Writing Audio Device Drivers

March 1, 2004

Contents

Tables, Figures, and Listings

6

Introduction to Writing Audio Device Drivers

This book describes the architecture, services, and mechanisms of the I/O Kit's Audio family, and explains how you use the APIs of the family to write an audio device driver for Mac OS X. It does not cover any aspect of user-space audio programming (MIDI, synthesizers, CD players, and so on) except to discuss the overall composition of the Mac OS X audio system, which includes Core Audio and other audio frameworks.

To gain the most value from reading this book, it helps to be familiar with the I/O Kit and object-oriented programming, preferably C++ programming. The book *I/O Kit Fundamentals* provides a thorough introduction to the I/O Kit; see "Additional Information on the I/O Kit" (page 8) for details on this and other I/O Kit documentation.

What This Book Covers

This document describes all aspects of creating an audio device driver using the I/O Kit's Audio family. It includes conceptual and procedural information and consists of the following chapters:

1. "About This Book" (this chapter)

2. "Audio on Mac OS X" (page 11)

    Describes the features, benefits, and architecture of the Mac OS X audio system. It includes an overview of the audio I/O model.

3. "Audio Family Design" (page 21)

    Presents a comprehensive overview of the Audio family's architecture, classes, object relationships, and primary mechanisms. It also goes into more detail about the workings of the audio I/O model in Mac OS X.

4. "Implementing an Audio Driver" (page 39)

    Describes the various steps required to design and implement an audio device driver using the Audio family. Most steps are amply illustrated with sample code.

For details of specific methods, structures, and other API elements, consult the reference documentation for the Audio family. See "Additional Information on the I/O Kit" (page 8) for instructions on accessing this documentation.

Other Resources for Audio Developers

Apple offers several other resources to developers of audio software for Mac OS X. On the Apple Developer Connection website, you'll find at http://developer.apple.com/audio a page full of links to audio-related material. One of these links takes you to a page devoted to audio on Mac OS X (http://developer.apple.com/audio/macosxaudio.html). From this page you can download a document entitled *Audio and MIDI on Mac OS X*; this document describes the Core Audio framework (Audio HAL) as well as the secondary audio and MIDI frameworks.

Furthermore, if you install the Developer package, you get code examples relevant to audio development. Check out `/Developer/Examples/Kernel/IOKit/Audio` for some sample audio driver projects.

Additional Information on the I/O Kit

When you install the Developer package, you get developer documentation as well as tools and example code. A subset of this information is specific to Darwin and the I/O Kit. Here you can find a number of documents that will introduce you to I/O Kit programming:

- Overviews of the Darwin kernel, including *Kernel Programming*.

- The aforementioned *I/O Kit Fundamentals* describes the features, architecture, classes, and general mechanisms of the I/O Kit and includes discussions of driver matching and loading, event handling, memory management, and power management.

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

- *Kernel Extensions Tutorials*, a collection of tutorials that introduce you to the development tools and take you through the steps required to create, debug, and package kernel extensions and I/O Kit drivers (a type of kernel extension).

- Reference documentation on I/O Kit families and classes.

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

The documentation is in HTML or PDF format (and often is in both formats). You can access the HTML documentation (and download the PDF) from the Xcode Help menu. To view, click Help > Show Documentation Window. You can then search for specific API or view the entire developer documentation library. You can also access developer documentation on the Apple Developer Connection website at http://developer.apple.com/documentation/index.html.

Other Information on the Web

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

- Apple Developer Connection: Mac OS X (http://developer.apple.com/macosx) offers SDKs, release notes, product notes and news, and other resources and information related to Mac OS X.

- AppleCare Knowledge Base (http://kbase.info.apple.com) enables you to locate technical articles on Mac OS X (and other areas) using a natural language search.

Audio on Mac OS X

The audio system on Mac OS X is not a straightforward port of Mac OS 9 audio technology. It's true that some APIs, such as Carbon's Sound Manager, have been carried forward. But Apple has overhauled the audio system for Mac OS X, redesigning it from top to bottom.

This chapter gives an overview of audio on Mac OS X, describing it capabilities, its interrelated technologies, and its architecture. Reading this chapter will help you to understand how the I/O Kit's Audio family fits together and interacts with the other pieces of audio software on Mac OS X.

Mac OS X Audio Capabilities

In versions of system software prior to Mac OS X, the sound capabilities of a system largely depended on the availability of third-party audio and MIDI protocols and services. Apple has designed the Mac OS X audio system to consolidate, integrate, and standardize these services and protocols, thereby streamlining configuration of audio and MIDI devices and development of future audio and MIDI technologies.

Audio on Mac OS X comprises several audio technologies that, taken together, offer the following capabilities:

- Built-in support for a variety of audio formats, including formats based on pulse code modulation (PCM) and encoded formats such as AC-3 and MP3

- Multi-channel audio I/O that is scalable to a virtually unlimited number of channels

- Variable sample rates

- A remarkably clean signal path requiring little overhead

- Simultaneous access for multiple clients to all of the audio devices attached to the host, no matter how the connection is made (PCI, USB, FireWire, and so on)

Outside the kernel, Mac OS X represents audio as 32-bit floating point data; this format allows efficient processing of data with today's advanced audio peripherals (for example, those capable of 24-bit, 96 kiloherz operation) and ensures that the system can scale to future high-resolution formats.

Architecture of Mac OS X Audio

The audio capabilities of Mac OS X arise from several software technologies that are accessible through their public programming interfaces. These technologies are situated at different levels of the operating system where their relationships with each other can be characterized as client and provider. In other words, Mac OS X audio software is layered, with one layer dependent on the layer "under" it and communicating, through defined interfaces, with adjoining layers (see Figure 2-1 (page 12)). The relative locations of these technologies within the layers of system software suggest their degree of abstraction and their proximity to audio hardware. Some audio technologies in Mac OS X are incorporated into the kernel environment (that is, Darwin) while others are packaged as frameworks for use by application environments, applications, and other user processes.

Figure 2-1   Mac OS X audio layers
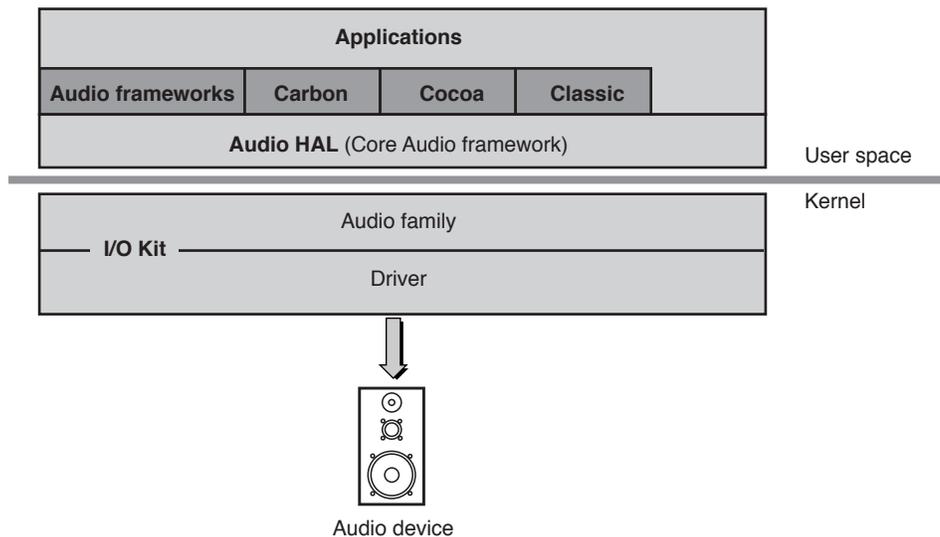
At the lowest level of the Mac OS X audio stack is the driver that controls audio hardware. The driver is based on the I/O Kit's Audio family, which provides much of the functionality and data structures needed by the driver; for example, the Audio family implements the basic timing mechanisms, the user-client objects that communicate with the upper layers, and maintains the sample and mix buffers (which hold audio data for the hardware and the hardware's clients, respectively). The basic role of the audio driver is to control the process that moves audio data between the hardware and the sample buffer. It is responsible for providing that sample data to the upper layers of the system when necessary, making any necessary format conversions in the process. In addition, an audio driver must make the necessary calls to audio hardware in response to format and control changes (for example, volume and mute).

Immediately above the driver and the I/O Kit's Audio family—and just across the boundary between kernel and user space—is the Audio Hardware Abstraction Layer (HAL). The Audio HAL functions as the device interface for the I/O Kit Audio family and its drivers. Its essential job is, for input streams, to make the audio data it receives from drivers accessible to its clients and, for output streams, to take the audio data from its clients and pass it "down" to a particular audio driver.

Built on top of the Audio HAL, which is implemented in the Core Audio framework, are two other frameworks that provide specialized audio services: the Audio Units and the Audio Toolbox frameworks. MIDI System Services, which comprises two other frameworks, is not directly dependent on the Audio HAL but does (as its name suggests) make MIDI services available to applications and presents an API for creating MIDI drivers.

Finally, the ultimate clients of audio on Mac OS X—applications, frameworks, and other user processes—can directly access the Audio HAL or indirectly access it through one of the higher-level audio frameworks. They can also indirectly access the Audio HAL through the audio-related APIs of the application environments they belong to: Sound Manager in Carbon, NSSound in Cocoa, and the Java sound APIs.

The following sections examine each of these audio technologies of Mac OS X in more detail.

Audio HAL (Core Audio)

The Audio Hardware Abstraction Layer (HAL) is the layer of the Mac OS X audio system that acts as an intermediary between the I/O Kit drivers controlling audio hardware and the programs and frameworks in user space that are clients of the hardware. More specifically, the Audio HAL is the standardized device interface for the I/O Kit's Audio family. It is implemented in the Core Audio framework (`CoreAudio.framework`) and presents both C-language and Java APIs. In the Audio HAL, all audio data is in 32-bit floating point format.

The API of the Audio HAL is divided among three abstractions: audio hardware, audio device, and audio stream.

- The audio hardware API gives clients access to audio entities that exist in the "global" space, such as the list of current devices and the default device.

- The audio device API enables clients to manage and query a specific audio device and the I/O engines that it contains. An audio device in the Audio HAL represents a single I/O cycle, a clock source based on it, and all the buffers that are synchronized to this cycle. The audio-device functions permit a client to, among other things, start and stop audio streams, retrieve and translate the time, and get and set properties of the audio device.

- The audio stream API enables a client to control and query an audio stream. Each audio device has one or more audio streams, which encapsulate the buffer of memory used for transferring audio data across the user space/kernel boundary. They also record the format of the audio data.

The abstractions of audio device and audio stream loosely correspond to different I/O Kit Audio family objects in the kernel (see "The Audio Family" (page 15)). For example, the entity referred to as "audio device" in the Audio HAL corresponds to a combination of an IOAudioDevice and IOAudioEngine in the kernel. For each IOAudioEngine the Audio HAL finds in the kernel, it generates an audio-device identifier. However, there is considerable overlap of role among the various Audio family and Audio HAL objects and entities.

A critical part of the APIs for audio hardware, devices, and streams involves audio properties and their associated notifications. These APIs allow clients to get and set properties of audio hardware. The "get" functions are synchronous, but the "set" functions work in an asynchronous manner that makes use of notifications. Clients of the Audio HAL implement "listener procs"—callback functions for properties associated with audio hardware, audio devices, or

audio streams. When an audio driver changes a property of the hardware, either as a result of user manipulation of a physical control or in response to a "set" function, it sends notifications to interested Audio HAL clients, which results in the appropriate "listener procs" being called.

Just as important as the property APIs is the callback prototype (`AudioDeviceIOProc`) that the audio-device subset of the Audio HAL API defines for I/O management. Clients of the Audio HAL must implement a function conforming to this prototype to perform I/O transactions for a given device. Through this function, the Audio HAL presents all input and output simultaneously in an I/O cycle to the client for processing. In this function, a client of the Audio HAL must send audio data to the audio device (for output), or copy and process the audio data received from the audio device (for input).

Secondary Audio Frameworks

Mac OS X has several frameworks other than the Core Audio framework that offer audio-related functionality to applications. Two of these frameworks—Audio Units and Audio Toolbox—are built directly on the Core Audio framework. MIDI System Services (consisting of the Core MIDI and Core MIDI Server frameworks) does not directly depend on the Core Audio framework, but is still a consumer of the services of the audio frameworks.

All of these secondary frameworks are implemented in the C language and present their public programming interfaces in C. Thus, any application or other program in any application environment can take advantage of their capabilities.

Audio Units

The Audio Units framework (`AudioUnits.framework`) offers applications the capability for generating, processing, receiving, and manipulating or transforming streams of audio data. This functionality is based on the notion of audio units.

Audio units are building blocks called components that clients can either use singly or can connect together to form an audio signal graph. A component is a piece of code that provides a defined set of services to one or more clients. Thus an audio unit is a processing unit that can have one or more inputs and one output, which is generally a buffer of audio data. The inputs can have a variety of sources, such as encoded audio data, synthesizers, or even other audio units. To compose an audio signal graph, clients can use the AUGraph API in the Audio Toolbox framework—see "Audio Toolbox" (page 15) for details.

Using a "pull I/O" model, an audio unit specifies the number and format of its inputs and outputs through its properties. Each output is in itself a stream of an arbitrary number of interleaved audio channels derived from the audio unit's inputs. Clients also manage the connections between units through properties.

Examples of audio units are DSP processors (such as reverbs, filters, and mixers), format converters (for example, 16-bit integer to floating-point converters), interleavers-deinterleavers, and sample rate converters. In addition to defining the interface for custom audio units in the Audio Units framework, Apple ships a set of audio units. One of these is the MusicDevice component, which presents an API targeted specifically toward software synthesis.

Audio Toolbox

The Audio Toolbox framework (`AudioToolbox.framework`) complements the Audio Units framework with two major abstractions: the AUGraph and the Music Player.

An AUGraph provides a complete description of an audio signal processing network. It is a programmatic entity that represents a set of audio units and the connections (input and output) among them. With the AUGraph APIs, you can construct arbitrary signal paths through which audio can be processed. Audio graphs enact real-time routing changes while audio is being processed, creating and breaking connections between audio units "on the fly." It maintains the representation of the graph even when constituent audio units have not been instantiated.

