBP2Lib.
- **IOFireWireAVCUnit** - An instance of IOFireWireAVCUnit is created for every device that contains AVC units. In user space, AVC commands may be issued using an instance **IOFireWireAVCLibUnitInterface** and AVC requests and PCR notifications may be received using **IOFireWireAVCProtocolInterface**. These interfaces are part of IOFireWireAVCLib.
- **IOFireWireLocalNode** – An instance of this class corresponds to the Mac itself. When FireWire is loaded, an instance of this class will always be present. Match on this class to implement drivers that do not require another device to be present, for example, peer-to-peer protocol drivers.

A matching dictionary that specifes an class name to match can be created by using `IOServiceMatching()`. This returns a CFMutableDictionaryRef to which additional matching properties may be added using the CFDictionary API *(see CoreFoundation/CFDictionary.h).* For an example that matches on more than just driver class name, see the FWCCMVDigComponent example in the FireWire SDK which contains code to match a unit with a specific unit_spec_ID and unit_sw_version.

1.2 Using a matching dictionary to find connected devices and units

Once a matching dictionary has been created, it is be passed to `IOServiceGetMatchingServices()` which will return an iterator (of type io_iterator_t). For more information, refer to the IOKitLib documentation. Objects returned by the iterator correspond to in-kernel driver class instances, or services, that match the criteria specified in the matching dictionary and that existed when the iterator was created.

Use `IOIteratorNext()` to iterate over an iterator. Each time this function is called it will return a new object reference or NULL (which indicates iteration is complete). When you are finished with the iterator, dispose of it by calling `IOObjectRelease()`.

1.3 Using a matching dictionary to receive device plug and unplug notifications

A matching dictionary can also be used to receive device hot-plug and unplug notifications.

Once a matching dictionary has been created, it can be passed to `IOServiceAddMatchingNotification()`. This will setup matching notification and return a notification object (of type io_notification_t). Your callback function will be called with the specified refcon and an iterator (type io_iterator_t). Iterate over the iterator to receive a reference to the services for which you are being notified (*see "Using a matching dictionary to find connected devices and units", above*). Do not release the interator in your callback.

You must directly call your callback immediately after calling `IOServiceAddMatchingNotification()` to activate notifications. Do not release the iterator.

`IOServiceAddMatchingNotification()` requires a notification port reference (of type IONotificationPortRef) which can be created by calling `IONotificationPortCreate()`. The notification port reference should be disposed of using `IONotificationPortDestroy()`.

A notification port has a run loop source that must be added to a run loop (an object representing a thread of execution) for it to function. Get the run loop source by calling `IONotificationPortGetRunLoopSource()`. In most cases, adding the source to the run loop returned by `CFRunLoopGetCurrent()` will be sufficient. Call `CFRunLoopAddSource()` to add a run loop source to a run loop. See the CoreFoundation documentation for more information about CFRunLoop.

The notification object returned from `IOServiceAddMatchingNotification()` should be released with `IOObjectRelease()` when you no longer wish to receive notifications.

Please refer to the "Notifier" example in the FireWire SDK for a sample implementation.

1.4 Bus reset and other device notifications

You can receive bus reset (and other) notifications. Use `IOServiceAddInterestNotification`() with an interest type of `kIOGeneralInterest`. The service parameter is the reference to the service from which you wish to receive notifications. The service is the same as the one obtained during matching.

Here is a short list of messages that may be sent to your message routine and their meanings:

- `KIOMessageServiceIsTerminated` - The in-kernel service has been terminated. When this message is sent following a `kIOMessageServiceIsSuspended` message, the device has been unplugged.
- `KIOMessageServiceIsSuspended` - This message means a bus reset cycle has started. You may receive this message more than once per reset cycle.
- `kIOMessageServiceIsResumed` - This message is sent once a bus reset cycle has completed.
- `kIOMessageServiceIsRequestingClose` - The service will be terminated. Please close any open interfaces.
- `kIOFWMessageServiceIsRequestingClose` - Same meaning as `kIOMessageServiceIsRequestingClose`, above. This message is currently defined in `IOKit/firewire/IOFireWireFamilyCommon.h`.

