Core Audio Overview

Developer

Contents

Figures, Tables, and Listings

Introduction

Core Audio provides software interfaces for implementing audio features in applications you create for iOS and OS X. Under the hood, it handles all aspects of audio on each of these platforms. In iOS, Core Audio capabilities include recording, playback, sound effects, positioning, format conversion, and file stream parsing, as well as:

- A built-in equalizer and mixer that you can use in your applications
- Automatic access to audio input and output hardware
- APIs to let you manage the audio aspects of your application in the context of a device that can take phone calls
- Optimizations to extend battery life without impacting audio quality

On the Mac, Core Audio encompasses recording, editing, playback, compression and decompression, MIDI, signal processing, file stream parsing, and audio synthesis. You can use it to write standalone applications or modular effects and codec plug-ins that work with existing products.

Core Audio combines C and Objective-C programming interfaces with tight system integration, resulting in a flexible programming environment that maintains low latency through the signal chain.

Core Audio is available in all versions of OS X, although older versions may not contain some features described here. Core Audio is available in iOS starting with version 2.0. This document describes Core Audio features available as of iOS 2.2 and OS X v10.5.

> **Note:** Core Audio does not provide direct support for audio digital rights management (DRM). If you need DRM support for audio files, you must implement it yourself.

Core Audio Overview is for all developers interested in creating audio software. Before reading this document you should have basic knowledge of general audio, digital audio, and MIDI terminology. You will also do well to have some familiarity with object-oriented programming concepts and with Apple's development environment, Xcode. If you are developing for iOS-based devices, you should be familiar with Cocoa Touch development as introduced in *iOS App Programming Guide*.

Organization of This Document

This document is organized into the following chapters:

- "What Is Core Audio?" (page 10) describes the features of Core Audio and what you can use it for.
- "Core Audio Essentials" (page 19) describes the architecture of Core Audio, introduces you to its programming patterns and idioms, and shows you the basics of how to use it in your applications.
- "Common Tasks in OS X" (page 61) outlines how you can use Core Audio to accomplish several audio tasks in OS X.

This document also contains four appendixes:

- "Core Audio Frameworks" (page 74) lists the frameworks and headers that define Core Audio.
- "Core Audio Services" (page 79) provides an alternate view of Core Audio, listing the services available in iOS, OS X, and on both platforms.
- "System-Supplied Audio Units in OS X" (page 85) lists the audio units that ship in OS X v10.5.
- "Supported Audio File and Data Formats in OS X" (page 89) lists the audio file and data formats that Core Audio supports in OS X v10.5.

See Also

For more detailed information about audio and Core Audio, see the following resources:

- *AVAudioPlayer Class Reference*, which describes a simple Objective-C interface for audio playback in iOS applications.
- *Audio Session Programming Guide*, which explains how to specify important aspects of audio behavior for iOS applications.
- *Audio Queue Services Programming Guide*, which explains how to implement recording and playback in your application.
- *Core Audio Data Types Reference*, which describes the data types used throughout Core Audio.
- *Audio File Stream Services Reference*, which describes the interfaces you use for working with streamed audio.
- *Audio Unit Programming Guide*, which contains detailed information about creating audio units for OS X.
- *Core Audio Glossary*, which defines terms used throughout the Core Audio documentation suite.
- *Apple Core Audio Format Specification 1.0*, which describes Apple's universal audio container format, the Core Audio File (CAF) format.

- The Core Audio mailing list: http://lists.apple.com/mailman/listinfo/coreaudio-api
- The OS X audio developer site: http://developer.apple.com/audio/
- The Core Audio SDK (software development kit), available at http://developer.apple.com/sdk/

What Is Core Audio?

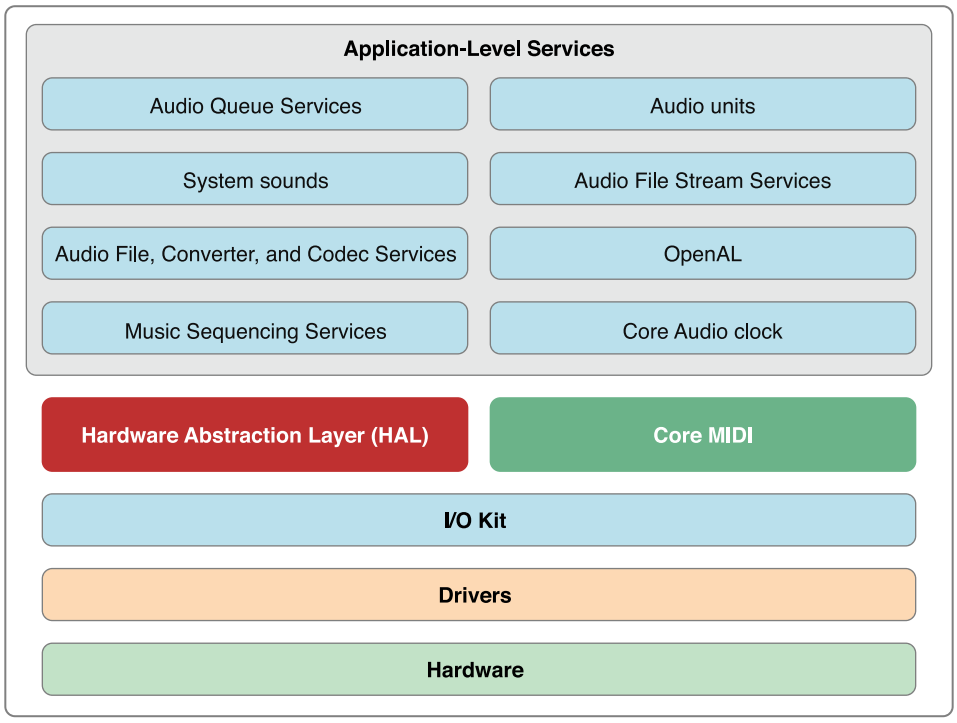
Core Audio is the digital audio infrastructure of iOS and OS X. It includes a set of software frameworks designed to handle the audio needs in your applications. Read this chapter to learn what you can do with Core Audio.

Core Audio in iOS and OS X

Core Audio is tightly integrated into iOS and OS X for high performance and low latency.

In OS X, the majority of Core Audio services are layered on top of the Hardware Abstraction Layer (HAL) as shown in Figure 1-1. Audio signals pass to and from hardware through the HAL. You can access the HAL using Audio Hardware Services in the Core Audio framework when you require real-time audio. The Core MIDI (Musical Instrument Digital Interface) framework provides similar interfaces for working with MIDI data and devices.