The Music Player APIs use AUGraphs to provide the services of a sequencing toolbox that collects audio events into tracks, which can then be copied, pasted, and looped within a sequence. The APIs themselves consist of a number of related programmatic entities. A Music Player plays a Music Sequence, which can be created from a standard MIDI file. A Music Sequence contains an arbitrary number of tracks (Music Tracks), each of which contains time-stamped audio events in ascending temporal order. A MusicSequence usually has an AUGraph associated with it, and a Music Track usually addresses its audio events to a specific Audio Unit within the graph. Events can involve tempo and extended events, as well as regular MIDI events.

The Audio Toolbox framework also includes APIs for converting audio data between different formats.

MIDI System Services

MIDI System Services is a technology that allows applications and MIDI devices to communicate with each other in a single, unified way. It comprises two frameworks: Core MIDI (`CoreMIDI.framework`) and Core MIDI Server (`CoreMIDIServer.framework`).

MIDI System Services gives user processes high-performance access to MIDI hardware. In a manner similar to the Audio HAL, MIDI System Services implements a plug-in interface that enables clients to communicate with a MIDI device via a driver (which, incidentally, isn't an I/O Kit driver) that is loaded as a CFPlugIn bundle from `/System/Library/Extensions` or `Library/Audio/MIDI Drivers`. Apple provides two default MIDI drivers: `AppleMIDISerialDriver.plugin` and `AppleMIDIUSBDriver.plugin`. Using the driver model specified by the MIDI System Services APIs, third-party MIDI manufacturers can create their own driver plug-ins that can "talk" with the I/O Kit. A MIDI server can then load and manage those drivers. Applications can communicate with the MIDI drivers through the client-side APIs of the Core MIDI framework. The Core MIDI Server framework is used by those developers who provide drivers for MIDI devices.

The Audio Family

The I/O Kit's Audio family facilitates the creation of drivers for audio hardware. Drivers created through the Audio family can support any hardware on the system, including PCI, USB and FireWire devices. Essentially, an I/O Kit audio driver transfers audio data between the hardware and the Audio HAL. It provides one or more sample buffers as well as a process that moves data between the hardware and the sample buffer. Typically this is done with the audio hardware's DMA engine. Because the native format of audio data on Mac OS X is 32-bit floating point, the driver must provide routines to convert between the hardware format of the data in the sample

buffer and 32-bit floating point. The sequence of steps that a driver follows depends on the direction of the stream. For example, with input audio data, the driver is asked for a block of data. It obtains it from the sample buffer, converts it to the expected client format (32-bit floating point), and returns it. That data is then passed by the family to the Audio HAL through a user-client mechanism.

The interactions between the DMA engine and the driver, and the driver and the Audio HAL, are based on the assumption that, in any one direction, the stream of audio data proceeds at the same rate, continuously. The Audio family sets up several timers (based on regularly taken time stamps) to synchronize the actions of the agents involved in this transfer of data. These timing mechanisms ensure that the audio data is processed at maximum speed and with minimum latency. Take again an input stream as an example. Shortly after the DMA engine writes sample frames to the driver's sample buffer, the driver reads that data, converts the integer format to 32-bit floating point, and writes the resulting frames to the mixer buffer, from whence they are passed on to the Audio HAL. Just before the DMA engine writes new frames to the same location in the sample buffer, an "erase head" zero-initializes the just-processed frames in the sample buffer. For more on the sample buffer and the timer mechanisms used by the Audio family, see .

An I/O Kit audio driver consists of a number of objects, the most important of which are derived from the IOAudioDevice, IOAudioEngine, IOAudioStream, and IOAudioControl classes. These objects perform the following roles for the driver:

■ A single instance of a custom subclass of IOAudioDevice represents the audio device itself. The IOAudioDevice subclass is the root object of a complete audio driver. It is responsible for mapping all hardware resources from the service-provider's nub and for controlling all access to the hardware (handled automatically through a provided command gate). An IOAudioDevice object manages one or more IOAudioEngine objects.

■ An audio driver must contain one or more instances of a custom subclass of IOAudioEngine. This custom subclass manages each audio I/O engine associated with the audio device. Its job is to control the process that transfers data between the hardware and a sample buffer. Typically the I/O process is implemented as a hardware DMA engine (although it doesn't have to be). The sample buffer must be implemented as a ring buffer so that when the I/O process of a running IOAudioEngine reaches the end of the buffer, it wraps back around to the beginning and keeps going.

    An IOAudioEngine object is also responsible for starting and stopping the engine, and for taking a time stamp each time the sample buffer wraps around to the beginning. It contains one or more IOAudioStream objects and can contain any number of IOAudioControl objects.

■ An instance of IOAudioStream represents a sample buffer, the associated mix buffer, and the direction of the stream. The IOAudioStream object also contains a representation of the current format of the sample buffer as well as a list of allowed formats for that buffer.

■ An instance of IOAudioControl represents any controllable attribute of an audio device, such as volume or mute.

An I/O Kit audio driver uses two user-client objects to communicate with the Audio HAL layer. The Audio HAL communicates with the IOAudioEngine and IOAudioControl objects through, respectively, the IOAudioEngineUserClient and IOAudioControlUserClient objects. The audio system creates these objects as they are needed. The IOAudioEngineUserClient class provides the main linkage to an IOAudioEngine subclass; it allows the Audio HAL to control the

IOAudioEngine and it enables the engine to pass notifications of changes back to the Audio HAL. For each IOAudioControl object in the driver, an IOAudioControlUserClient object passes notifications of value changes to the Audio HAL.

For more detailed information on the classes and general architecture of the Audio family, see the chapter "Audio Family Design" (page 21).

Apple Audio Drivers

Apple ships several audio device drivers with a standard Mac OS X installation. These drivers are suitable for much of the audio hardware commonly found on Power PC computers. The "onboard driver" kernel extension—`AppleOnboardAudio.kext`—contains almost a half dozen audio drivers packaged as plug-ins. Each of these drivers is based on a specific subclass of IOAudioDevice and each uses the code in the AppleDBDMAAudio kernel extension for the IOAudioEngine subclass. The I/O Kit, through its matching process, finds and loads the appropriate plug-ins based on existing audio hardware. For USB audio hardware, Apple includes the driver defined in the `AppleUSBAudio.kext` kernel extension.

Important
The set of audio drivers provided by Apple might change at any time.

The Audio I/O Model on Mac OS X

Mac OS 9 and Mac OS X perform audio I/O in much different ways. The differences between them are most salient in the lower layers of the audio stack, particularly the audio driver model and the audio access libraries.

In Mac OS 9, an audio driver's DMA engine transfers audio data between a sample buffer, which is provided by the driver, and the hardware. The buffer holds a segment of the audio data containing a sequence of sample frames in temporal order. The Mac OS 9 driver model uses double buffering to exchange audio data between the driver and its clients, so there are actually two sample buffers. In the case of audio output, after the driver's clients (using the Sound Manager API) fill one of the buffers, the hardware (usually through its DMA engine) signals the driver (typically through an interrupt) that it is finished playing the other buffer and ready for a more data. The driver then gives the hardware the buffer it just filled, receives the just-played buffer from the hardware, and signals the application that it needs more data.

Figure 2-2   Access to the sample buffer on Mac OS 9

Audio device
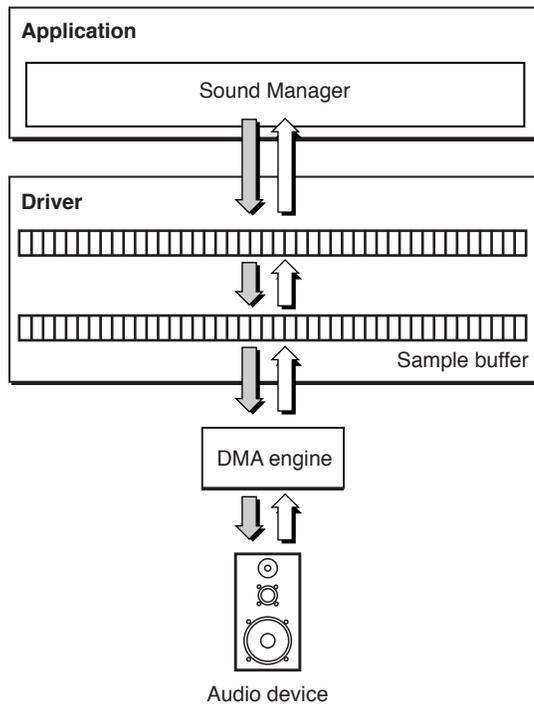
The architecture and goals of Mac OS X made this design untenable. With the Mac OS X kernel, an audio driver incurs a greater cost than on Mac OS 9 when it signals an application that more audio data is needed (or that new data is available). Moreover, a major goal of the Mac OS X audio system is to support multiple simultaneous clients, which is not possible with the Mac OS 9 model. A new audio I/O model was needed not only for this goal, but to provide the highest possible performance and the lowest possible latency. Figure 2-3 (page 19) depicts the audio I/O model on Mac OS X.

Figure 2-3   The Mac OS X audio model
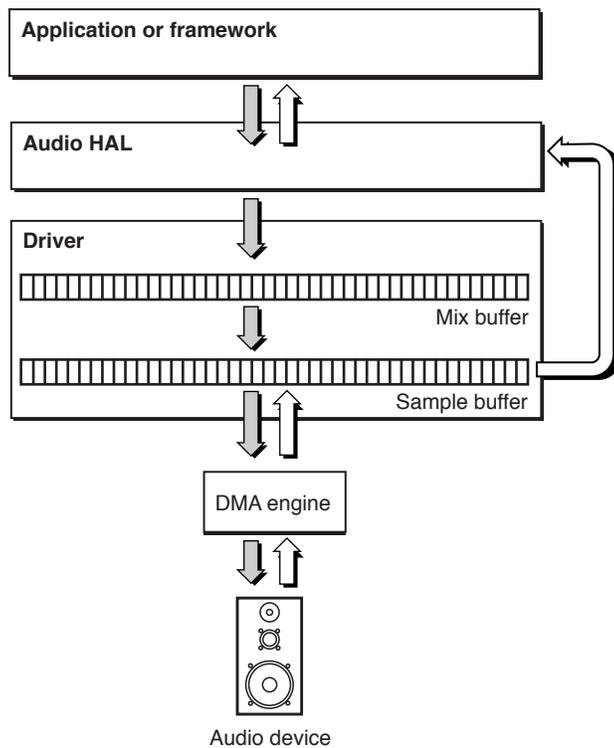
The key facet of the Mac OS X audio I/O model involves predictive timing mechanisms. Instead of requiring the driver to message an application directly when an I/O cycle has completed, the timing mechanisms enable the Audio HAL to predict when the cycle will complete; the Audio HAL uses the extremely accurate timing code on Mac OS X to ensure that clients perform their I/O at the proper time, based on the size of their buffers. The audio driver does its part to make this possible by setting up the hardware's sample buffer as a ring buffer and by taking an accurate timestamp every time the I/O engine wraps to the beginning of the buffer.

The Audio HAL keeps track of each timestamp and uses the sequence of timestamps to predict the current location of the audio I/O engine (in terms of sample frame read or written) at any time. Given that information, it can predict when a cycle will complete and sets its wake-up timestamp accordingly. This model, combined with the ability of the I/O Kit Audio family to receive audio data from each client asynchronously, allows any number of clients to provide audio data that gets mixed into the final output. It also allows different client buffer sizes; one client can operate at a very low buffer size (and a correspondingly low latency) while at the same time another client may use a much larger buffer. As long as the timestamps provided by the driver are accurate, the family and the Audio HAL do all of the work to make this possible.

Another important difference between the audio I/O model on Mac OS 9 and the one on Mac OS X is the native format of audio data in the system. In Mac OS 9, because the application (through the Sound Manager) has direct access to the hardware buffer, it has to deal with the native hardware format. Because of this reality, the Mac OS 9 audio libraries only support 16-bit 1-channel or 2-channel PCM audio data to simplify things. In Mac OS X, an application cannot directly access the sample buffer. This indirection permits the use of the 32-bit floating point format between the Audio HAL and an audio driver. Consequently, the driver is responsible for providing a routine that can clip and convert that 32-bit floating point output data into the buffer's

native format. It might also have to implement a routine to convert input data into 32-bit floating point. Both routines are called asynchronously as Audio HAL clients pass audio data to the driver and receive data from it.

For detailed information on the Mac OS X audio I/O model, see "The Audio I/O Model Up Close" (page 31).

Audio Family Design

All audio drivers, regardless of platform, must perform the same general actions. For input streams, drivers receive digital audio data from the hardware in a stream of frames consistent with the current sampling rate and audio format. They modify the data, if necessary, to a form acceptable to the clients of the device (say, 32-bit floating point) and make the altered frames accessible to those clients at the current sampling rate. In the reverse (output) direction, the job of the audio driver is essentially the same. It accepts digital audio data from the clients of the device, changes that stream of sample frames to a form required by the hardware (say, 16-bit integer), and gives the data to the device's controller at the current sampling rate.

Drivers must also initially configure the hardware, respond to client requests to change device attributes (for example, volume), and notify clients when some attribute or state of the audio device has changed. They must guard against data corruption in a multithreaded environment, and they must be prepared to respond to system-wide events, such as sleep and wake transitions.

The Audio family provides object-oriented abstractions to help your driver deal with many of these things. The family itself takes care of much of the work for you; you just supply the behavior that is specific to your hardware. To do this, it is useful know how your code fits together with the family implementation, which is what this chapter is about.

The Classes of the Audio Family

As you can with any object-oriented system, you can come to an understanding of the design of the I/O Kit's Audio family by examining the classes of the family. The examination in this section looks at the roles of the family, the properties they encapsulate, the audio entities they represent, and the relationships they have with each other. The relationships considered are not only the static relationships imposed by inheritance, but the dynamic relationships characterized by containment, dependency, and control.

The Audio family consists of about a dozen classes, all having the prefix "IOAudio":

- IOAudioDevice
- IOAudioEngine
- IOAudioStream
- IOAudioControl

■ IOAudioPort

■ IOAudioEngineUserClient

■ IOAudioControlUserClient

■ IOAudioLevelControl

■ IOAudioSelectorControl

■ IOAudioToggleControl

The inheritance relationships among these classes, as depicted in Figure 3-1 (page 22), are uncomplicated.