Most of the message constants are defined in `IOKit/IOMessage.h`. Please refer to the "Notifier" example in the FireWire SDK for a sample implementation.

1.5 Obtaining properties of a connected device

Many services in the kernel have properties and values that are very useful. FireWire device objects have a GUID property which contains the GUID of the device, for example. You can obtain properties of a device using the two functions `IORegistryEntryCreateCFProperty()` and `IORegistryEntryCreateCFProperties()`, defined in `IOKit/IOKitLib.h` Note that you do not need to open a user client to do this. Here is a table of FireWire related registry classes and properties of interest:

1. General - IOKitLib

Class Name	Property Key	Description *[Type]*
IOFireWireDevice	`FireWire Node ID`	Current node ID of the device. *[CFNumber]*
	`Vendor_ID`	Device vendor ID. *[CFNumber]*
	`FireWire Device ROM`	Currently read contents of device config ROM. Note: this may not contain the entire ROM contents. *[CFData]*
	`FireWire Vendor Name`	*[CFString]*
	`FireWire Self IDs`	Self IDs received for this device. *[CFData]*
	`FireWire Speed`	Current operating speed of the device. *[CFNumber]*
	`GUID`	Device GUID. *[CFNumber]*
IOFireWireUnit	`Vendor_ID`	Device vendor ID. *[CFNumber]*
	`FireWire Vendor Name`	*[CFString]*
	`GUID`	Device GUID. *[CFNumber]*
	`Unit_Spec_ID`	Unit spec ID for this unit. *[CFNumber]*
	`Unit_SW_Version`	Unit software version for this unit. *[CFNumber]*
	`FireWire Product Name`	*[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 SBP2 target. *[CFNumber]*
	`Command_Set_Spec_ID`	SBP2 command set spec ID. *[CFNumber]*
	`Vendor_ID`	Device vendor ID. *[CFNumber]*
	`Device_Type`	SBP2 device type. *[CFNumber]*
	`Firmware_Revision`	SBP2 firmware revision
	`Unit_SW_Version`	Unit software version for this unit. *[CFNumber]*
IOFireWireAVCUnit	`Unit_Spec_ID`	Unit spec ID for this unit. *[CFNumber]*
	`Unit_Type`	AVC Unit Type. *[CFNumber]*
	`Vendor_ID`	Device vendor ID. *[CFNumber]*
	`GUID`	Device GUID. *[CFNumber]*
	`Unit_SW_Version`	Unit software version for this unit. *[CFNumber]*
	`FireWire Product Name`	*[CFString]*
	`GUID`	Device GUID. *[CFNumber]*

Table 1. 1

1.6 Finding provider classes in the registry

You can also use IOKit to find the provider of a service. For example, you might want a find the device which contains the unit you are connected to. Call `IORegistryEntryGetParentEntry()` to get the parent service of a service. Each call to `IORegistryEntryGetParentEntry()` will return the parent of the passed in service. Use `IOServiceConformsTo()` to determine the class of parent service. For example, when finding the parent of an IOFireWireUnit class instance, we might check that the parent is an IOFireWireDevice.

1.7 Loading plugins

All the FireWire user space interfaces are contained in plugins, more specifically, CFPlugIn's. These are dynamically loadable software bundles. To control or communicate with any device on the FireWire bus from user space, a CFPlugIn must be loaded.

Loading a CFPlugIn, i.e. IOFireWireLib, IOFireWireSBP2Lib, and IOFireWireAVCLib is a two step process: First obtain the plugin's IUnknown interface. Then, call `QueryInterface()` on the IUnknown interface. This will return a reference to the specific interface in which you are interested.

For example, to open the FireWire user client on a FireWire unit and obtain an IOFireWireDeviceInterface, first call `IOCreatePlugInInterfaceForService()`.Pass the service reference obtained from matching as the service parameter. Pass the constant for the plugin you are trying to open in `pluginType` *(see the table below)*. For `interfaceType`, pass `kIOCFPlugInInterfaceID` *(IOKit/IOCFPlugIn.h)*. This will return an IUnknown interface from IOFireWireLib.