Figure 1-1 OS X Core Audio architecture

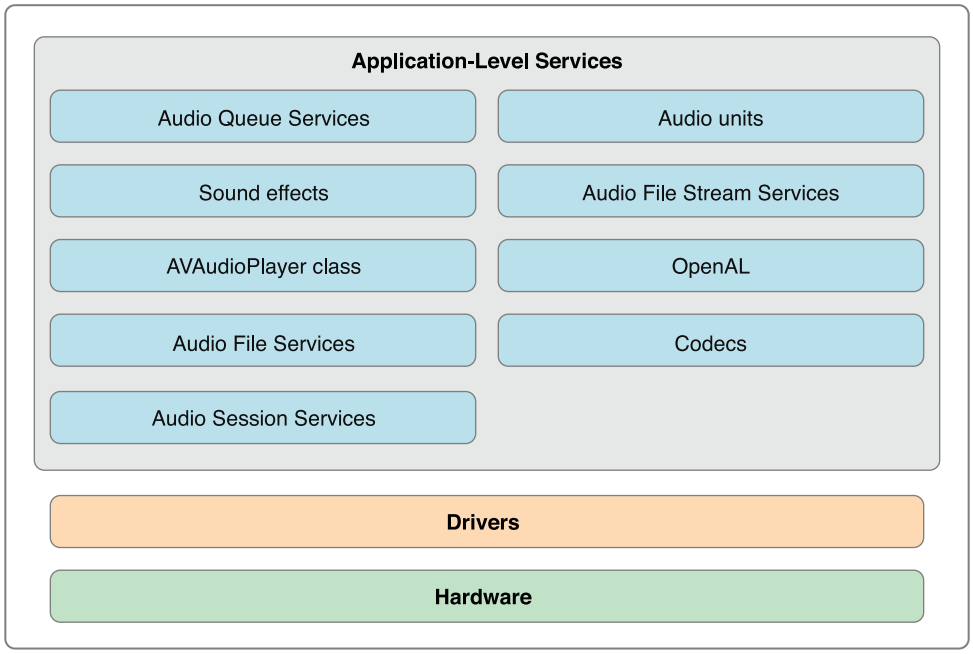


You find Core Audio application-level services in the Audio Toolbox and Audio Unit frameworks.

- Use Audio Queue Services to record, play back, pause, loop, and synchronize audio.
- Use Audio File, Converter, and Codec Services to read and write from disk and to perform audio data format transformations. In OS X you can also create custom codecs.
- Use Audio Unit Services and Audio Processing Graph Services (represented in the figure as "Audio units") to host audio units (audio plug-ins) in your application. In OS X you can also create custom audio units to use in your application or to provide for use in other applications.
- Use Music Sequencing Services to play MIDI-based control and music data.
- Use Core Audio Clock Services for audio and MIDI synchronization and time format management.
- Use System Sound Services (represented in the figure as "System sounds") to play system sounds and user-interface sound effects.

Core Audio in iOS is optimized for the computing resources available in a battery-powered mobile platform. There is no API for services that must be managed very tightly by the operating system—specifically, the HAL and the I/O Kit. However, there are additional services in iOS not present in OS X. For example, Audio Session Services lets you manage the audio behavior of your application in the context of a device that functions as a mobile telephone and an iPod. Figure 1-2 provides a high-level view of the audio architecture in iOS.

Figure 1-2 iOS Core Audio architecture



A Little About Digital Audio and Linear PCM

Most Core Audio services use and manipulate audio in linear pulse-code-modulated (**linear PCM**) format, the most common uncompressed digital audio data format. Digital audio recording creates PCM data by measuring an analog (real world) audio signal's magnitude at regular intervals (the **sampling rate**) and converting each sample to a numerical value. Standard compact disc (CD) audio uses a sampling rate of 44.1 kHz, with a 16-bit integer describing each sample—constituting the resolution or **bit depth**.

- A **sample** is single numerical value for a single channel.
- A **frame** is a collection of time-coincident samples. For instance, a stereo sound file has two samples per frame, one for the left channel and one for the right channel.
- A **packet** is a collection of one or more contiguous frames. In linear PCM audio, a packet is always a single frame. In compressed formats, it is typically more. A packet defines the smallest meaningful set of frames for a given audio data format.

In linear PCM audio, a sample value varies linearly with the amplitude of the original signal that it represents. For example, the 16-bit integer samples in standard CD audio allow 65,536 possible values between silence and maximum level. The difference in amplitude from one digital value to the next is always the same.

Core Audio data structures, declared in the `CoreAudioTypes.h` header file, can describe linear PCM at any sample rate and bit depth. "Audio Data Formats" (page 25) goes into more detail on this topic.

In OS X, Core Audio expects audio data to be in native-endian, 32-bit floating-point, linear PCM format. You can use Audio Converter Services to translate audio data between different linear PCM variants. You also use these converters to translate between linear PCM and compressed audio formats such as MP3 and Apple Lossless. Core Audio in OS X supplies codecs to translate most common digital audio formats (though it does not supply an encoder for converting to MP3).

iOS uses integer and fixed-point audio data. The result is faster calculations and less battery drain when processing audio. iOS provides a Converter audio unit and includes the interfaces from Audio Converter Services. For details on the so-called *canonical* audio data formats for iOS and OS X, see "Canonical Audio Data Formats" (page 27).

In iOS and OS X, Core Audio supports most common file formats for storing and playing audio data, as described in "iPhone Audio File Formats" (page 35) and "Supported Audio File and Data Formats in OS X" (page 89).

Audio Units

Audio units are software plug-ins that process audio data. In OS X, a single audio unit can be used simultaneously by an unlimited number of channels and applications.

iOS provides a set of audio units optimized for efficiency and performance on a mobile platform. You can develop audio units for use in your iOS application. Because you must statically link custom audio unit code into your application, audio units that you develop cannot be used by other applications in iOS.

The audio units provided in iOS do not have user interfaces. Their main use is to provide low-latency audio in your application. For more on iPhone audio units, see "Core Audio Plug-ins: Audio Units and Codecs" (page 50).

In Mac apps that you develop, you can use system-supplied or third-party-supplied audio units. You can also develop an audio unit as a product in its own right. Users can employ your audio units in applications such as GarageBand and Logic Studio, as well as in many other audio unit hosting applications.

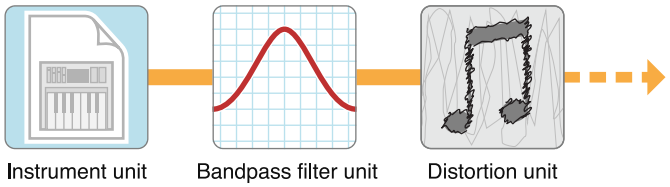
Some Mac audio units work behind the scenes to simplify common tasks for you—such as splitting a signal or interfacing with hardware. Others appear onscreen, with their own user interfaces, to offer signal processing and manipulation. For example, effect units can mimic their real-world counterparts, such as a guitarist's distortion box. Other audio units generate signals, whether programmatically or in response to MIDI input.