Figure 3-1   The Audio family class hierarchy

```
OSObject ─── IORegistryEntry ─── IOService ──┬── IOAudioControl ──┬── IOAudioJackControl
                                             │                    ├── IOAudioLevelControl
                                             ├── IOAudioDevice    ├── IOAudioSelectorControl
                                             ├── IOAudioEngine    └── IOAudioToggleControl
                                             ├── IOAudioPort
                                             ├── IOAudioStream
                                             │                    ┌── IOAudioControlUserClient
                                             └── IOUserClient ─────┴── IOAudioEngineUserClient
```
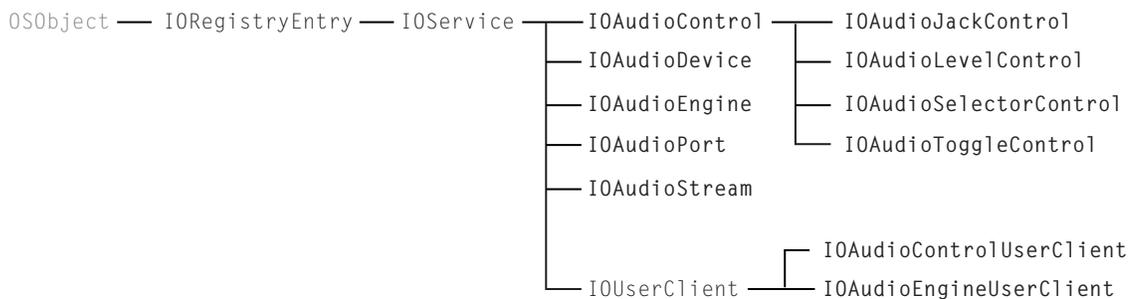
All classes of the Audio family directly or indirectly inherit from IOService; thus objects of these classes are full-fledged driver objects, with the capability for responding to driver life-cycle messages and for participating in the matching process. In practice, however, an instance of an IOAudioDevice subclass, as root object of the audio driver, usually matches against the provider's nub (the provider being a PCI controller or FireWire or USB device, in most cases). Audio drivers are typically "leaf" objects in the driver stack, and typically their only client is the Audio HAL, in user space. Therefore they do not publish nubs of their own.

Two classes, IOAudioEngineUserClient and IOAudioControlUserClient, inherit from the IOUserClient class. Objects of these classes represent user-client connections that enable the Audio family to communicate with the Audio HAL. Five of the Audio family classes are subclasses of IOAudioControl, providing behavior specific to certain types of audio-device controls (mute switches, volume controls, and so on). For further details on the user-client and control classes of the Audio Family, see "The Roles of Audio Family Objects" (page 25).

An understanding of the static inheritance relationships between classes of the Audio family goes only so far to clarify what instances of those classes do in a typical audio driver. It is more illuminating to consider the dynamic relationships among these objects.

Dynamic Relationships in the Audio Family

An I/O Kit audio driver consists of a variable number of objects that represent or encapsulate certain aspects of an audio device. These objects maintain relationships with one another that are characterized by "ownership" or containment. The most significant objects in a "live" audio driver derive from four Audio family classes:

- IOAudioDevice

- IOAudioEngine

- IOAudioStream

- IOAudioControl

Figure 3-2 (page 23) illustrates the dynamic relationships of these objects.

Figure 3-2   Audio family objects in a typical driver and what they represent
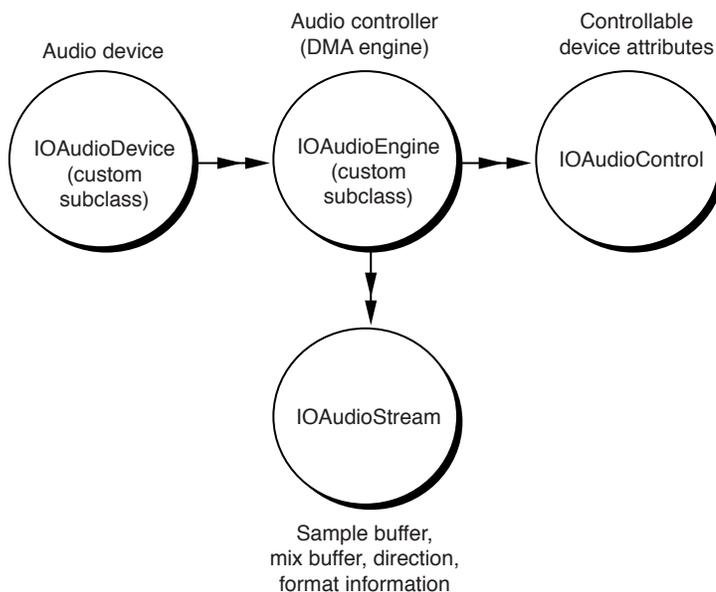
The root object in an audio driver is an instance of a custom subclass of IOAudioDevice. It represents an audio device in a general, overall sense. An IOAudioDevice object is the root object for a couple reasons: It creates and coordinates many of the other objects in the driver, and it is typically the object that must match against the provider's nub.

The custom subclass of IOAudioDevice adds attributes and implements behavior that are specific to the device. It is responsible for identifying, configuring, and creating all necessary audio-engine objects and attaching those objects to itself. It must map all hardware resources from the provider's nub and, when requested by the system, it must change the values of controls. Furthermore, an IOAudioDevice object is the power controller and power policy maker for the driver; in coordination with its IOAudioEngine objects, it must properly determine system idleness and deal with power-state transitions (sleep and wake), deactivating and reactivating its audio engines as necessary.

A driver's IOAudioDevice object contains one or more IOAudioEngine objects as instance variables. Each of these objects is an instance of a custom subclass of IOAudioEngine. An IOAudioEngine object represents the I/O engine (usually a DMA engine) of the audio device; its job is to transfer audio data to or from one or more sample buffers and the hardware. The object starts and stops the audio I/O engine when requested; once started, it should run continuously, looping through the sample buffers until stopped. While it is running, an IOAudioEngine takes a time stamp and increments a loop count each time it "wraps around" a

sample buffer (see "The Audio I/O Model Up Close" (page 31). The Core Audio framework (Audio HAL) uses this timing information to calculate the exact position of the audio engine at any time.

An audio driver needs only one IOAudioEngine object unless it needs to manage sample buffers of different sizes or to have sample frames transferred at different rates. In these cases, it should instantiate and configure the required number of IOAudioEngines.

An IOAudioEngine object itself contains one or more instances of the IOAudioStream class. An IOAudioStream object primarily represents a sample buffer, which it encapsulates. It also encapsulates the mix buffer for an output audio stream. It describes the direction of the stream as well as the format information that can be applied to the sample buffer. The format information includes such data as number of channels, sampling format, and bit depth. If a sample buffer has multiple channels, the channels are typically interleaved (although separate IOAudioStreams can be used to represent non-interleaved different channels). Often an audio engine has one IOAudioStream object for an input stream and another for an output stream.

An IOAudioEngine also contains one or more IOAudioControl objects as instance variables. Such an object represents a controllable attribute of the audio device, such as mute, volume, or master gain. An IOAudioControl is usually associated with a specific channel in a specific stream. However, it can control all channels of an IOAudioStream or even all channels of an IOAudioEngine. At hardware-initialization time, an IOAudioEngine (or perhaps the driver's IOAudioDevice object) creates the necessary IOAudioControls and adds them to the appropriate IOAudioEngine.

Each IOAudioControl is known as a "default control" because the Audio HAL recognizes and uses it based on its attributes. Most value changes to the controls originate with clients of the Audio HAL, which passes them via the user-client interface to the designated value-change handlers in the driver. Notifications of value changes in audio controls can also travel in the other way; for example, if a user turns the volume knob on a speaker, the driver communicates this change to Audio HAL clients.

The Audio Family and the I/O Registry

As it does with all I/O Kit drivers, the I/O Registry captures the client-provider relationships and the properties of audio drivers. By using the I/O Registry Explorer application or the `ioreg` command-line tool, you can view the objects of "live" audio drivers in the I/O Registry and examine the properties of those objects. The visual presentation that these tools make possible clarifies the client-provider relationships among Audio-family objects and the relationships between audio objects and other objects in the driver stack. You can use these tools to verify driver status and for debugging problems.

Figure 3-3 (page 25) shows how some of the objects in a USB audio driver appear in the I/O Registry.

Figure 3-3   A USB audio driver displayed in the IORegistryExplorer application
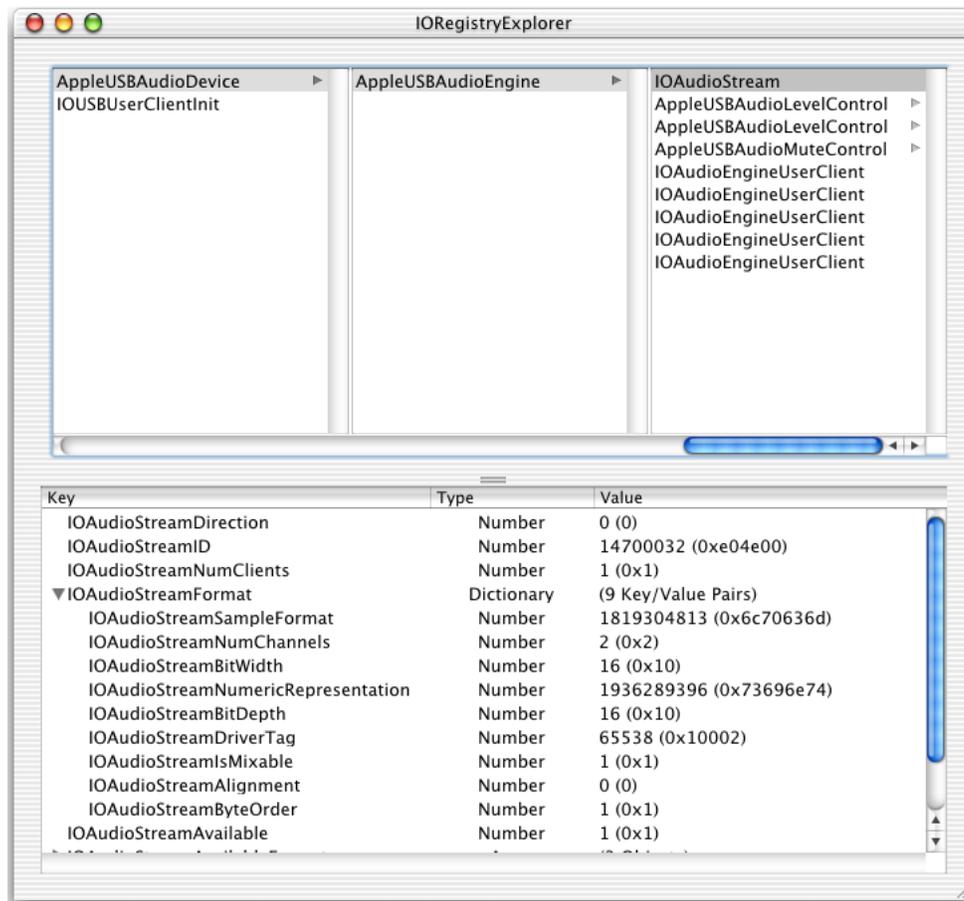
In addition to these programming uses, the I/O Registry also serves a critical function in the architecture of the Mac OS X audio system. The Audio HAL uses I/O registry for to find the control attributes of drivers. The control properties of an audio driver—volume, mute, gain settings—are stored in the I/O Registry and are associated with an IOAudioControl object. This The Audio HAL looks for, recognizes, and uses the object based on its attributes, which it discovers through the I/O Registry. See "IOAudioControl" (page 28) for further information.

For details on matching properties, device attributes, and other I/O Registry keys, see the header file `Kernel.framework/Headers/IOKit/audio/IOAudioDefines.h` or the associated reference documentation.

The Roles of Audio Family Objects

The previous section looked in a general way at the major objects in a "live" audio driver, describing what those objects basically do and what their relationships are with one another. This section probes a little deeper and examines the roles of all Audio family classes (and objects) in more detail.

IOAudioDevice

Every audio driver based on the Audio family must have one instance of a custom subclass of IOAudioDevice. The driver's IOAudioDevice object is the central, coordinating node of the driver's object tree—the "root" object. All other objects ultimately depend on it or are contained by it.

Because of its status as root object, the IOAudioDevice object represents the audio hardware generally. This central position gives it several roles:

- It is the object that usually matches against the provider's nub.

- It initializes the device, mapping hardware resources from the provider's nub, and otherwise reads and writes to the device registers as necessary.

- It creates the IOAudioEngine objects of the driver and can create the IOAudioControl objects used by the driver.

- It usually implements value changes in the audio controls associated with its audio-engine objects.

- It acts as the power controller and policy maker for the audio hardware; in this role, it must respond to system sleep, system wake, and domain idleness transitions by deactivating and reactivating its audio engines as necessary.

The driver's IOAudioDevice object fulfills other functions. It controls access to the audio hardware to ensure, in the driver's multithreaded environment, that the hardware doesn't get into an inconsistent state. Toward this end, the IOAudioDevice superclass provides a separate work loop (IOWorkLoop) and a command gate (IOCommandGate) to synchronize access by all objects in the driver. All other objects in the driver—IOAudioEngine, IOAudioStream, and IOAudioControl—contain references to the IOAudioDevice's work loop and the command gate as instance variables. All Audio family classes take care of executing I/O and hardware-related code on the command gate for all calls into the driver that they know about. Generally, drivers should ensure that all I/O and hardware-specific operations are executed with the command gate closed.

IOAudioDevice also offers timer services to the audio driver. These services allow different objects within the driver to receive notifications that are guaranteed to be delivered at the requested timer interval, if not sooner. Different target objects can register for timer callbacks at a specific interval; however, IOAudioDevice makes the actual timer interval the smallest of those requested. The idea behind this design is that there is no harm in having timed events in an audio driver occur sooner than requested. By coalescing the callback intervals, the Audio family obviates the overhead of multiple timers in a single driver.

Your driver itself can have localized strings that are accessible by the Audio HAL. These strings can include such things as name, manufacturer, and input sources. Follow the Mac OS X localization procedure for these strings, putting them in a file named `Localizable.strings` in the locale-specific subdirectories of your bundle. The driver should have a property named `IOAudioDeviceLocalizedBundleKey`, which has a value of the path of the bundle or kernel extension holding the localized strings, relative to `/System/Library/Extensions`. The driver's IOAudioDevice object should set this property in its implementation of the `initHardware` method.