Plugin Name	PluginType constant	Defined In
IOFireWireLib	`kIOFireWireLibTypeID`	`IOKit/firewire/IOFireWireLib.h`
IOFireWireSBP2Lib	`kIOFireWireSBP2LibTypeID`	`IOKit/sbp2/IOFireWireSBP2Lib.h`
IOFireWireAVCLib	`kIOFireWireAVCLibUnitTypeID` *(for IOFireWireLibAVCInterface)*	`IOKit/avc/IOFireWireAVCLib.h`
	`kIOFireWireAVCLibProtocolTypeID` *(for IOFireWireLibAVCProtocolInterface)*	`IOKit/avc/IOFireWireAVCLib.h`

Table 1. 2

Once the IUnknown interface is obtained, call `QueryInterface()` on it and specify a universally unique ID (UUID) corresponding to the interface and version you wish to obtain. See the table below for a list of UUIDs and corresponding plugin interfaces. In this example use `kIOFireWireDeviceInterfaceID_v3` to obtain version 3 of the IOFireWireDeviceInterface. All interfaces obtained through `QueryInterface()` should be released by calling `Release()`. The interface obtained from `IOCreatePlugInInterfaceForService()` should always be released by calling `IODestroyPlugInInterface()`.

Plugin Name	Interface Name	Vers.	UUID
IOFireWireLib	IOFireWireDeviceInterface	—	`kIOFireWireDeviceInterfaceID`
		v2	`kIOFireWireDeviceInterfaceID_v2`
		v3	`kIOFireWireDeviceInterfaceID_v3`
IOFireWireSBP2Lib	IOFireWireSBP2LibLUNInterface	—	`kIOFireWireSBP2LibLunInterfaceID`
IOFireWireAVCLib	IOFireWireAVCLibUnitInterface	—	`kIOFireWireAVCLibUnitInterfaceID`
	IOFireWireAVCLibProtocolInterface	—	`kIOFireWireAVCLibProtocolInterfaceID`

Table 1. 3

1. General - IOKitLib

Almost all of the SDK sample projects demonstrate how an interface is obtained and released. For more information on CFPlugIn, please consult the CoreFoundation documentation.

3. IOFireWireLib – Isochronous

2 IOFireWireLib – Basics & Asynchronous

2.1 Establishing a connection to a device

You create an instance of IOFireWireDeviceInterface to communicate with a device on the FireWire bus. You should match against individual units instead of entire devices when possible *(see "Establishing a connection to a unit," below)*.

To create an instance of IOFireWireDeviceInterface, you must first find the device you wish to control. (*See "Creating a matching dictionary to locate and match devices in the registry," above)*. The IOKit class name of interest is IOFireWireDevice.

IOFireWireDeviceInterface is the most important interface in IOFireWireLib. There are other interfaces in IOFireWireLib, however, all device and unit communication starts with an instance of IOFireWireDeviceInterface.

2.2 Establishing a connection to a unit

You create an instance of IOFireWireDeviceInterface to communicate with units in devices on the FireWire bus.

To create an instance of IOFireWireDeviceInterface, you must first find the unit you wish to control. See "Creating a matching dictionary to locate and match devices in the registry," above. The IOKit class name of interest is IOFireWireUnit.

IOFireWireDeviceInterface is the most important interface in IOFireWireLib. There are other interfaces in IOFireWireLib, however, all device and unit communication starts with an instance of IOFireWireDeviceInterface.

2.3 Performing read, write and lock (compare-and-swap) transactions

There are two ways to perform read, write, and lock transactions on the FireWire bus: directly from the IOFireWireDeviceInterface or using FireWire command objects. Both methods allow you use 64-bit (absolute) addressing and 48-bit (device-relative) addressing.

The following IOFireWireDeviceInterface methods allow direct execution of read, write, and lock transactions. Note that these functions execute synchronously; the calling thread will block until the transaction is completed or times out. Use command objects *(see below)* if you require asynchronous completion.

A kernel command object is allocated and deallocated each time these functions are called. You can avoid this by using command objects.