Some examples of audio units are:

- A signal processor (for example, a high-pass filter, reverb, compressor, or distortion unit). Each of these is generically an **effect unit** and performs digital signal processing (DSP) in a way similar to a hardware effects box or outboard signal processor.
- A musical instrument or software synthesizer. These are called **instrument units** (or, sometimes, music devices) and typically generate musical notes in response to MIDI input.
- A signal source. Unlike an instrument unit, a **generator unit** is not activated by MIDI input but rather through code. For example, a generator unit might calculate and generate sine waves, or it might source the data from a file or network stream.
- An interface to hardware input or output. For more information on **I/O units**, see "The Hardware Abstraction Layer" (page 14) and "Interfacing with Hardware" (page 63).
- A format converter. A **converter unit** can translate data between two linear PCM variants, merge or split audio streams, or perform time and pitch changes. See "Core Audio Plug-ins: Audio Units and Codecs" (page 50) for details.
- A mixer or panner. A **mixer unit** can combine audio tracks. A **panner unit** can apply stereo or 3D panning effects.
- An effect unit that works offline. An **offline effect unit** performs work that is either too processor-intensive or simply impossible in real time. For example, an effect that performs time reversal on a file must be applied offline.

In OS X you can mix and match audio units in whatever permutations you or your end user requires. Figure 1-3 shows a simple chain of audio units. There's an instrument unit to generate an audio signal based on control data received from an outboard MIDI keyboard. The generated audio then passes through effect units to apply bandpass filtering and distortion. A chain of audio units is called an **audio processing graph**.

Figure 1-3 A simple audio processing graph



If you develop audio DSP code that you want to make available to multiple applications, you should package your code as an audio unit.

If you develop Mac audio apps, supporting audio units lets you and your users leverage the library of existing audio units (both third-party and Apple-supplied) to extend the capabilities of your application.

To experiment with audio units in OS X, see the AU Lab application, available in the Xcode Tools installation at `/Developer/Applications/Audio`. AU Lab lets you mix and match audio units to build a signal chain from an audio source through an output device.

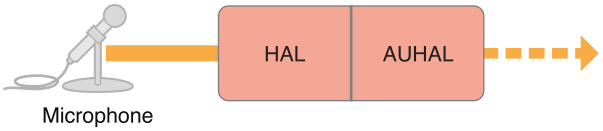
See "System-Supplied Audio Units in OS X" (page 85) for a listing of the audio units that ship with OS X v10.5 and iOS 2.0.

The Hardware Abstraction Layer

Core Audio uses a hardware abstraction layer (HAL) to provide a consistent and predictable interface for applications to interact with hardware. The HAL can also provide timing information to your application to simplify synchronization or to adjust for latency.

In most cases, your code does not interact directly with the HAL. Apple supplies a special audio unit—called the AUHAL unit in OS X and the AURemoteIO unit in iOS—which allows you to pass audio from another audio unit to hardware. Similarly, input coming from hardware is routed through the AUHAL unit (or the AURemoteIO unit in iOS) and made available to subsequent audio units, as shown in Figure 1-4.

Figure 1-4 Hardware input through the HAL and the AUHAL unit



The AUHAL unit (or AURemoteIO unit) also takes care of any data conversion or channel mapping required to translate audio data between audio units and hardware.

MIDI Support in OS X

Core MIDI is the part of Core Audio that supports the MIDI protocol. (MIDI is not available in iOS.) Core MIDI allows applications to communicate with MIDI devices such as keyboards and guitars. Input from MIDI devices can be stored as MIDI data or used to control an instrument unit. Applications can also send MIDI data to MIDI devices.

Core MIDI uses abstractions to represent MIDI devices and mimic standard MIDI cable connections (MIDI In, MIDI Out, and MIDI Thru) while providing low-latency input and output. Core Audio also supports a music player programming interface that you can use to play MIDI-based control or music data.

For more details about the capabilities of the MIDI protocol, see the MIDI Manufacturers Association site, http://midi.org.

The Audio MIDI Setup Application

The Audio MIDI Setup application lets users:

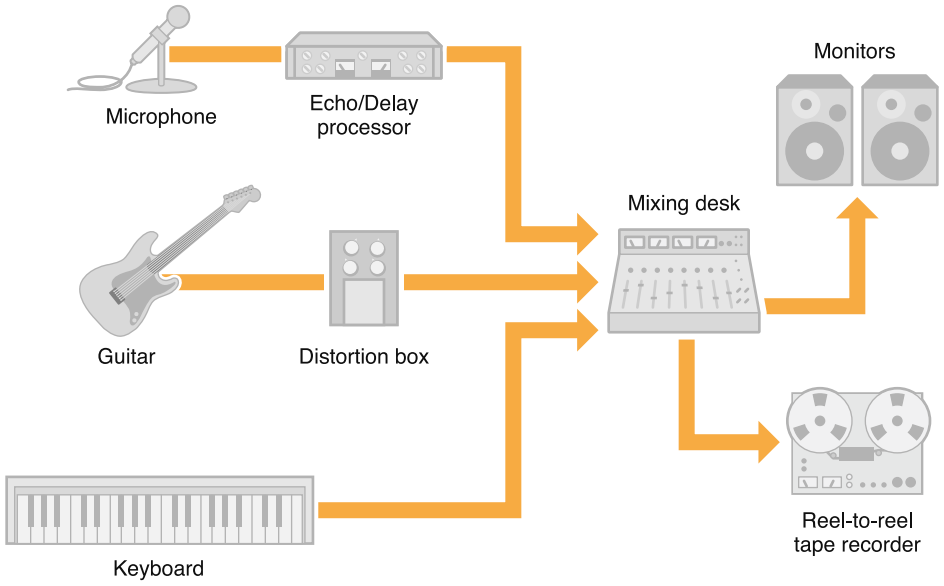
- Specify the default audio input and output devices.
- Configure properties for input and output devices, such as sampling rate and bit depth.
- Map audio channels to available speakers (for stereo, 5.1 surround, and so on).
- Create aggregate devices. (For information about aggregate devices, see "Using Aggregate Devices" (page 65).)
- Configure MIDI networks and MIDI devices.

You find Audio MIDI Setup in the `/Applications/Utilities` folder.