IOAudioEngine

An audio-engine object represents and manages an audio device's I/O engine. In an audio driver, the object is an instance of a custom subclass of IOAudioEngine. Its main job is to allow the I/O engine to transfer audio data (in the form of a stream of sample frames) between the device and the sample buffer at a specific sampling rate. (Its secondary task is to move the data between the sample buffer and the mix buffer after appropriately converting the data to the format expected, depending on direction, by client or hardware; you can find more information on this topic later in "The Audio I/O Model Up Close" (page 31).)

An instance of IOAudioStream (described in "IOAudioStream" (page 28)) represents and encapsulates a sample (and mix) buffer in a driver. Each IOAudioEngine in a driver must create one or more IOAudioStream objects for each sample buffer required by the I/O engine. A typical driver has at least an input IOAudioStream and an output IOAudioStream. An IOAudioEngine object may also create the audio-control objects (IOAudioControl) required by the device, although this task can be handled by the driver's IOAudioDevice. During the initialization phase, the driver must add all created IOAudioStreams and IOAudioControls to the IOAudioEngine as instance variables using the appropriate IOAudioEngine methods. This must happen before it activates the IOAudioEngine (`activateAudioEngine` method).

In addition to facilitating the transfer of audio data in and out of the sample and mix buffers, an IOAudioEngine has a number of functions:

- It must stop and start the I/O engine when requested.

- When the I/O engine is started, the IOAudioEngine object must ensure that it runs continuously and, at the end of the sample buffer, loops to the beginning of the buffer. As the engine loops, the IOAudioEngine takes a time stamp and increments a loop count.

- It must provide the current sample on demand.

- If the IOAudioEngine supports multiple stream formats or sampling rates, it must modify the hardware appropriately when a format or rate changes.

An IOAudioEngine has several attributes and structures associated with it. Most important of these is a status buffer (`IOAudioEngineStatus`) that it shares with the Audio HAL. This status buffer is a structure that the IOAudioEngine must update each time the I/O engine loops around to the start of the sample buffer. The structure contains four fields, three of which hold critical values:

- The number of times the audio engine has looped to the start of the sample buffer

- The time stamp of the most recent occurrence of this looping

- The current location of the erase head (in terms of sample frame)

It is important that these fields, and especially the time-stamp field, be as accurate as possible. The Core Audio framework (Audio HAL) uses this timing information to calculate the exact position of the audio engine at any time. The shared status buffer is thus the basis for the timer and synchronization mechanism used by the Mac OS X audio subsystem.

The erase head mentioned in the previous paragraph is another attribute of an IOAudioEngine object. The erase head is a software construct that zeroes out the mix and sample buffers just after the sample frames have been played in an output stream. It is always moving just behind the audio engine and just ahead of the IOAudioEngine clipping and conversion routines that convert the audio data in the mix buffer.

An IOAudioEngine object performs a number of initializations to fine-tune the synchronization mechanism described above. For example, it provides methods for setting the latency of the audio engine and for varying the offset between the Audio HAL and the audio engine's I/O head.

IOAudioStream

An IOAudioStream object represents and encapsulates an input or output sample buffer in an audio driver. It also encapsulates the driver's mix buffer, the possible formats of the audio stream (which include sample rate, bit depth, and the number of channels), the starting channel ID, and the number of current clients. It also contains (as instance variables) all IOAudioControls that affect the channels of the stream. An IOAudioStream is an instance variable of the IOAudioEngine object that creates it.

When an IOAudioStream is created, it must have added to it all the possible sample formats as well as all the supported sample rates for each format. The current format must be explicitly set. If a sample buffer has multiple channels, the channels are typically interleaved in the series of sample frames. However, you may use separate IOAudioStreams for different channels).

The IOAudioStream class defines the `AudioIOFunction` prototype for the member functions (typically implemented by the owning IOAudioEngine) that clip and convert output audio data from the float mix buffer to the sample buffer in the format required by the hardware. See "The Audio I/O Model Up Close" (page 31) for further information.

IOAudioStream includes convenience methods that permit IOAudioStream objects to be created from, and saved to, OSDictionary objects.

IOAudioControl

An IOAudioControl object represents a controllable attribute of an audio device, such as mute, volume, input/output selector, or master gain. It is usually associated with a specific channel in a specific stream, but can be used to control all channels in an IOAudioStream or even all channels in an IOAudioEngine. An IOAudioEngine object holds its IOAudioControl objects as instance variables; the engine's IOAudioStream objects may also hold, as instance variables, the audio-control objects that are specifically tied to the stream's channels.

Usually an instance of an IOAudioEngine subclass creates its IOAudioControls when it initializes the hardware (in the `initHardware` method). However, the driver's IOAudioDevice object may be the object that creates the necessary IOAudioControls. In either case, the driver must add the control objects to the appropriate IOAudioEngine using the `addDefaultAudioControl` method.

Thus an IOAudioControl object is associated with an IOAudioEngine object, an IOAudioStream object, and a channel of the stream. All of its attributes are stored in the I/O Registry. It is known as a "default control" because the Audio HAL recognizes and uses it based on its attributes, which it discovers through the I/O Registry.

When the IOAudioEngine (or IOAudioDevice) object creates IOAudioControl objects, it must obtain from the audio device the starting channel identifier (an integer) for the audio stream. When the driver creates the first IOAudioControl for the stream, it assigns this channel ID to it. When it creates IOAudioControls for any other channels of the stream (based on the number of channels the stream supports), it simply increments the ID of the channel associated with the control. The Audio family assigns `enum` identifiers to channel IDs in `IOAudioTypes.h`; these include identifiers for left, right, center, left-rear, and right-rear channels, as well as an identifier for all channels.

In addition to channel ID, the Audio family uses a multi-tier classification scheme (defined by enums in `IOAudioTypes.h`) to identify IOAudioControls:

- Audio types: output, input, mixer, pass-through, and processing

- Audio subtypes:

   ❏ For output: internal speaker, external speaker, headphones, line, and S/PDIF

   ❏ For input: internal microphone, external microphone, CD, line, and S/PDIF

- Control types: level and selector

- Control subtypes: volume, mute, input, output, clock services

- Usage type: input, output, and pass-through

When you create an IOAudioControl object, you specify control type, control subtype, usage type, and channel name (in addition to channel ID).

The level and selector control types correspond to subclasses of IOAudioControl, described in Table 3-1 (page 29).

Table 3-1   Subclasses of IOAudioControl

Subclass	Description
IOAudioLevelControl	Implements an audio control based on a minimum and maximum value. A control subtype specifically creates a volume control using minimum and maximum decibels associated with these levels.
IOAudioSelectorControl	Implements an audio control based on selection of discrete elements. Control subtypes include those for mute, input/output, and clock services.
IOAudioToggleControl	Implements an audio control based on binary values (off and on, start and stop, and so on) such as might pertain to a mute control.

Some object in an audio driver—typically the IOAudioDevice object because of its central role—must implement what is known as "value change handlers." A value change handler is a callback routine that conforms to one of three prototypes defined in `IOAudioControl.h` based on the type of value (integer, OSObject, or `void *` data). When invoked, a value change handler should write the change in value to the audio hardware.

Changes to control values that originate with clients of the Audio HAL—for example, a user moves a volume dial on a CD-playing application—initiate a long series of actions in the Audio HAL and the Audio family:

1. The Audio HAL goes through the I/O Registry to determine the property or properties associated with the value change.

2. Via the IOAudioEngineUserClient object, the IOAudioControl superclass's implementation of `setProperties` is invoked.

3. Using the dictionary of properties passed into `setProperties`, IOAudioControl locates the target control object and calls `setValueAction` on it.

4. The `setValueAction` method calls `setValue` on the driver's work loop while holding the driver's command gate.

5. The `setValue` method first calls `performValueChange`, which does two things:

    a. It calls the value change handler for the IOAudioControl (which must conform to the appropriate function prototype for the callback).

    b. It sends a notification of the change to all clients of the IOAudioControl (`sendValueChangeNotification`).

6. Finally, `setValue` calls `updateValue` to update the I/O Registry with the new value.

When a change is physically made to audio hardware—for example, a user turns a volume dial on an external speaker—what must be done is much abbreviated. When the driver detects a control-value change in hardware, it simply calls `hardwareValueChanged` on the driver's work loop. This method updates the value in the IOAudioControl and in the I/O Registry, and then sends a notification to all interested clients.

User Client Classes

The Audio family provides two user-client classes, IOAudioEngineUserClient and IOAudioControlUserClient. The Audio family automatically instantiates objects of each class for each IOAudioEngine object and each IOAudioControl in a driver. These objects enable the communication of audio data and notifications between the driver and the Audio HAL. You should not have to do anything explicitly in your code to have the default user-client objects created for, and used by, your driver.

For further details, see "User Client Objects" (page 35).

IOAudioPort

The IOAudioPort class instantiates objects that represent a logical or physical port, or a functional unit in an audio device. An IOAudioPort object represents an element in the signal chain in the audio device and may contain one or more IOAudioControl objects through which different attributes of the port can be represented and adjusted.

The IOAudioPort class is deprecated and may eventually be made obsolete. The class is currently public to maintain compatibility. Driver writers are encouraged not to use IOAudioPort objects in their code.

The Audio I/O Model Up Close

In the previous chapter, the section "The Audio I/O Model on Mac OS X" (page 17) described the Mac OS X audio I/O model from the perspective of how that model compares to the Mac OS 9 model. Because it was a comparative overview, that description left out some important details. The following discussion supplies those details, with the intent that a fuller understand of the audio I/O model is of particular benefit to audio driver writers.

Ring Buffers and Time Stamps

The distinctive architecture of the Mac OS X audio I/O model starts with how the hardware's DMA engine is programmed. Recall that in Mac OS 9 the engine is programmed for double buffering where the sizes of the sample buffers are set to the client's (typically small) buffer size; the engine is also programmed to fire an interrupt whenever it has filled a buffer (for input) or it is ready to play a buffer (output). The interrupt signals the client that it has data for it, or is ready to accept more data. Needless to say, the heavy use of hardware interrupts in this model is detrimental to the efficiency of the audio system.

In Mac OS X, the Audio family programs the DMA engine for a single ring buffer, typically a buffer of large size. In a ring buffer, the DMA engine "wraps" to the start of the buffer when it finishes writing to (or reading from) the end of the buffer. Thus the DMA engine is set to continuously loop through the sample buffer, reading or writing audio data, depending on direction. As it wraps, the engine is programmed to fire an interrupt. The driver (in an IOAudioEngine object) handles the interrupt by calling `takeTimeStamp` in the driver's work loop.

In calling `takeTimeStamp`, the driver writes two critical pieces of data to an area of memory shared between the IOAudioEngine and its Audio HAL clients. The first is an extremely accurate time stamp (based on `utime`), and the other is an incremented loop count. The structure defining these (and other) fields is `IOAudioEngineStatus`. The IOAudioEngine's user-client object maps the memory holding the `IOAudioEngineStatus` information into the address spaces of the Audio HAL clients.

The Audio HAL Predicts

The Audio HAL uses the accumulated time stamps and loop counts in a sophisticated calculation that predicts when the I/O engine will be at any location in the sample buffer; as a result, it can also predict when each client of a particular I/O engine should be ready to provide audio data to the hardware or accept audio data from it. This calculation takes account not only of the current sample-frame position of the I/O engine but the buffer sizes of the clients, which can be variable.

Each client of the Audio HAL has its own I/O thread. The Audio HAL puts this thread to sleep until the time comes for the client to read or write audio data. Then the Audio HAL wakes the client thread. This is a kind of software-simulated interrupt, which involves much less overhead than a hardware interrupt.

Interpolation

Before going further, it is worthwhile to consider some of the theory behind this design. The Mac OS X audio system makes the assumption that a hardware I/O engine, as it processes audio data in the sample buffer, is proceeding continuously at a *more or less* constant rate. The "more or less" qualification is important here because, in reality, there will be slight variations in this rate for various reasons, such as imperfections in clock sources. So the mechanism by which the Audio HAL continually uses time stamps to calculate and predict a wake-up time for each of its client I/O threads can be considered an interpolation engine. It is a highly accurate predictive mechanism that "smooths out" these slight variations in engine rate, building in some leeway so that there is no discernible effect on audio quality.

Client Buffers and I/O Procedures

As noted earlier, each client of the Audio HAL can define the size of its audio buffer. There are no restrictions, except that the buffer can be no larger than the size of the hardware sample buffer. For performance reasons, almost all clients prefer buffers sizes that are considerably smaller. Buffer sizes are typically a power of two. The Audio HAL takes the buffer sizes of its clients into account when it calculates the next I/O cycle for those clients.

Each client of the Audio HAL must also implement a callback function conforming to the type `AudioDeviceIOProc`. When the Audio HAL wakes a sleeping client I/O thread, it calls this function, passing in the buffers (input and output) whose sizes were specified by the client. It is in this implementation of the `AudioDeviceIOProc` routine that the client gives audio data to the hardware or receives it from the hardware.

The following section, "A Walk Through the I/O Model" (page 33), discusses what happens next in detail.

Figure 3-4   Multiple Audio HAL client buffers and the mix buffer (output)
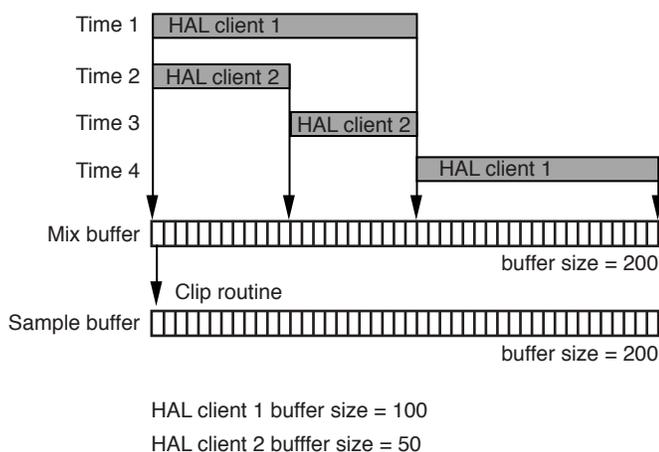
A Walk Through the I/O Model

With the essential timing mechanism used for audio I/O in mind, we can now follow a cycle of that I/O through the audio system to see what exactly happens. What happens is different between input and output audio streams. An output stream involves a more complicated path because each client is contributing, frame by frame, to the final sound played by speakers or recorded by some device.

Output Streams

Let's begin with an output stream. Figure 3-4 (page 32) illustrates the relationship between the buffers of Audio HAL clients and the buffers of the audio driver during an output cycle. Refer to this diagram during the following discussion.