Function	Usage
`Read()`	Block read
`ReadQuadlet()`	Quadlet read
`Write()`	Block write
`WriteQuadlet()`	Quadlet write
`CompareSwap()`	Quadlet lock (compare and swap)

Table 2. 1

2.4 Performing read, write and lock (compare-and-swap) transactions using command objects

Read, write, and lock transactions can also be performed using command objects. These objects are allocated using the **IOFireWireLibDeviceInterface** functions listed below

Function	Usage	UUID
CreateReadCommand	Creates an IOFireWireLibReadCommand for performing block reads and quadlet reads	`kIOFireWireReadCommandInterfaceID_v2`
CreateWriteCommand	Creates an IOFireWireLibWriteCommand for performing block writes and quadlet writes	`kIOFireWireWriteCommandInterfaceID_v2`
CreateCompareSwapCommand	Creates an IOFireWireLibCompareSwapCommand for performing 32-bit and 64-bit compare-and-swap transactions	`kIOFireWireCompareSwapCommandInterfaceID_v2`
CreateReadQuadletCommand	(obsolete)	
CreateWriteQuadletCommand	(obsolete)	

Table 2. 2

Once a command object is created, it is configured and submitted. Command objects can be executed synchronously or asynchronously, and configured and submitted multiple times. Use the `Submit()` method to begin command execution.

2.5 Receiving read, write, and lock (compare-and-swap) transactions

A client can respond to read, write, and lock requests using an “address space”. An address space in this sense is a portion of the Macintosh’s 48-bit bus-addressable memory which is reserved for a software client’s use. Read, write, and lock transactions from the bus to addresses in the allocated range are sent to client software, allowing it to receive communication from the bus.

There are two major types of address spaces: pseudo address spaces and physical address spaces.

Physical address spaces occupy areas of the Macintosh’s FireWire address space that map directly to physical memory in the Macintosh. Accesses to these address spaces can be made with no software interactionl incoming data is read from or written directly to physical memory with no client interaction. Physical address spaces can be used for large transfers where the client does not need to inspect each packet of data sent or received. Physical address spaces may appear non-contiguously in the Macintosh’s FireWire address space.

Pseudo-address spaces are “virtual”; FireWire addresses within a pseudo-address space do not map directly to addresses in the Macintosh’s main memory. Each packet written to or read from a pseudo-address space must be acknowledged by software, although clients can request that the FireWire family handle these accesses for them. There are two types of pseudo address spaces: plain and initial units space.

Initial units space pseudo-address spaces are special type of pseudo-address space. They a) reside in the FireWire initial units space address range of the Macintosh and b) they can be shared among multiple clients.

Address space type		Function	UUID
Pseudo	Plain	`CreatePseudoAddressSpace()`	`kIOFireWirePseudoAddressSpaceInterfaceID`
	Initial Units	`CreateInitialUnitsPseudoAddressSpace()`	`kIOFireWirePseudoAddressSpaceInterfaceID`
Physical		`CreatePhysicalAddressSpace()`	`kIOFireWirePhysicalAddressSpaceInterfaceID`

Table 2. 3

2.6 Browsing remote device config ROMs

You can use IOFireWireLib to read a device’s config ROM. This is by creating an instance of IOFireWireConfigDirectoryInterface which becomes the interface to the device’s config ROM. Call `GetConfigDirectory()` and pass kIOFireWireConfigDirectoryInterfaceID in *iid*.

If you have obtained the device interface for a FireWire dev ice, `GetConfigDirectory()` returns the config ROM root directory. If you obtained the device interface for a FireWire unit, `GetConfigDirectory()` returns the unit directory for that unit.

2.7 Creating local unit directories

The FireWire user client can be used to create unit directories and make them appear in the Macintosh’s config ROM. First create an IOFireWireLocalUnitDirectory interface. This represents the unit directory you will be creating. Add entries to the unit directory using the methods of IOFireWireLocalUnitDirectoryInterface. To make the directory appear in the config ROM, call `Publish()`. To remove the unit directory from the config ROM, call `Unpublish()`