A Mac Core Audio Recording Studio

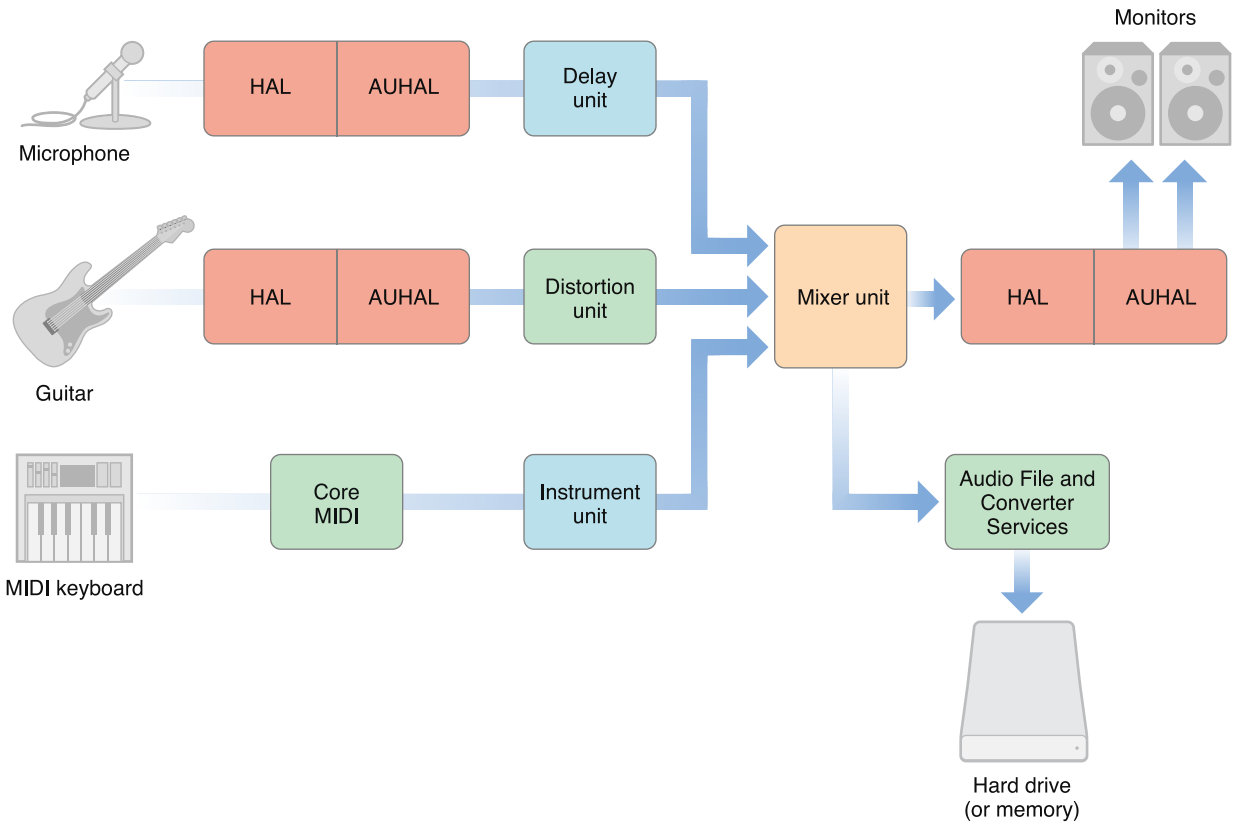
A traditional—non-computer-based—recording studio can serve as a conceptual framework for approaching Core Audio. Such a studio may have a few "real" instruments and effect units feeding a mixing desk, as shown in Figure 1-5. The mixer can route its output to studio monitors and a recording device (shown here, in a rather retro fashion, as a tape recorder).

Figure 1-5 A simple non-computer-based recording studio



Many of the pieces in a traditional studio can be replaced by software-based equivalents—all of which you have already met in this chapter. On a desktop computing platform, digital audio applications can record, synthesize, edit, mix, process, and play back audio. They can also record, edit, process, and play back MIDI data, interfacing with both hardware and software MIDI instruments. Mac apps rely on Core Audio services to handle all of these tasks, as shown in Figure 1-6.

Figure 1-6 A Core Audio "recording studio"



As you can see, audio units can make up much of an audio signal chain. Other Core Audio interfaces provide application-level support, allowing applications to obtain audio or MIDI data in various formats and output it to files or output devices. "Core Audio Services" (page 79) discusses the constituent interfaces of Core Audio in more detail.

Core Audio lets you do much more than mimic a recording studio on a desktop computer. You can use it for everything from playing sound effects to creating compressed audio files to providing an immersive sonic experience for game players.

On a mobile device such as an iPhone or iPod touch, the audio environment and computing resources are optimized to extend battery life. After all, an iPhone's most essential identity is as a telephone. From a development or user perspective, it wouldn't make sense to place an iPhone at the heart of a virtual recording

studio. On the other hand, an iPhone's special capabilities—including extreme portability, built-in Bonjour networking, multitouch interface, and accelerometer and location APIs—let you imagine and create audio applications that were never possible on the desktop.

Mac Development Using the Core Audio SDK

To assist audio developers, Apple supplies a software development kit (SDK) for Core Audio in OS X. The SDK contains many code samples covering both audio and MIDI services as well as diagnostic tools and test applications. Examples include:

- A test application to interact with the global audio state of the system, including attached hardware devices (HALLab).
- A reference audio unit hosting application (AU Lab). The AU Lab application is essential for testing audio units you create, as described in "Audio Units" (page 12).
- Sample code to load and play audio files (PlayFile) and MIDI files (PlaySequence).

This document points to additional examples in the Core Audio SDK that illustrate how to accomplish common tasks.

The SDK also contains a C++ framework for building audio units for OS X. This framework simplifies the amount of work you need to do by insulating you from the details of the Component Manager plug-in interface. The SDK also contains templates for common audio unit types; for the most part, you only need override those methods that apply to your custom audio unit. Some sample audio unit projects show these templates and frameworks in use. For more details on using the framework and templates, see *Audio Unit Programming Guide* .

> **Note:** Apple supplies the C++ audio unit framework as sample code to assist audio unit development. Feel free to modify and adapt the framework based on your needs.

The Core Audio SDK assumes you will use Xcode as your development environment.

You can download the latest SDK from http://developer.apple.com/sdk/. After installation, the SDK files are located in `/Developer/Examples/CoreAudio`. The HALLab and AU Lab applications are located in `/Developer/Applications/Audio`.