For each of its clients, the Audio HAL calculates intervals that are based on the accumulated time stamps and loop counts associated with an I/O engine as well as client buffer sizes. The Audio HAL sleeps the I/O threads of its clients for these intervals, waking each thread when it's time for the client to give the hardware its data. In waking the thread, it calls the `AudioDeviceIOProc` routine implemented by the client, passing in a number of buffers and time stamps:

■ a list of input buffers along with a time stamp that indicates when the data was recorded

■ a list of output buffers along with a time stamp that indicates when the data will be played

■ a time stamp to be used for "now" rather than the device clock

The input and output time stamps allow the client to make various calculations, such as how much time they have before the data is played. The inclusion of both input and output parameters enables clients that are both producers and consumers of audio data (for example, a recording unit with playback capabilities) to process both streams at the same time. In this case, the client first takes the data in the list of input buffers before filling the output buffers with 32-bit floating point samples.

When the client returns in its `AudioDeviceIOProc` routine, the Audio HAL puts the I/O thread to sleep until the next time data is required from the client. The Audio HAL gives the samples in the output buffer to the associated IOAudioEngineUserClient object, which calls the appropriate IOAudioStream object to have the samples moved from the client buffer to the appropriate frames in the IOAudioEngine's mix buffer. Other clients can also deposit data in the same locations in the mix buffer. If another client already has deposited data in those frames, the new client's floating-point values are simply added to the existing values.

Clients can contribute output data to a frame almost until the I/O engine is ready for that data. The IOAudioEngine object containing the mix buffer knows how many clients it has and when each has contributed its share of data to any one frame of the mix buffer (for the current loop through it). In addition, the driver (through the Audio family) maintains a "watchdog" timer that tracks the current location of each client relative to the I/O engine. If a client has not provided audio data by the time the I/O engine needs to accesses it, the watchdog timer fires and clips all of the currently mixed samples into the sample buffer. Because some time is needed to perform this clip operation, the watchdog actually fires a short amount of time before the data is needed. It is possible that a "late" client could attempt to put data in the location of the mix buffer after the watchdog has fired, but before the I/O engine has processed the data. To accommodate this situation, the driver backs up and remixes and clip the data in an attempt to get it ready the I/O engine in time with the "late" samples.

Next, the driver's clip routine, `clipOutputSamples`, is invoked. In its implementation of this method, the driver must clip any excess floating-point values under -1.0 and over 1.0 —which can happen when multiple clients are adding their values to existing values in the same frame—and then convert these values to whatever format is required by the hardware. When `clipOutputSamples` returns, the converted values are written to the corresponding locations in the sample buffer. The DMA engine grabs the frames as it progresses through the sample buffer and the hardware plays them as sound.

Figure 3-5   Interplay of the I/O engine, erase heads, and clip routine (output)
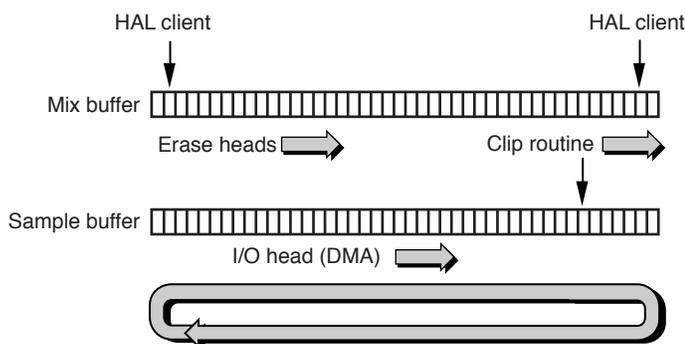
Erase Heads and Timer Services

The Audio family includes a further refinement to the synchronized actions described in the preceding paragraphs. Between the I/O engine and the clipping and converting done by the driver, it runs parallel "erase heads" in both the mix and sample buffers. These erase heads simply zero-fill the corresponding frames at the same time. This precaution reduces the possibility that any frame could become polluted with left-over bits.

The erase heads are run in a separate thread and have their own timer. They are programmed to run four times per sample-buffer cycle. They do not erase the entire range of frames between the current locations of the DMA engine and the driver's clip routine, allowing a little space for the remixing of data from tardy clients.

The erase-heads timer is run using IOAudioDevice's timer services. Its interval is, of course, closely tied to the rate of the I/O engine and the time stamps taken by the IOAudioEngine.

Input Streams

With input audio streams, the picture is much simpler. There is no mix buffer and there are no erase heads. The Audio HAL clients are consumers of the data in this case, and not producers of it, so there is no need for these things.

Neither is there any need for a clip routine. The driver has to convert the integer data coming from the hardware to the 32-bit floating point required by the Audio HAL. But in the input direction, the driver is in a position to fit the converted data within the -1.0 to 1.0 floating-point maximum range.

So the simplified sequence is this: Shortly after the I/O engine writes the input data into the sample buffer, the driver—in its implementation of the IOAudioEngine method `convertInputSamples`—converts that data to 32-bit floating point. Then the data is given, via the IOAudioEngineUserClient interface, to each Audio HAL client in that client's `AudioDeviceIOProc` callback routine.

Interfaces With the Audio HAL

Audio drivers communicate with the Audio HAL and its clients using two mechanisms. The principle mechanism uses user-client objects to pass audio data, control value changes, and notifications across the kernel-user space boundary. The other mechanism allows driver writers to export custom device properties to Audio HAL clients.

User Client Objects

As described earlier in "User Client Classes" (page 30), the Audio family automatically configures and creates the appropriate number of user-client objects for a driver. These objects enable the communication of audio data and notifications between the driver and the Audio HAL. The objects typically are instances of one of two classes: IOAudioEngineUserClient and IOAudioControlUserClient. One IOAudioEngineUserClient object is created for each IOAudioEngine in a driver and one IOAudioControlUserClient object is created for each IOAudioControl.

An IOAudioEngineUserClient acts as the conduit through which audio data is passed between the audio driver and the Audio HAL. It is also the agent that maps the buffer maintained by an IOAudioEngine into memory shared with the associated Audio HAL device. (As you may recall, this buffer contains the time stamp and count of the most recent "wrap" of the sample buffer by the I/O engine.) Finally, the IOAudioEngineUserClient responds to requests by Audio HAL clients to get and set the properties of the hardware. An IOAudioControlUserClient has a more limited role compared to a IOAudioEngineUserClient. It merely sends a notification to interested clients of the Audio HAL when the value of a control (represented by IOAudioControl) changes.

Figure 3-6 The Audio family's user clients
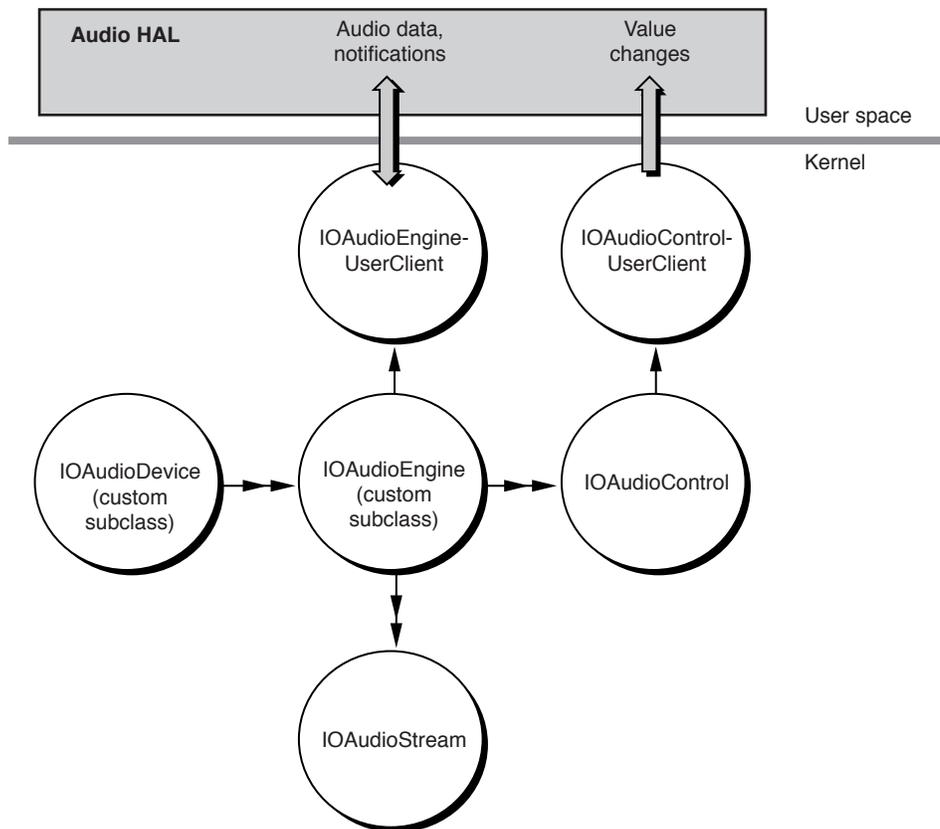
You do not have to do anything explicitly in your code to have the Audio family create the default user-client objects for your driver, nor do you have write any code to enable your driver to use these objects. It all happens automatically for your driver. But it could happen that you want custom behavior from your user clients; for example, you might want the user client to perform hardware mixing, writing the combined samples into a new buffer used by a single client. When you want custom user-client behavior, you can subclass the IOAudioEngineUserClient or IOAudioControlUserClient classes. That's why Apple makes the programmatic interface for these classes public in `IOAppleEngineUserClient.h` and `IOAppleControlUserClient.h`.

Custom Core Audio Properties

Sometimes you might have an audio device with properties that are not covered by what the Audio HAL specifies (in the Core Audio framework's `AudioHardware.h`.). For these situations, you can create a bundle that contains code implementing these device-specific properties on behalf of the Audio HAL. Then you can put this bundle in a location where it can be exported to user space. The bundle must be accessible by Core Foundation Bundle Services APIs (CFBundle).

To give the Audio HAL access to your device-property code, the bundle must also present the programmatic interface defined in the Core Audio header file `AudioDriverPlugIn.h`. These routines allow the Audio HAL's clients to open and close the device and to get and set the device

properties. When the driver changes a property, it calls one of two notification callbacks implemented by clients (one defined for Audio HAL device properties and the other for stream properties) to notify them of the change.

When you have created a bundle conforming to the interface in `AudioDriverPlugIn.h`, you usually install it inside your driver's kernel extension in `/System/Library/Extensions`. Because it is a bundle, it can also contain localizations of strings relevant to the new properties. The Audio HAL finds the bundle by looking in the registry for two keys: `kIOAudioEngineCoreAudioPlugInKey` and `kIOAudioDeviceLocalizedBundleKey`.

Implementing an Audio Driver

As discussed in the chapter "Audio Family Design" (page 21), writing an audio driver using the Audio families requires, in object-oriented terms, that you do some certain things in your code:

- Create a subclass of IOAudioDevice which, among other things, initializes the hardware.

- Create a subclass of IOAudioEngine which, among other things, initializes the I/O engine and stops and starts it.

- Create, configure, and attach to the IOAudioEngine object the number of IOAudioStream and IOAudioControl objects appropriate to your driver.

- Respond to value changes in the IOAudioControl objects.

- In a separate code module (but as part of the IOAudioEngine subclass implementation), implement the driver's clipping and converting routines

This chapter will guide you through these implementation steps. It uses as a code source the SamplePCIAudioDriver example project (located in `/Developer/Examples/Kernel/IOKit/Audio/Templates` when you install the Developer package). In the interest of brevity, this chapter does not use all the code found in that project and strips the comments from the code. Refer to the SamplePCIAudioDriver project for the full range of code and comments on it.

Setting Up the Project

Even before you create a project for your audio driver, you should consider some elemental facets of design. Examine the audio hardware and decide which Audio-family objects are required to support it. Of course, your driver must have one IOAudioDevice object (instantiated from a custom subclass); but how many IOAudioEngine, IOAudioStream, and IOAudioControl objects should you create?

Table 4-1 (page 40) provides a kind of decision matrix for determining how many Audio-family objects of each kind that you need.

Table 4-1   Deciding which Audio family objects to create (and other design decisions)

Question	What to create
Are there sample buffers of different sizes?	Create a custom IOAudioEngine object for each sample buffer.
How many I/O or DMA engines are there on the device	Create a custom IOAudioEngine object for each I/O or DMA engine.
How many separate or interleaved sample buffers are there?	Create an IOAudioStream object for each buffer (both input and output).
How many controllable attributes are there (volume, gain, mute, and so on)?	Create an IOAudioControl object for each attribute.

The SamplePCIAudioDriver project requires one custom IOAudioEngine, two IOAudioStreams (input and output), and six IOAudioControls (left and right output volume, left and right input gain, and input and output mute).

You also should decide what properties your driver must have to match against your provider's nub and specify those properties in your driver's IOKitPersonalities dictionary. In the SamplePCIAudioDriver personality (see Figure 4-1 (page 40)), the provider is the PCI family and the nub class is IOPCIDevice. In addition, a PCI audio driver would usually specify the vendor and device ID registers (primary or subsystem) as the value of the `IOPCIMatch` key.(Note that in the SamplePCIAudioDriver example, the vendor and device ID registers are specified as zeros; for your driver, you would substitute the appropriate values.) Finally, for your `IOClass` property, append the name of your IOAudioDevice subclass to the standard reverse-DNS construction `com_company_driver_`; in the case of the SamplePCIAudioDriver project, the `IOClass` value is "com_MyCompany_driver_SamplePCIAudioDevice".

Figure 4-1   Bundle settings of the sample PCI audio driver

Note: For more on PCI device matching, see the document Inside Mac OS X: PCI Cookbook.

Of course, if your driver's provider is different (say, USB or FireWire), the matching properties that you would specify in an IOKitPersonalities dictionary would be different.

As Figure 4-1 (page 40) suggests, also make sure that you specify other necessary properties in your driver's information property list, including the versioning and dependency information in the OSBundleLibraries dictionary.

Implementing an IOAudioDevice Subclass

Every I/O Kit audio driver must implement a subclass of IOAudioDevice from which one instance of the class is created when the driver is loaded. A IOAudioDevice object is the central, coordinating object of the driver; it represents the audio hardware in an overall sense.

Despite its central role, an IOAudioDevice subclass generally does not do as much as an IOAudioEngine subclass. It merely initializes the hardware at start-up and creates the custom IOAudioEngine objects required by the driver. It may also create the IOAudioControls used by the driver and respond to requests to change the values of these controls, but the IOAudioEngine subclass could do these tasks instead. In the example used for this chapter (SamplePCIAudioDriver), the IOAudioDevice subclass creates and manages the device's controls.

Begin by adding in Xcode a header file and an implementation file for the IOAudioDevice superclass you are going to implement. In the header file, specify IOAudioDevice as the superclass and provide the necessary declarations. Listing 4-1 (page 41) shows the beginning of SamplePCIAudioDevice.h.

Listing 4-1   Partial class declaration of the IOAudioDevice subclass

```
#include <IOKit/audio/IOAudioDevice.h>

typedef struct SamplePCIAudioDeviceRegisters {
    UInt32 reg1;
    UInt32 reg2;
    UInt32 reg3;
    UInt32 reg4;
} SamplePCIAudioDeviceRegisters;

class IOPCIDevice;
class IOMemoryMap;

#define SamplePCIAudioDevice com_MyCompany_driver_SamplePCIAudioDevice

class SamplePCIAudioDevice : public IOAudioDevice
{
    friend class SampleAudioEngine;

    OSDeclareDefaultStructors(SamplePCIAudioDevice)

    IOPCIDevice     *pciDevice;
    IOMemoryMap     *deviceMap;

    SamplePCIAudioDeviceRegisters *deviceRegisters;
```

```
// ...
```

Hardware Initialization

I/O Kit audio drivers do not need to override the IOService `start` method. Instead, IOAudioDevice's default implementation of `start` first invokes the superclass implementation and then calls the `initHardware` method of the subclass. Your IOAudioDevice subclass must override the `initHardware` method.

Your implementation of `initHardware` must do two general things:

- It must perform any necessary hardware-specific initializations (on both the provider and the audio sides), such as mapping resources and setting the hardware to a known state. It also involves creating and initializing the necessary Audio family objects.

- It must set the names by which the driver is to be known to the Audio HAL and its clients.

If the `initHardware` call succeeds, the IOAudioDevice superclass (in the `start` method) sets up power management if the family is supposed to manage power and then calls `registerService` to make the IOAudioDevice object visible in the IORegistry.

Listing 4-2 (page 42) shows how the SamplePCIAudioDevice class implements the `initHardware` method.

Listing 4-2   Implementing the initHardware method

```
bool SamplePCIAudioDevice::initHardware(IOService *provider)
{
    bool result = false;

    IOLog("SamplePCIAudioDevice[%p]::initHardware(%p)\n", this, provider);

    if (!super::initHardware(provider)) {
        goto Done;
    }

    pciDevice = OSDynamicCast(IOPCIDevice, provider);
    if (!pciDevice) {
        goto Done;
    }

    deviceMap =
pciDevice->mapDeviceMemoryWithRegister(kIOPCIConfigBaseAddress0);
    if (!deviceMap) {
        goto Done;
    }

    deviceRegisters = (SamplePCIAudioDeviceRegisters
*)deviceMap->getVirtualAddress();
    if (!deviceRegisters) {
        goto Done;
    }
```

```
    pciDevice->setMemoryEnable(true);

    setDeviceName("Sample PCI Audio Device");
    setDeviceShortName("PCIAudio");
    setManufacturerName("My Company");

#error Put your own hardware initialization code here...and in other
routines!!

    if (!createAudioEngine()) {
        goto Done;
    }

    result = true;

Done:

    if (!result) {
        if (deviceMap) {
            deviceMap->release();
            deviceMap = NULL;
        }
    }

    return result;
}
```

The first part of this method does some provider-specific initializations. The implementation gets the provider, a PCIDevice object, and with it configures a map for the PCI config base registers, and with this map gets the virtual address for the registers. Then, by calling `setMemoryEnable`, it enables PCI memory access.

Next, the SamplePCIAudioDevice implementation sets the full and short name of the device as well as the manufacturer's name, making this information available to the Audio HAL.

The last significant call in this implementation, to `createAudioEngine`, invokes a method that creates the driver's IOAudioEngine and IOAudioControl objects (and indirectly, the driver's IOAudioStream objects).

Creating the IOAudioEngine Objects

In the `initHardware` method, create an instance of your driver's IOAudioEngine subclass for each I/O engine on the device. Once it's created and initialized, call `activateAudioEngine` to signal to the system that the engine is ready to begin vending audio services.

The SamplePCIAudioDevice subclass creates its sole IOAudioEngine object in a subroutine of `initHardware` named `createAudioEngine` (see).

Listing 4-3   Creating an IOAudioEngine object

```
bool SamplePCIAudioDevice::createAudioEngine()
{
    bool result = false;
    SamplePCIAudioEngine *audioEngine = NULL;
```

```
    IOAudioControl *control;

    audioEngine = new SamplePCIAudioEngine;
    if (!audioEngine) {
        goto Done;
    }
    if (!audioEngine->init(deviceRegisters)) {
        goto Done;
    }
     // example code skipped...
     // Here create the driver's IOAudioControl objects
     // (see next section)...

    activateAudioEngine(audioEngine);

    audioEngine->release();
    result = true;

Done:
    if (!result && (audioEngine != NULL)) {
        audioEngine->release();
    }
    return result;
}
```

> **Note:** In the interest of concision, this listing excludes the code that creates the driver's
> IOAudioControl objects; see "Creating and Adding the IOAudioControl Objects" (page 44) for
> this step.

In this example, the IOAudioDevice subclass creates a raw instance of the driver's subclass of
IOAudioEngine (SamplePCIAudioEngine) and then initializes it, passing in the device registers
so the object can access those registers. You can define your init method to take any number of
parameters.

Next, the IOAudioDevice implementation activates the audio engine (activateAudioEngine);
this causes the newly created IOAudioEngine object's start and initHardware methods to be
invoked. When activateAudioEngine returns, the IOAudioEngine is ready to begin vending
audio services to the system. Because the IOAudioDevice superclass retains the driver's
IOAudioEngine objects, be sure to release each IOAudioEngine so that it is freed when the driver
is terminated.

Creating and Adding the IOAudioControl Objects

A typical I/O Kit audio driver must instantiate several IOAudioControl objects to help it manage
the controllable attributes of the audio hardware. These attributes include such things as volume,
mute, and input/output selection. You can create and manage these control objects in your
IOAudioEngine subclass or in your IOAudioDevice subclass; it doesn't matter which.

As summarized in Table 4-2 (page 45), the Audio family provides three subclasses of
IOAudioControl that implement behavior specific to three functional types of control. Instantiate
a control from the subclass that is appropriate to a controllable attribute of the device.

Table 4-2   Subclasses of IOAudioControl

Subclass	Purpose
IOAudioLevelConrol	For controls such as volume, where a range of measurable values (such as decibels) is associated with an integer range.
IOAudioToggleControl	For controls such as mute, where the state is either off or on.
IOAudioSelectorControl	For controls that select a discrete attribute, such as input gain.

Each subclass (or control type) has a `create` method and a convenience method specific to a subtype of control. The `IOAudioTypes.h` header file, which defines constants for control type and subtype, also defines other constants intended to be supplied as parameters in the control-creation methods. summaries the categories that these constants fall into.

Table 4-3   Categories of audio-control constants in IOAudioTypes.h

Category	Purpose	Examples and Comments
Type	General function of control	Level, toggle, or selector (each corresponding to an IOAudioControl subclass)
Subtype	Purpose of the control	Volume, mute, or input/output; subclass convenience methods assume a subtype.
Channel ID	Common defaults for channels	Default right channel, default center channel, default sub woofer, all channels
Usage	How the control is to be used	Output, input, or pass-through

See `IOAudioTypes.h` for the complete set of audio-control constants.

After you create an IOAudioControl object you must take two further steps:

■ Set the value-change handler for the control.

The value-change handler is a callback routine that is invoked when a client of the Audio HAL requests a change in a controllable attribute. See for more on these routines.

■ Add the IOAudioControl to the IOAudioEngine object they are associated with.

In our SamplePCIAudioDriver example, the IOAudioDevice subclass creates and initializes the driver's IOAudioControl objects. This happens in the `createAudioEngine` method; show the creation and initialization of one control.

Listing 4-4   Creating an IOAudioControl object and adding it to the IOAudioEngine

```
    // ... from createAudioEngine()
    control = IOAudioLevelControl::createVolumeControl(
         65535,     // initial value
         0,      // min value
         65535,      // max value
         (-22 << 16) + (32768),      // -22.5 in IOFixed (16.16)
         0,      // max 0.0 in IOFixed
         kIOAudioControlChannelIDDefaultLeft,
         kIOAudioControlChannelNameLeft,
         0,      // control ID - driver-defined
         kIOAudioControlUsageOutput);
    if (!control) {
        goto Done;
    }
    control->setValueChangeHandler((IOAudioControl::IntValueChangeHandler)
                                   volumeChangeHandler, this );
    audioEngine->addDefaultAudioControl(control);
    control->release();

/* Here create more IOAudioControl objects for right output channel,
** output mute,left and right input gain, and input mute. For each, set
** value change handler and add to the IOAudioEngine
*/
// ...
```

Note: This code fragment shows the creation of one IOAudioControl only. What must be done for the other controls—right output volume, output mute, left and right input gain, and input mute—is similar.

In this example, the IOAudioDevice subclass creates a left output volume control with an integer range from 0 to 65535 and a corresponding decibel range from -22.5 to 0.0. A channel must always be associated with an IOAudioControl object. You do this when you create the object by specifying constants (defined in `IOAudioDefines.h`) for both channel ID and channel name. You must also specify a "usage" constant that indicates how the IOAudioControl will be used (input, output, or pass-through).

Once you have added an IOAudioControl to its IOAudioEngine, you should release it so that it is properly freed when the IOAudioEngine object is done with it.

Implementing Control Value-Change Handlers

For each IOAudioControl object that your driver creates, it must implement what is known as a value-change handler for it. (This doesn't imply that you need you need to create a separate handler for each control; one handler could be used to manage multiple controls of similar type.) The value-change handler is a callback routine that is invoked when the controllable device attribute associated with an IOAudioControl object needs to be changed.

The header file `IOAudioControl.h` defines three prototypes for control value-change handlers:

```
    typedef IOReturn (*IntValueChangeHandler)(OSObject *target,
        IOAudioControl *audioControl, SInt32 oldValue, SInt32 newValue);
```

```
typedef IOReturn (*DataValueChangeHandler)(OSObject *target,
    IOAudioControl *audioControl, const void *oldData, UInt32
    oldDataSize, const void *newData, UInt32 newDataSize);
typedef IOReturn (*ObjectValueChangeHandler)(OSObject *target,
    IOAudioControl *audioControl, OSObject *oldValue,
    OSObject *newValue);
```

Each prototype is intended for a different kind of control value: integer, pointer to raw data (`void *`), and (libkern) object. For most cases, the integer handler should be sufficient. All of the existing IOAudioControl subclasses pass integer values to the `IntValueChangeHandler`.

The essential task of the value-change handler is to update the proper attribute of the audio hardware to the new control value. shows how one might implement a value-change handler (excluding the actual attribute-setting code).

Listing 4-5   Implementing a control value-change handler

```
IOReturn SamplePCIAudioDevice::volumeChangeHandler(IOService *target,
        IOAudioControl *volumeControl, SInt32 oldValue, SInt32 newValue)
{
    IOReturn result = kIOReturnBadArgument;
    SamplePCIAudioDevice *audioDevice;

    audioDevice = (SamplePCIAudioDevice *)target;
    if (audioDevice) {
        result = audioDevice->volumeChanged(volumeControl, oldValue,
                    newValue);
    }

    return result;
}

IOReturn SamplePCIAudioDevice::volumeChanged(IOAudioControl *volumeControl,
                            SInt32 oldValue, SInt32 newValue)
{
    IOLog("SamplePCIAudioDevice[%p]::volumeChanged(%p, %ld, %ld)\n", this,
            volumeControl, oldValue, newValue);

    if (volumeControl) {
        IOLog("\t-> Channel %ld\n", volumeControl->getChannelID());
    }

    // Add hardware volume code change

    return kIOReturnSuccess;
}
```

The reason for the nested implementation in this example is that the value-change callback itself must be a straight C-language function (in this case it's a static member function). The static function simply forwards the message to the actual target for processing.

Implementing an IOAudioEngine Subclass

In addition to the implementing a subclass of IOAudioDevice, writers of audio drivers must also implement a subclass of IOAudioEngine. This subclass should define the attributes and behavior of the driver that are specific to the hardware's I/O engine. These include specifying the size and characteristics of the sample and mix buffers, getting the current sample frame on demand, handling interrupts to take a time stamp, handling format changes, and starting and stopping the I/O engine upon request.

Start by defining the interface of your IOAudioEngine subclass in a header file. shows the main contents of the `SamplePCIAudioEngine.h` file.

Listing 4-6   Interface definition of the SamplePCIAudioEngine class

```
class SamplePCIAudioEngine : public IOAudioEngine
{
    OSDeclareDefaultStructors(SamplePCIAudioEngine)

    SamplePCIAudioDeviceRegisters   *deviceRegisters;

    SInt16                          *outputBuffer;
    SInt16                          *inputBuffer;

    IOFilterInterruptEventSource    *interruptEventSource;

public:

    virtual bool init(SamplePCIAudioDeviceRegisters *regs);
    virtual void free();

    virtual bool initHardware(IOService *provider);
    virtual void stop(IOService *provider);

    virtual IOAudioStream *createNewAudioStream(IOAudioStreamDirection
            direction, void *sampleBuffer, UInt32 sampleBufferSize);

    virtual IOReturn performAudioEngineStart();
    virtual IOReturn performAudioEngineStop();

    virtual UInt32 getCurrentSampleFrame();

    virtual IOReturn performFormatChange(IOAudioStream *audioStream,
            const IOAudioStreamFormat *newFormat, const IOAudioSampleRate
            *newSampleRate);

    virtual IOReturn clipOutputSamples(const void *mixBuf, void *sampleBuf,

            UInt32 firstSampleFrame, UInt32 numSampleFrames, const
        IOAudioStreamFormat *streamFormat, IOAudioStream *audioStream);
    virtual IOReturn convertInputSamples(const void *sampleBuf, void
*destBuf,
            UInt32 firstSampleFrame, UInt32 numSampleFrames, const
        IOAudioStreamFormat *streamFormat, IOAudioStream *audioStream);
```

```
    static void interruptHandler(OSObject *owner, IOInterruptEventSource

            *source, int count);
    static bool interruptFilter(OSObject *owner,
IOFilterInterruptEventSource
            *source);
    virtual void filterInterrupt(int index);
};
```

Most of the functions and types declared here will be explained in the following sections—including (for example) why there is a cluster of interrupt-related functions. You should be aware, however, that the `clipOutputSamples` and `convertInputSamples` methods, although they are declared in the class interface-definition file (in this case, `SamplePCIAudioEngine.h`), are implemented in a separate source module. The reason for that, and the procedures for implementing these functions, are found in "Clipping and Converting Samples" (page 56).