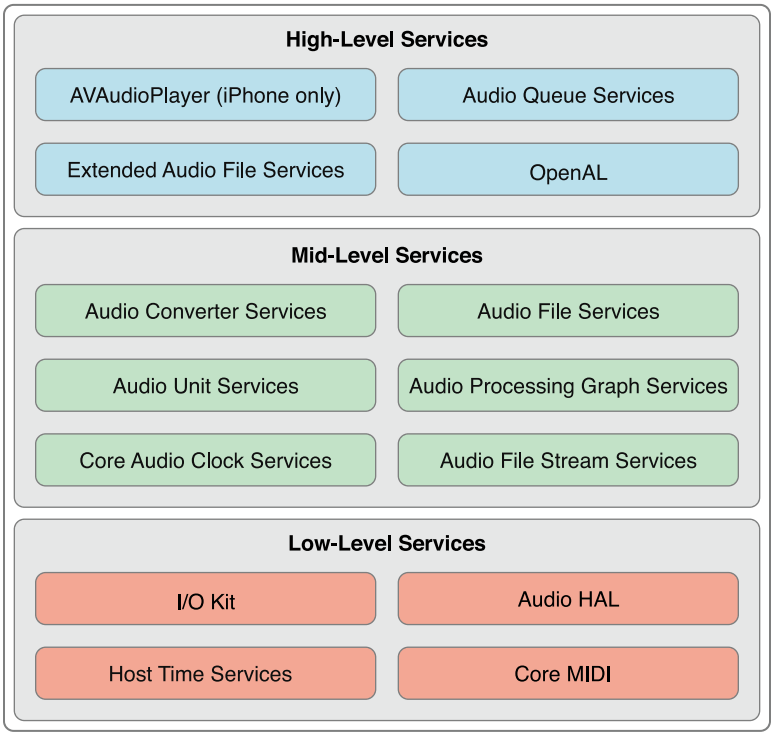
Core Audio Essentials

Apple has designed the software interfaces to Core Audio using a layered, cooperative, task-focused approach. Read the first two sections in this chapter for a brief introduction to these interfaces and how they work together. Continue reading to understand the design principles, use patterns, and programming idioms that pervade Core Audio. The later sections in this chapter introduce you to how Core Audio works with files, streams, recording and playback, and plug-ins.

API Architectural Layers

The programming interfaces for Core Audio are arranged into three layers, as illustrated in Figure 2-1.

Figure 2-1 The three API layers of Core Audio



The lowest layer includes:

- The I/O Kit, which interacts with drivers

- The audio hardware abstraction layer (audio HAL), which provides a device-independent, driver-independent interface to hardware
- Core MIDI, which provides software abstractions for working with MIDI streams and devices
- Host Time Services, which provides access to the computer's clock

Mac apps can be written to use these technologies directly when they require the highest possible, real-time performance. Many audio applications, however, don't access this layer. Indeed, Core Audio in iOS provides ways to achieve real-time audio using higher level interfaces. OpenAL, for example, employs direct I/O for real-time audio in games. The result is a significantly smaller, tuned API set appropriate for a mobile platform.

The middle layer in Core Audio includes services for data format conversion, reading and writing to disk, parsing streams, and working with plug-ins.

- Audio Converter Services lets applications work with audio data format converters.
- Audio File Services supports reading and writing audio data to and from disk-based files.
- Audio Unit Services and Audio Processing Graph Services let applications work with digital signal processing (DSP) plug-ins such as equalizers and mixers.
- Audio File Stream Services lets you build applications that can parse streams, such as for playing files streamed over a network connection.
- Core Audio Clock Services supports audio and MIDI synchronization as well as time-base conversions.
- Audio Format Services (a small API, not shown in the figure) assists with managing audio data formats in your application.

Core Audio in iOS supports most of these services, as shown in Figure 1-2 (page 11).

The highest layer in Core Audio includes streamlined interfaces that combine features from lower layers.

- Audio Queue Services lets you record, play, pause, loop, and synchronize audio. It employs codecs as necessary to deal with compressed audio formats.
- The `AVAudioPlayer` class provides a simple Objective-C interface for playing and looping audio in iOS applications. The class handles all audio formats supported in iOS, and provides a straightforward means to implement features such as rewind and fast-forward.
- Extended Audio File Services combines features from Audio File Services and Audio Converter services. It gives you a unified interface for reading and writing uncompressed and compressed sound files.
- OpenAL is the Core Audio implementation of the open-source OpenAL standard for positional audio. It is built on top of the system-supplied 3D Mixer audio unit. All applications can use OpenAL, although it is best suited for games development.

Frameworks

You get another view of Core Audio by considering its API frameworks, located in `/System/Library/Frameworks/`. This section quickly lists them to give you a sense of where to find the pieces that make up the Core Audio layers.

Take note that the Core Audio framework is not an umbrella to the other frameworks here, but rather a peer.

- The Audio Toolbox framework (`AudioToolbox.framework`) provides interfaces for the mid- and high-level services in Core Audio. In iOS, this framework includes Audio Session Services, the interface for managing your application's audio behavior in the context of a device that functions as a mobile phone and iPod.
- The Audio Unit framework (`AudioUnit.framework`) lets applications work with audio plug-ins, including audio units and codecs.
- The AV Foundation framework (`AVFoundation.framework`), available in iOS, provides the `AVAudioPlayer` class, a streamlined and simple Objective-C interface for audio playback.
- The Core Audio framework (`CoreAudio.framework`) supplies data types used across Core Audio as well as interfaces for the low-level services.
- The Core Audio Kit framework (`CoreAudioKit.framework`) provides a small API for creating user interfaces for audio units. This framework is not available in iOS.
- The Core MIDI framework (`CoreMIDI.framework`) lets applications work with MIDI data and configure MIDI networks. This framework is not available in iOS.
- The Core MIDI Server framework (`CoreMIDIServer.framework`) lets MIDI drivers communicate with the OS X MIDI server. This framework is not available in iOS.
- The OpenAL framework (`OpenAL.framework`) provides the interfaces to work with OpenAL, an open source, positional audio technology.

The Appendix "Core Audio Frameworks" (page 74) describes all these frameworks, as well as each of their included header files.

Proxy Objects

Core Audio uses the notion of **proxy objects** to represent such things as files, streams, audio players, and so on. When you want your application to work with an on-disk audio file, for example, the first step is to instantiate an audio file object of type `AudioFileID`. This object is declared as an opaque data structure in the `AudioFile.h` header file:

```
typedef struct OpaqueAudioFileID *AudioFileID;
```

You instantiate an audio file object—and create an actual audio file tied to that object—by calling the `AudioFileCreateWithURL` function. The function gives you a reference to the new audio file object. From that point on, you work with the real audio file by communicating with the proxy object.

This sort of pattern is consistent throughout Core Audio, whether you are working with audio files, iPhone audio sessions (described in "Audio Sessions: Cooperating with Core Audio" (page 37)), or even hardware devices.

Properties, Scopes, and Elements

Most Core Audio interfaces use a **property** mechanism for managing object state or refining object behavior. A property is a key-value pair.

- A property key is typically an enumerator constant with a mnemonic name, such as `kAudioFilePropertyFileFormat` or `kAudioQueueDeviceProperty_NumberChannels`.
- A property value is of a particular data type appropriate for the purpose of the property—a `void*`, a `Float64`, an `AudioChannelLayout` structure, and so on.

There are many Apple-defined properties. You'll find their definitions in the various Core Audio framework header files. Some Core Audio interfaces, such as Audio Unit Services, let you define your own properties as well.

Core Audio interfaces use accessor functions for retrieving a property value from an object and, in the case of a writable property, changing its value. You'll also find a third accessor function for getting information about properties. For example, the Audio Unit Services function `AudioUnitGetPropertyInfo` tells you a given property value data type's size and whether you can change it. The Audio Queue Services function `AudioQueueGetPropertySize` gets a specified property value's size.

Core Audio interfaces provide a mechanism for informing your application that a property has changed. You can read about this in the next section, "Callback Functions: Interacting with Core Audio" (page 23).

In some cases, a property applies to an audio object as a whole. For example, to enable audio level metering in a playback audio queue object, you set the value of its `kAudioQueueProperty_EnableLevelMetering` property to `true`.

Other Core Audio objects have an internal structure, each part of which may have its own set of properties. For example, an audio unit has an input scope, an output scope, and a global scope. An audio unit's input or output scope consists of one or more elements—each of which is analogous to a channel bus in audio hardware.

When you call the `AudioUnitGetProperty` function with the `kAudioUnitProperty_AudioChannelLayout` property, you specify not only the audio unit you want information about but also the scope (input or output) and element (0, 1, 2, etc.).

Callback Functions: Interacting with Core Audio

Many Core Audio interfaces can communicate with your application using callback functions. Core Audio uses callbacks for such things as:

- Delivering a new set of audio data to your application (such as for recording; your callback then writes the new data to disk).
- Requesting a new set of audio data from your application (such as for playback; your callback reads from disk and provides the data).
- Telling your application that a software object has changed state (your callback takes appropriate action).

One way to understand callbacks is to reverse your perspective on who calls whom. In a normal function call, such as `AudioQueueNewOutput`, your application invokes behavior that is defined by Apple in the implementation of the operating system. You don't know—and don't need to know—what goes on under the hood. Your application requests a playback audio queue object and gets one back. It works because, in making the call, you adhere to the function interface specified in the function's header file.

In the case of a callback, the operating system—when it chooses to—invokes behavior that you implement in your application. By defining a callback in your application according to a template, the operating system can successfully invoke it. For example, Audio Queue Services specifies a template for a callback—that you can implement—that lets you get and react to messages when an audio queue object property changes. This callback template, declared in the `AudioQueue.h` header file, is shown in Listing 2-1.

Listing 2-1 A template for a callback function

```
typedef void    (*AudioQueuePropertyListenerProc) (
                    void *                  inUserData,
                    AudioQueueRef           inAQ,
                    AudioQueuePropertyID    inID
                );
```

To implement and use a callback in your application, you do two things:

1. Implement the callback function. For example, you might implement the audio queue property listener callback to update the titles and the enabled/disabled state of buttons in your user interface, depending on whether an audio queue object is running or stopped.
2. Register your callback function with the object you want to interact with. One way to register a callback—typically used for callbacks that send or receive audio data—is during object creation: In the function call that creates the object, you pass a reference to your callback as a function parameter. The other way—typically used for property listeners—is by using a dedicated function call, as you will see later in this section.

Listing 2-2 shows one way you might implement a property listener callback function to respond to property changes in a playback audio queue object.

Listing 2-2 A property listener callback implementation

```
static void propertyListenerCallback (
    void                    *inUserData,
    AudioQueueRef           queueObject,
    AudioQueuePropertyID    propertyID
) {
    AudioPlayer *player = (AudioPlayer *) inUserData;
        // gets a reference to the playback object
    [player.notificationDelegate updateUserInterfaceOnAudioQueueStateChange:
player];
        // your notificationDelegate class implements the UI update method
}
```

(In an Objective-C class definition, you place this callback definition outside the class implementation. This is why, in the body of the function, there is a statement to get a reference to your playback object—in this example, the `inUserData` parameter is your object. You make this reference available for use in the callback when you register the callback, described next.)

You would register this particular callback as shown in Listing 2-3.

Listing 2-3 Registering a property listener callback

```
AudioQueueAddPropertyListener (
    self.queueObject,               // the object that will invoke your callback
```

```
    kAudioQueueProperty_IsRunning,   // the ID of the property you want to listen
 for
    propertyListenerCallback,        // a reference to your callback function
    self
);
```

Audio Data Formats

Core Audio insulates you from needing detailed knowledge of audio data formats. Not only does this make it easier to deal with a specific format in your code—it means that one set of code can work with any format supported by the operating system.

> **Note:** An **audio data format** describes audio data per se, including such things as sample rate, bit depth, and packetization. An **audio file format** describes how audio data, audio metadata, and filesystem metadata for a sound file are arranged on disk. Some audio file formats can contain just one sort of audio data format (for instance, an MP3 file can contain only MP3 audio data). Other file formats—such as Apple's CAF format—can contain a variety of audio data formats.

Universal Data Types in Core Audio

In Core Audio, you use two universal data types to represent any audio data format. These types are the data structures `AudioStreamBasicDescription` (Listing 2-4) and `AudioStreamPacketDescription` (Listing 2-5), both declared in the `CoreAudioTypes.h` header file and described in *Core Audio Data Types Reference*.

Listing 2-4 The `AudioStreamBasicDescription` data type

```
struct AudioStreamBasicDescription {
    Float64 mSampleRate;
    UInt32  mFormatID;
    UInt32  mFormatFlags;
    UInt32  mBytesPerPacket;
    UInt32  mFramesPerPacket;
    UInt32  mBytesPerFrame;
    UInt32  mChannelsPerFrame;
    UInt32  mBitsPerChannel;
    UInt32  mReserved;
```

```
};
typedef struct AudioStreamBasicDescription  AudioStreamBasicDescription;
```

In this structure, the `mReserved` member must always have a value of `0`. Other members may have a value of `0` as well. For example, compressed audio formats use a varying number of bits per sample. For these formats, the value of the `mBitsPerChannel` member is `0`.

> **About the name:** Although this data type has "stream" in its name, you use it in every instance where you need to represent an audio data format in Core Audio—including in non-streamed, standard files. You could think of it as the "audio format basic description" data type. The "stream" in the name refers to the fact that audio formats come into play whenever you need to move (that is, stream) audio data around in hardware or software.
>
> In discussions involving Core Audio, you'll often hear "audio stream basic description" abbreviated to "ASBD," which this document does as well.

The audio stream packet description type comes into play for certain compressed audio data formats, as described in "Audio Data Packets" (page 30).

Listing 2-5 The `AudioStreamPacketDescription` data type

```
struct  AudioStreamPacketDescription {
    SInt64  mStartOffset;
    UInt32  mVariableFramesInPacket;
    UInt32  mDataByteSize;
};
typedef struct AudioStreamPacketDescription AudioStreamPacketDescription;
```

For constant bit-rate audio data (as described in "Audio Data Packets" (page 30)), the `mVariableFramesInPacket` member in this structure has a value of `0`.

Obtaining a Sound File's Data Format

You can fill in the members of an ASBD by hand in your code. When you do, you may not know the correct value for some (or any) of the structure's members. Set those values to 0 and then use Core Audio interfaces to flesh out the structure.

For example, you can use Audio File Services to give you the complete audio stream basic description for a sound file on disk as shown in Listing 2-6.