Hardware Initialization

As you did in your IOAudioDevice subclass, you must implement the `initHardware` method in your IOAudioEngine subclass to perform certain hardware initializations. The IOAudioEngine `initHardware` method is invoked indirectly when the IOAudioDevice object calls `activateAudioEngine` on an IOAudioEngine object.

In your implementation of `initHardware`, you should accomplish two general tasks: configure the I/O engine and create the IOAudioStream objects used by the engine. As part of initialization you should also implement the `init` method if anything special should happen prior to the invocation of `initHardware`; in the case of the SamplePCIAudioEngine class, the `init` method calls the superclass implementation and then assigns the passed-in device registers to an instance variable.

Configuring the I/O Engine

Configuring the audio hardware's I/O engine involves the completion of many recommended tasks:

- Determine the current sample rate and set the initial sample rate using `setSampleRate`.

- Call `setNumSampleFramesPerBuffer` to specify the number of sample frames in each buffer serviced by this I/O engine.

- Call `setDescription` to make the name of the I/O engine available to Audio HAL clients.

- Call `setOutputSampleLatency` or `setInputSampleLatency` (or both methods, if appropriate) to indicate how much latency exists on the input and output streams. The Audio family makes this information available to the Audio Hal so it can pass it along to its clients for synchronization purposes.

- Call `setSampleOffset` to make sure that the Audio HAL stays at least the specified number of samples away from the I/O engine's head. This setting is useful for block-transfer devices.

- Create the IOAudioStream objects to by used by the I/O engine and add them to the IOAudioEngine. See "Creating IOAudioStream Objects" (page 51) for details.

■ Set up a command-gate routine to handle the interrupt fired by the I/O engine when it wraps to the beginning of the sample buffer. (This assumes a "traditional" interrupt.)

■ Perform any necessary engine-specific initializations.

Listing 4-7 (page 50) illustrates how the SamplePCIAudioEngine class does some of these steps. Note that some initial values, such as INITIAL_SAMPLE_RATE, have been defined earlier using #define preprocessor commands.

Listing 4-7   Configuring the I/O engine

```
bool SamplePCIAudioEngine::initHardware(IOService *provider)
{
    bool result = false;
    IOAudioSampleRate initialSampleRate;
    IOAudioStream *audioStream;
    IOWorkLoop *workLoop;

    if (!super::initHardware(provider)) {
        goto Done;
    }
    initialSampleRate.whole = INITIAL_SAMPLE_RATE;
    initialSampleRate.fraction = 0;
    setSampleRate(&initialSampleRate);
    setDescription("Sample PCI Audio Engine");
    setNumSampleFramesPerBuffer(NUM_SAMPLE_FRAMES);

    workLoop = getWorkLoop();
    if (!workLoop) {
        goto Done;
    }

    interruptEventSource =
IOFilterInterruptEventSource::filterInterruptEventSource(this,
                        SamplePCIAudioEngine::interruptHandler,
                        SamplePCIAudioEngine::interruptFilter,
                        audioDevice->getProvider());
    if (!interruptEventSource) {
        goto Done;
    }
    workLoop->addEventSource(interruptEventSource);

    outputBuffer = (SInt16 *)IOMalloc(BUFFER_SIZE);
    if (!outputBuffer) {
        goto Done;
    }
    inputBuffer = (SInt16 *)IOMalloc(BUFFER_SIZE);
    if (!inputBuffer) {
        goto Done;
    }

    audioStream = createNewAudioStream(kIOAudioStreamDirectionOutput,
                outputBuffer, BUFFER_SIZE);
    if (!audioStream) {
        goto Done;
    }
```

```
    addAudioStream(audioStream);
    audioStream->release();

    audioStream = createNewAudioStream(kIOAudioStreamDirectionInput,
                inputBuffer, BUFFER_SIZE);
    if (!audioStream) {
        goto Done;
    }
    addAudioStream(audioStream);
    audioStream->release();
    result = true;
Done:
    return result;
}
```

The following section, "Creating IOAudioStream Objects" (page 51), describes the implementation of `createAudioStreams`, which this method calls. A couple of other things in this function merit a bit more discussion.

In the middle of the method are a few lines of code that create a filter interrupt event source and add it to the work loop. Through this event source, an event handler specified by the driver will receive interrupts fired by the I/O engine. In the case of SamplePCIAudioEngine, the driver wants the interrupt at primary interrupt time instead of secondary interrupt time because of the better periodic accuracy. To do this, it creates an IOFilterInterruptEventSource that makes a filtering call to the primary interrupt handler (`interruptFilter`); the usual purpose of this callback is to determine which secondary interrupt handler should be called, if any. The SamplePCIAudioEngine in the `interruptFilter` routine (as you'll see in "Taking a Time Stamp" (page 54)) calls the method that actually takes the time stamp and always returns `false` to indicate that the secondary handler should not be called. For the driver to receive interrupts, the event source must be enabled. This is typically done when the I/O engine is started.

Second, this function allocates input and output sample buffers in preparation for the creation of IOAudioStream objects in the two calls to `createNewAudioStreams`. The method of allocation in this example is rather rudimentary and would be more robust in a real driver. Also note that `BUFFER_SIZE` is defined earlier as

```
NUM_SAMPLE_FRAMES * NUM_CHANNELS * BIT_DEPTH / 8
```

In other words, compute the byte size of your sample buffers by multiplying the number of sample frames in the buffer by the number of the channels in the audio stream; then multiply that amount by the bit depth and divide the resulting amount by 8 (bit size of byte).

Creating IOAudioStream Objects

Your IOAudioEngine subclass should also create its IOAudioStream objects when it initializes the I/O engine (`initHardware`). You should have one IOAudioStream instance for each sample buffer serviced by the I/O engine. In the process of creating an object, make sure that you do the following things:

■ Initialize it with the IOAudioEngine object that uses it (in this case, your IOAudioEngine subclass instance).

■ Initialize the fields of a IOAudioStreamFormat structure with the values specific to a particular format.

- Call `setSampleBuffer` to pass the actual hardware sample buffer to the stream. If the sample buffer resides in main memory, it should be allocated before you make this call.

  The SamplePCIAudioEngine subclass allocates the sample buffers (input and output) in `initHardware` before it calls `createNewAudioStream`.

- Call `addAvailableFormat` for each format to which the stream can be set. As part of the `addAvailableFormat` call, specify the minimum and maximum sample rates for that format.

- Once you have added all supported formats to an IOAudioStream, call `setFormat` to specify the format the hardware should be set to now. Currently, `performFormatChange` is invoked as a result of the `setFormat` call.

Listing 4-8 (page 52) shows how the SamplePCIAudioEngine subclass creates and initializes an IOAudioStream object.

Listing 4-8   Creating and initializing an IOAudioStream object

```
IOAudioStream
*SamplePCIAudioEngine::createNewAudioStream(IOAudioStreamDirection
            direction, void *sampleBuffer, UInt32 sampleBufferSize)
{
    IOAudioStream *audioStream;

    audioStream = new IOAudioStream;
    if (audioStream) {
        if (!audioStream->initWithAudioEngine(this, direction, 1)) {
            audioStream->release();
        } else {
            IOAudioSampleRate rate;
            IOAudioStreamFormat format = {
                2,      // number of channels
                kIOAudioStreamSampleFormatLinearPCM, // sample format
                kIOAudioStreamNumericRepresentationSignedInt,
                BIT_DEPTH,      // bit depth
                BIT_DEPTH,      // bit width
                kIOAudioStreamAlignmentHighByte,  // high byte aligned
                kIOAudioStreamByteOrderBigEndian, // big endian
                true,      // format is mixable
                0      // driver-defined tag - unused by this driver
            };
            audioStream->setSampleBuffer(sampleBuffer, sampleBufferSize);

            rate.fraction = 0;
            rate.whole = 44100;
            audioStream->addAvailableFormat(&format, &rate, &rate);
            rate.whole = 48000;
            audioStream->addAvailableFormat(&format, &rate, &rate);
            audioStream->setFormat(&format);
        }
    }

    return audioStream;
}
```

Starting and Stopping the I/O Engine

Your IOAudioEngine subclass must implement `performAudioEngineStart` and `performAudioEngineStop` to start and stop the I/O engine. When you start the engine, make sure it starts at the beginning of the sample buffer. Before starting the I/O engine, your implementation should do things:

■ Enable the interrupt event source to allow the I/O engine to fire interrupts as it wraps from the end to the beginning of the sample buffer; in its interrupt handler, the IOAudioEngine instance can continually take time stamps

■ Take an initial timestamp to mark the moment the audio engine started but do so without incrementing the loop count.

By default, the method `takeTimeStamp` automatically increments the current loop count as it takes the current time stamp. But because you are starting a new run of the I/O engine and are not looping, you don't want the loop count to be incremented. To indicate that, pass `false` into `takeTimeStamp`.

Listing 4-9 (page 53) shows how the SamplePCIAudioEngine class implements the `performAudioEngineStart` method; the actual hardware-related code that starts the engine is not supplied.

Listing 4-9   Starting the I/O engine

```
IOReturn SamplePCIAudioEngine::performAudioEngineStart()
{
    IOLog("SamplePCIAudioEngine[%p]::performAudioEngineStart()\n", this);


    assert(interruptEventSource);
    interruptEventSource->enable();

    takeTimeStamp(false);

    // Add audio - I/O start code here

#error performAudioEngineStart() - add engine-start code here; driver will

                            not work without it

    return kIOReturnSuccess;
}
```

In `performAudioEngineStop` be sure to disable the interrupt event source before you stop the I/O engine.

Taking a Time Stamp

A major responsibility of your IOAudioEngine subclass is to take a time stamp each time the I/O engine loops from the end of the sample buffer to the beginning of the sample buffer. Typically, you program the hardware to throw the interrupt when this looping occurs. You must also set up an interrupt handler to receive and process the interrupt. In the interrupt handler, simply call `takeTimeStamp` with no parameters; this method does at least a couple of things:

- It gets the current (machine) time and sets it as the loop time stamp in the `IOAudioEngineStatus`-defined area of memory shared with Audio clients defined.

- It increments the loop count in the same `IOAudioEngineStatus`-defined area of shared memory.

The Audio HAL requires both pieces of updated information so it can track where the I/O engine currently is and predict where it will be in the immediate future.

The SamplePCIAudioEngine subclass uses an IOFilterInterruptEventSource object in its interrupt-handling mechanism. As "Hardware Initialization" (page 49) describes, when the subclass creates this object, it specifies both an interrupt-filter routine and an interrupt-handler routine. The interrupt-handler routine, however, is never called; instead, the interrupt-filter routine calls another routine directly (`filterInterrupt`), which calls `takeTimeStamp`. Listing 4-10 (page 54) shows this code.

Listing 4-10   SamplePCIAudioEngine's interrupt filter and handler

```
bool SamplePCIAudioEngine::interruptFilter(OSObject *owner,
                             IOFilterInterruptEventSource *source)
{
    SamplePCIAudioEngine *audioEngine = OSDynamicCast(SamplePCIAudioEngine,

            owner);

    if (audioEngine) {
        audioEngine->filterInterrupt(source->getIntIndex());
    }
    return false;
}

void SamplePCIAudioEngine::filterInterrupt(int index)
{

    takeTimeStamp();
}
```

Note that you can specify your own time stamp in place of the system's by calling `takeTimeStamp` with a `AbsoluteTime` parameter. This alternative typically isn't necessary, but may be used in cases where the looping isn't detectable until some time after the actual loop time. In that case, the delay can be subtracted from the current time to indicate when the loop occurred in the past.

Providing a Sample Counter

An IOAudioEngine subclass must implement the `getCurrentSampleFrame` to return the current value of a hardware sample counter to the caller. The erase-head process uses this value; it erases (zeroes out) frames in the sample and mix buffers up to, but not including, the sample frame returned by this function.

The sample-counter value doesn't have to be exact, but it should never be larger than the current sample counter. It it is larger, audio data that hasn't been played will be erased.

Implementing Format and Rate Changes

If an audio driver supports multiple audio formats or sample rates, it must implement the `performFormatChange` method to make these changes in the hardware when clients request them. The method has parameters for a new format and for a new sample rate; if either of these parameters is `NULL`, the IOAudioEngine subclass should only change the item that isn't `NULL`.

Although the SamplePCIAudioDriver driver deals with only one audio format, it is capable of two sample rates, 44.1 kilohertz and 48 kilohertz. illustrates how `performFormatChange` is implemented to change a sample rate upon request.

Listing 4-11   Changing the sample rate

```
IOReturn SamplePCIAudioEngine::performFormatChange(IOAudioStream
        *audioStream, const IOAudioStreamFormat *newFormat,
        const IOAudioSampleRate *newSampleRate)
{
    IOLog("SamplePCIAudioEngine[%p]::peformFormatChange(%p, %p, %p)\n",
this,
        audioStream, newFormat, newSampleRate);


    if (newSampleRate) {
        switch (newSampleRate->whole) {
            case 44100:
                IOLog("/t-> 44.1kHz selected\n");

                // Add code to switch hardware to 44.1khz
                break;
            case 48000:
                IOLog("/t-> 48kHz selected\n");

                // Add code to switch hardware to 48kHz
                break;
            default:
                IOLog("/t Internal Error - unknown sample rate
selected.\n");
                break;
        }
    }
    return kIOReturnSuccess;
}
```

Clipping and Converting Samples

Arguably, the most important work that an audio device driver does is converting audio samples between the format expected by the hardware and the format expected by the clients of the hardware. On Mac OS X, the default format of audio data in the kernel as well as in the Audio HAL and all of its clients is 32-bit floating point. However, audio hardware typically requires audio data to be in an integer format.

To perform these conversions, your audio driver must implement at least one of two methods, depending on the directions of the audio streams supported:

- Implement `clipOutputSamples` if your driver has an output IOAudioStream object.

- Implement `convertInputSamples` if your driver has an input IOAudioStream object.