Listing 2-6 Obtaining an audio stream basic description for playing a sound file

```
- (void) openPlaybackFile: (CFURLRef) soundFile {

    AudioFileOpenURL (
        (CFURLRef) self.audioFileURL,
        0x01,                   // read only
        kAudioFileCAFType,
        &audioFileID
    );

    UInt32 sizeOfPlaybackFormatASBDStruct = sizeof ([self audioFormat]);

    AudioFileGetProperty (
        [self audioFileID],
        kAudioFilePropertyDataFormat,
        &sizeOfPlaybackFormatASBDStruct,
        &audioFormat          // the sound file's ASBD is returned here
    );
}
```

Canonical Audio Data Formats

Depending on the platform, Core Audio has one or two "canonical" audio data formats in the sense that these formats may be:

- Required as an intermediate format in conversions
- The format for which a service in Core Audio is optimized
- A default, or assumed, format, when you do not otherwise specify an ASBD

The canonical formats in Core Audio are as follows:

- **iOS input and output** Linear PCM with 16-bit integer samples
- **iOS audio units and other audio processing** Noninterleaved linear PCM with 8.24-bit fixed-point samples
- **Mac input and output** Linear PCM with 32-bit floating point samples

- **Mac audio units and other audio processing** Noninterleaved linear PCM with 32-bit floating point samples

Here is a fully fledged audio stream basic description example that illustrates a two-channel, canonical iPhone audio unit sample format with a 44.1 kHz sample rate.

```
struct AudioStreamBasicDescription {
    mSampleRate       = 44100.0;
    mFormatID         = kAudioFormatLinearPCM;
    mFormatFlags      = kAudioFormatFlagsAudioUnitCanonical;
    mBytesPerPacket   = 1 * sizeof (AudioUnitSampleType);    // 8
    mFramesPerPacket  = 1;
    mBytesPerFrame    = 1 * sizeof (AudioUnitSampleType);    // 8
    mChannelsPerFrame = 2;
    mBitsPerChannel   = 8 * sizeof (AudioUnitSampleType);    // 32
    mReserved         = 0;
};
```

The constants and data types used as values here are declared in the `CoreAudioTypes.h` header file and described in *Core Audio Data Types Reference*. Using the `AudioUnitSampleType` data type here (and the `AudioSampleType` data type when handling audio I/O) ensures that an ASBD is platform agnostic.

Two more concepts round out this introduction to audio data formats in Core Audio: magic cookies and packets, described next.

Magic Cookies

In the realm of Core Audio, a **magic cookie** is an opaque set of metadata attached to a compressed sound file or stream. The metadata gives a decoder the details it needs to properly decompress the file or stream. You treat a magic cookie as a black box, relying on Core Audio functions to copy, read, and use the contained metadata.

For example, Listing 2-7 shows a method that obtains a magic cookie from a sound file and provides it to a playback audio queue object.

Listing 2-7 Using a magic cookie when playing a sound file

```
- (void) copyMagicCookieToQueue: (AudioQueueRef) queue fromFile: (AudioFileID)
file {
```

```
    UInt32 propertySize = sizeof (UInt32);

    OSStatus result = AudioFileGetPropertyInfo (
                            file,
                            kAudioFilePropertyMagicCookieData,
                            &propertySize,
                            NULL
                        );

    if (!result && propertySize) {

        char *cookie = (char *) malloc (propertySize);

        AudioFileGetProperty (
            file,
            kAudioFilePropertyMagicCookieData,
            &propertySize,
            cookie
        );

        AudioQueueSetProperty (
            queue,
            kAudioQueueProperty_MagicCookie,
            cookie,
            propertySize
        );

        free (cookie);
    }
}
```

Audio Data Packets

The previous chapter defined a packet as a collection of one or more frames. It is the smallest meaningful set of frames for a given audio data format. For this reason, it is the best unit of audio data to represent a unit of time in an audio file. Synchronization in Core Audio works by counting packets. You can use packets to calculate useful audio data buffer sizes, as shown in Listing 2-8 (page 30).

Every audio data format is defined, in part, by the way its packets are configured. The audio stream basic description data structure describes basic information about a format's packets in its `mBytesPerPacket` and `mFramesPerPacket` members, as shown in Listing 2-4 (page 25). For formats that require additional information, you use the audio stream packet description data structure, as described in a moment.

There are three kinds of packets used in audio data formats:

- In CBR (constant bit rate) formats, such as linear PCM and IMA/ADPCM, all packets are the same size.
- In VBR (variable bit rate) formats, such as AAC, Apple Lossless, and MP3, all packets have the same number of frames but the number of bits in each sample value can vary.
- In VFR (variable frame rate) formats, packets have a varying number of frames. There are no commonly used formats of this type.

To use VBR or VFR formats in Core Audio, you use the audio stream packet description structure (Listing 2-5 (page 26)). Each such structure describes a single packet in a sound file. To record or play a VBR or VFR sound file, you need an array of these structures, one for each packet in the file.

Interfaces in Audio File Services and Audio File Stream Services let you work with packets. For example, the `AudioFileReadPackets` function in `AudioFile.h` gives you a set of packets read from a sound file on disk, placing them in a buffer. At the same time, it gives you an array of `AudioStreamPacketDescription` structures that describes each of those packets.

In CBR and VBR formats (that is, in all commonly used formats), the number of packets per second is fixed for a given audio file or stream. There is a useful implication here: the packetization implies a unit of time for the format. You can use packetization when calculating a practical audio data buffer size for your application. For example, the following method determines the number of packets to read to fill a buffer with a given duration of audio data:

Listing 2-8 Calculating playback buffer size based on packetization

```
- (void) calculateSizesFor: (Float64) seconds {

    UInt32 maxPacketSize;
    UInt32 propertySize = sizeof (maxPacketSize);
```

```
    AudioFileGetProperty (
        audioFileID,
        kAudioFilePropertyPacketSizeUpperBound,
        &propertySize,
        &maxPacketSize
    );

    static const int maxBufferSize = 0x10000;   // limit maximum size to 64K
    static const int minBufferSize = 0x4000;    // limit minimum size to 16K

    if (audioFormat.mFramesPerPacket) {
        Float64 numPacketsForTime =
            audioFormat.mSampleRate / audioFormat.mFramesPerPacket * seconds;
        [self setBufferByteSize: numPacketsForTime * maxPacketSize];
    } else {
        // if frames per packet is zero, then the codec doesn't know the
        // relationship between packets and time. Return a default buffer size
        [self setBufferByteSize:
            maxBufferSize > maxPacketSize ? maxBufferSize : maxPacketSize];
    }

    // clamp buffer size to our specified range
    if (bufferByteSize > maxBufferSize && bufferByteSize > maxPacketSize) {
        [self setBufferByteSize: maxBufferSize];
    } else {
        if (bufferByteSize < minBufferSize) {
            [self setBufferByteSize: minBufferSize];
        }
    }

    [self setNumPacketsToRead: self.bufferByteSize / maxPacketSize];
}
```

Data Format Conversion

To convert audio data from one format to another, you use an audio converter. You can make simple conversions such as changing sample rate or interleaving/deinterleaving. You can also perform complex conversions such as compressing or decompressing audio. Three types of conversions are available:

- Decoding an audio format (such as AAC (Advanced Audio Coding)) to linear PCM format.
- Converting linear PCM data into a different audio format.
- Converting between different variants of linear PCM (for example, converting 16-bit signed integer linear PCM to 8.24 fixed-point linear PCM).