Because these methods execute floating-point code, you cannot include them in the same source file as the other IOAudioEngine methods you implement. The compiler, by default, enables floating-point emulation to prevent floating-point instructions from being generated. To get around this, create a separate library that contains the floating-point code and compile and link this library into the resulting kernel module. The separate library for the SamplePCIAudioDriver project is `libAudioFloatLib`.

Important
See the project configuration for SamplePCIAudioDriver (or any other example audio-driver project) to find out how to generate this separate static library and include it in your project. Pay particular attention to the required compiler options.

The `clipOutputSamples` method is passed six parameters:

- a pointer to the start of the mix buffer

- a pointer to the start of the sample buffer

- an index to the first sample frame in the buffers to clip and convert

- the number of sample frames to clip and convert

- a pointer to the current format (structure `IOAudioFormat`) of the audio stream

- a pointer to the IOAudioStream object this method is working on

Your implementation must first clip any floating-point samples in the mix buffer that exceed -1.0 or 1.0 and then convert the floating-point value to the comparable value in the format expected by the hardware. Then copy that value to the corresponding positions in the sample buffer. Listing 4-12 (page 56) illustrates how the SamplePCIAudioDriver implements the `clipOutputSamples` method.

Listing 4-12   Clipping and converting output samples

```
IOReturn SamplePCIAudioEngine::clipOutputSamples(const void *mixBuf,
        void *sampleBuf, UInt32 firstSampleFrame, UInt32 numSampleFrames,
```

```
        const IOAudioStreamFormat *streamFormat, IOAudioStream
*audioStream)
{
    UInt32 sampleIndex, maxSampleIndex;
    float *floatMixBuf;
    SInt16 *outputBuf;

    floatMixBuf = (float *)mixBuf;
    outputBuf = (SInt16 *)sampleBuf;

    maxSampleIndex = (firstSampleFrame + numSampleFrames) *
                        streamFormat->fNumChannels;

    for (sampleIndex = (firstSampleFrame * streamFormat->fNumChannels);
                        sampleIndex < maxSampleIndex; sampleIndex++) {
        float inSample;
        inSample = floatMixBuf[sampleIndex];

        // Note: A softer clipping operation could be done here
        if (inSample > 1.0) {
            inSample = 1.0;
        } else if (inSample < -1.0) {
            inSample = -1.0;
        }
        if (inSample >= 0) {
            outputBuf[sampleIndex] = (SInt16) (inSample * 32767.0);
        } else {
            outputBuf[sampleIndex] = (SInt16) (inSample * 32768.0);
        }
    }
    return kIOReturnSuccess;
}
```

Here are a few comments on this specific example:

1. It starts by casting the void *-typed buffers to float * for the mix buffer and SInt16 *
    for the sample buffer; in this project, the hardware uses signed 16-bit integers for its samples
    while the mix buffer is always float *.

2. Next it calculates the upper limit on the sample index for the upcoming clipping and
    converting loop.

3. The method loops through the mix and sample buffers and performs the clip and conversion
    operations on one sample frame at a time.

    a. It fetches the floating-point sample from the mix buffer and clips it (if necessary) to a
        range between -1.0 and 1.0.

    b. Finally, it scales and converts the floating-point value to the appropriate signed 16-bit
        integer sample and writes it to the corresponding location in the sample buffer.

The parameters passed into the convertInputSamples method are almost the same as those for
the clipOutputSamples method. The only difference is that, instead of a pointer to the mix
buffer, a pointer to a floating-point destination buffer is passed; this is the buffer that the Audio
HAL uses. In your driver's implementation of this method, do the opposite of the

`clipOutputSamples`: convert from the hardware format to the system 32-bit floating point format. No clipping is necessary because your conversion process can control the bounds of the floating-point values.

Listing 4-13 (page 58) shows how the SamplePCIAudioDriver project implements this method.

Listing 4-13 Converting input samples.

```
IOReturn SamplePCIAudioEngine::convertInputSamples(const void *sampleBuf,

        void *destBuf, UInt32 firstSampleFrame, UInt32 numSampleFrames,
        const IOAudioStreamFormat *streamFormat, IOAudioStream
        *audioStream)
{
    UInt32 numSamplesLeft;
    float *floatDestBuf;
    SInt16 *inputBuf;

    floatDestBuf = (float *)destBuf;
    inputBuf = &(((SInt16 *)sampleBuf)[firstSampleFrame *
                           streamFormat->fNumChannels]);

    numSamplesLeft = numSampleFrames * streamFormat->fNumChannels;

    while (numSamplesLeft > 0) {
        SInt16 inputSample;
        inputSample = *inputBuf;

        if (inputSample >= 0) {
            *floatDestBuf = inputSample / 32768.0;
        } else {
            *floatDestBuf = inputSample / 32768.0;
        }

        ++inputBuf;
        ++floatDestBuf;
        --numSamplesLeft;
    }

    return kIOReturnSuccess;
}
```

This code does the following things:

1. It starts by casting the destination buffer to a `float *`.

2. It casts the sample buffer to signed 16-bit integer and determines the starting point within this input buffer for conversion.

3. It calculates the number of actual samples to convert.

4. It loops through each sample, scales it to within a range of -1.0 to 1.0, converts it to float, and stores it in the destination buffer at the proper location.
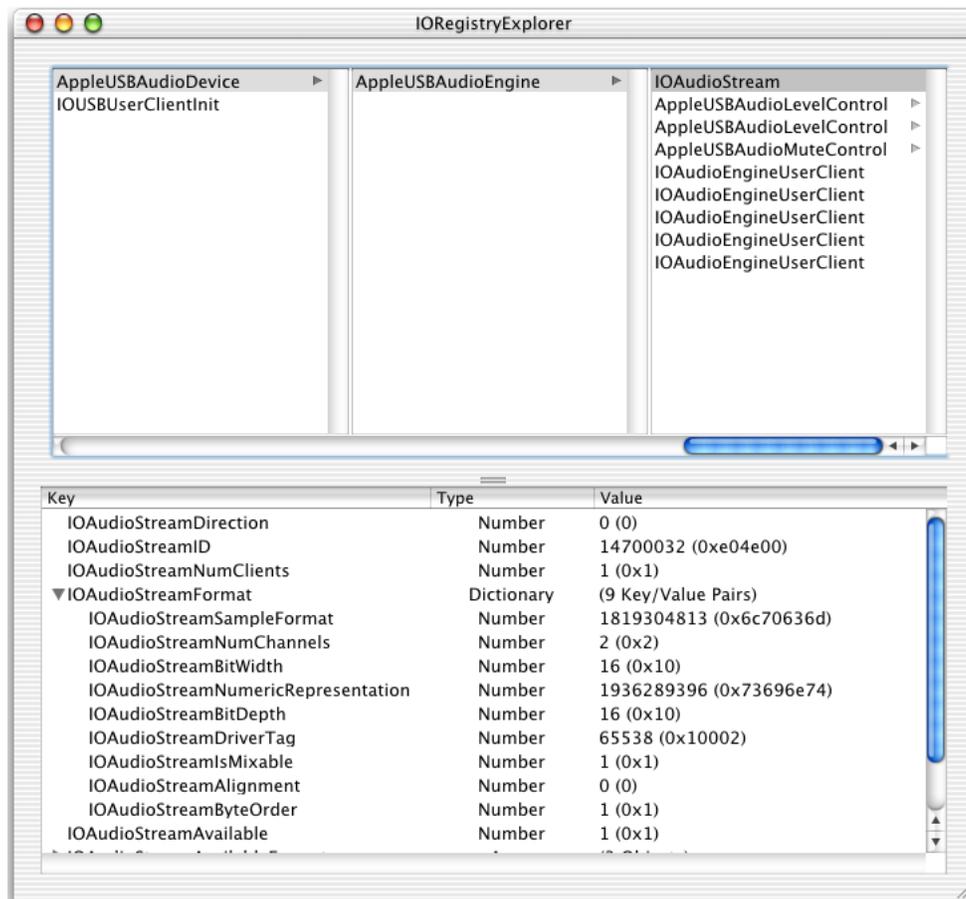
Debugging and Testing the Driver

Many of the techniques you would use in debugging and testing an audio driver are the same ones you'd use with other types of device drivers. After all, any I/O Kit driver has a structure and a behavior that are similar to any other I/O Kit driver, regardless of family.

For example, it's always a good idea when a driver is under development to make `IOLog` calls at critical points in your code, such as before and after an I/O transfer. The `IOLog` function writes a message to the console (accessible through the Console application) and to `/var/log/system.log`. You can format the message string with variable data in the style of `printf`.

Similarly, you can examine the I/O Registry with the I/O Registry Explorer application or the `ioreg` command-line utility. The I/O Registry will show the position of your driver's objects in the driver stack, the client-provider relationships among them, and the attributes of those driver objects. In Figure 4-2 (page 59), the I/O Registry Explorer shows part of the objects and their attributes in a USB audio device driver.

**Figure 4-2**   The I/O Registry (via I/O Registry Explorer)

However, as "Custom Debugging Information in the I/O Registry" (page 62) explains, your driver can insert information in the I/O Registry to assist the testing and debugging of your driver.

Tools for Testing Audio Drivers

The Mac OS X Developer package provides an application and a command-line tool that are helpful when you're testing audio software. These items are not shipped as executables, but are instead included as example-code projects installed in `/Developer/Examples/CoreAudio/HAL`. The application project is called Daisy and the command-line tool project is named DisplayHALDeviceInfo. To obtain the executables, copy the project folders to your home directory (or any file-system location where you have write-access) and build the projects.

The Daisy application helps you verify the controls and other attributes of a loaded audio driver. With it, you can play back sound files to any channel of device, check whether muting and volume changes work for every channel, and do many other tests. Figure 4-3 (page 60) shows you what the Daisy application looks like.

Figure 4-3   The Daisy application
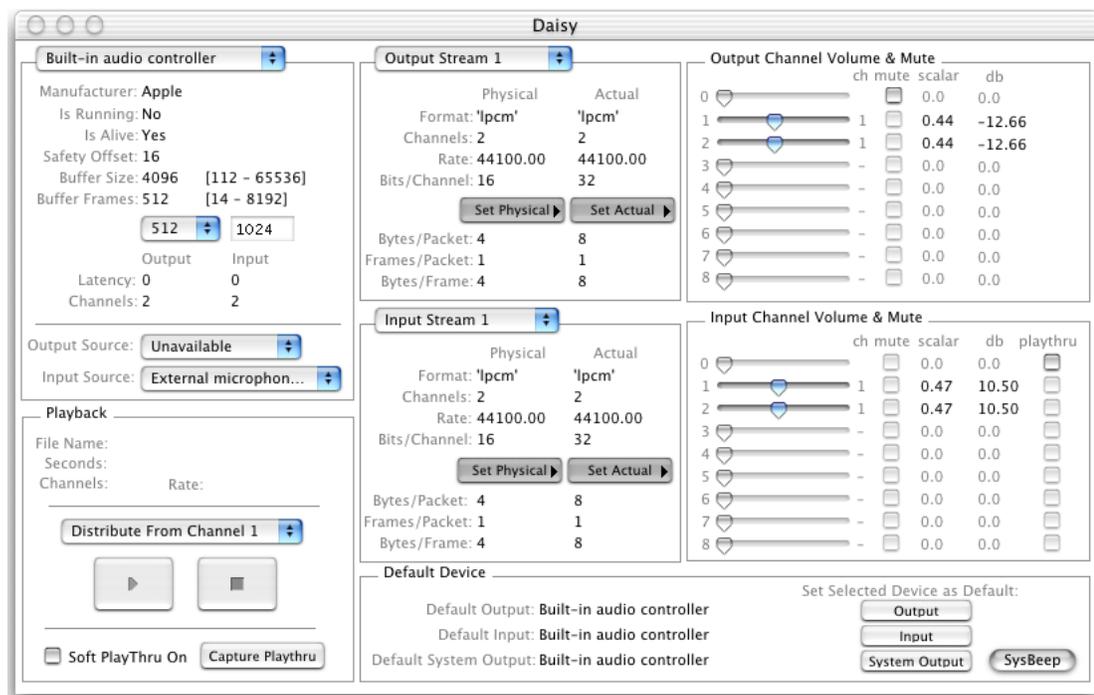
The `displayhaldeviceinfo` command-line utility has functionality similar to Daisy. Basically, it tells you what the Audio HAL thinks your device is capable of. Listing 4-14 (page 61) gives some typical output of this tool.

Listing 4-14   Output from the displayhaldeviceinfo utility

```
DisplayHALDeviceInfo, v2.2

*************************

Default input device: 0x0
Default output device: 0xA
Default system output device: 0xA

*************************

Number of devices: 1

-------------------------

Device #1
  ID: 0xA
  Name: SoundSticks
  Manufacturer: harman/kardon
  UID: AppleUSBAudioEngine:harman/kardon:SoundSticks:19100000:1
  Buffer size in frames: 512

  ========================
  Input stream configuration
    Number streams: 0

  ========================
  Output stream configuration
    Number streams: 1
      Channels in stream 1: 2
      Buffer size of stream 1: 4096
    Output stream: 1
    ID: 0xB
    Couldn't get the name, OSStatus code: who?
    Direction: Output
    Starting channel: 1
    ---------------------
    Current format
      Format ID: lpcm
      Format flags: 0xB
      Sample rate: 44100.000
      Bytes per packet: 8
      Frames per packet: 1
      Bytes per frame: 8
      Channels per frame: 2
      Bits per channel: 32
    ---------------------
    Current physical format
      Format ID: lpcm
      Format flags: 0xC
      Sample rate: 44100.000
      Bytes per packet: 4
      Frames per packet: 1
      Bytes per frame: 4
      Channels per frame: 2
      Bits per channel: 16
```

Custom Debugging Information in the I/O Registry

Another way you can test and debug your audio device driver is to write custom properties to the I/O Registry. For example, you may want to track hardware register state or internal driver state (if the driver has any). Whenever your driver makes a change in the hardware, it could read the hardware register values and call `setProperty` with the current value. Then, when testing the driver, run the I/O Registry Explorer application and note what the registry shows this value to be. This technique allows you to easily determine if the driver is putting the hardware in the correct state.

Document Revision History

Table RH-1 (page 63) describes revisions to *Writing Audio Device Drivers*.

Table RH-1

Date	Notes
March 25, 2004	Added installed location of SamplePCIAudioDriver project.
December 15, 2001	First version.