When you use Audio Queue Services (described in "Recording and Playback using Audio Queue Services" (page 42)), you get the appropriate converter automatically. Audio Codec Services (Mac only) lets you create specialized audio codecs, such as for handling digital rights management (DRM) or proprietary audio formats. After you create a custom codec, you can use an audio converter to access and use it.

When you use an audio converter explicitly in OS X, you call a conversion function with a particular converter instance, specifying where to find the input data and where to write the output. Most conversions require a callback function to periodically supply input data to the converter. For examples of how to use audio converters, see `SimpleSDK/ConvertFile` and the `AFConvert` command-line tool in `Services/AudioFileTools` in the Core Audio SDK.

"Supported Audio File and Data Formats in OS X" (page 89) lists standard Core Audio codecs for translating in either direction between compressed formats and Linear PCM. For more on codecs, see "Codecs" (page 52).

Sound Files

Whenever you want to work with sound files in your application, you can use Audio File Services, one of the mid-level services in Core Audio. Audio File Services provides a powerful abstraction for accessing audio data and metadata contained in files, and for creating sound files.

Besides letting you work with the basics—unique file ID, file type, and data format—Audio File Services lets you work with regions and markers, looping, playback direction, SMPTE time code, and more.

You also use Audio File Services for discovering system characteristics. The functions you use are `AudioFileGetGlobalInfoSize` (to let you allocate memory to hold the information you want) and `AudioFileGetGlobalInfo` (to get that information). A long list of properties declared in `AudioFile.h` lets you programmatically obtain system characteristics such as:

- Readable file types

- Writable file types
- For each writable type, the audio data formats you can put into the file

This section also introduces you to two other Core Audio technologies:

- Audio File Stream Services, an audio stream parsing interface, lets you read audio data from disk or from a network stream.
- Extended Audio File Services (Mac only) encapsulates features from Audio File Services and Audio Converter Services, simplifying your code.

Creating a New Sound File

To create a new sound file to record into, you need:

- The file system path for the file, in the form of a `CFURL` or `NSURL` object.
- The identifier for the type of file you want to create, as declared in the Audio File Types enumeration in `AudioFile.h`. For example, to create a CAF file, you use the `kAudioFileCAFType` identifier.
- The audio stream basic description for the audio data you will put in the file. In many cases you can provide a partial ASBD and later ask Audio File Services to fill it in for you.

You give these three pieces of information to Audio File Services as parameters to the `AudioFileCreateWithURL` function, which creates the file and gives you back an `AudioFileID` object. You then use this object for further interaction with the sound file. The function call is shown in Listing 2-9 (page 33).

Listing 2-9 Creating a sound file

```
AudioFileCreateWithURL (
    audioFileURL,
    kAudioFileCAFType,
    &audioFormat,
    kAudioFileFlags_EraseFile,
    &audioFileID   // the function provides the new file object here
);
```

Opening a Sound File

To open a sound file for playback, you use the `AudioFileOpenURL` function. You supply this function with the file's URL, a file type hint constant, and the file access permissions you want to use. `AudioFileOpenURL` gives you back a unique ID for the file.

You then use property identifiers, along with the `AudioFileGetPropertyInfo` and `AudioFileGetProperty` functions, to retrieve what you need to know about the file. Some often-used property identifiers—which are fairly self-explanatory—are:

- `kAudioFilePropertyFileFormat`
- `kAudioFilePropertyDataFormat`
- `kAudioFilePropertyMagicCookieData`
- `kAudioFilePropertyChannelLayout`

There are many such identifiers available in Audio File Services that let you obtain metadata that may be in a file, such as region markers, copyright information, and playback tempo.

When a VBR file is long—say, a podcast—obtaining the entire packet table can take a significant amount of time. In such a case, two property identifiers are especially useful: `kAudioFilePropertyPacketSizeUpperBound` and `kAudioFilePropertyEstimatedDuration`. You can use these to quickly approximate a VBR sound file's duration or number of packets, in lieu of parsing the entire file to get exact numbers.

Reading From and Writing To a Sound File

In iOS you typically use Audio File Services to read audio data from, and write it to, sound files. Reading and writing are essentially mirror images of each other when you use Audio File Services. Both operations block until completion, and both can work using bytes or packets. However, unless you have special requirements, always use packets.

- Reading and writing by packet is the only option for VBR data.
- Using packet-based operations makes it much easier to compute durations.

Another option in iOS for reading audio data from disk is Audio File Stream Services. For an introduction to this technology, see "Sound Streams" (page 36) later in this chapter.

Audio Queue Services, declared in the `AudioQueue.h` header file in the Audio Toolbox framework, is the Core Audio interface for recording and playback. For an overview of Audio Queue Services, see "Recording and Playback using Audio Queue Services" (page 42) later in this chapter.

Extended Audio File Services

Core Audio provides a convenience API called Extended Audio File Services. This interface encompasses the essential functions in Audio File Services and Audio Converter Services, providing automatic data conversion to and from linear PCM. See "Reading and Writing Audio Data" (page 61) for more on this.

iPhone Audio File Formats

iOS supports the audio file formats listed in Table 2-1 (page 35). For information on audio data formats available in iOS, see "Codecs" (page 52).

Table 2-1 iOS audio file formats

Format name	Format filename extensions
AIFF	`.aif`, `.aiff`
CAF	`.caf`
MPEG-1, layer 3	`.mp3`
MPEG-2 or MPEG-4 ADTS	`.aac`
MPEG-4	`.m4a`, `.mp4`
WAV	`.wav`

CAF Files

iOS and OS X have a native audio file format, the Core Audio Format (or *CAF*) file format. The CAF format was introduced in OS X v10.4 "Tiger" and is available in iOS 2.0 and newer. It is unique in that it can contain any audio data format supported on a platform.

CAF files have no size restrictions—unlike AIFF and WAVE files—and can support a wide range of metadata, such as channel information and text annotations. For detailed information about the CAF file format, see *Apple Core Audio Format Specification 1.0*.

Sound Streams

Unlike a disk-based sound file, an **audio file stream** is audio data whose beginning and end you may not have access to. You encounter streams, for example, when you build an Internet radio player application. A provider typically sends their stream continuously. When a user presses Play to listen in, your application needs to jump aboard no matter what data is going by at the moment—the start, middle, or end of an audio packet, or perhaps a magic cookie.

Also unlike a sound file, a stream's data may not be reliably available. There may be dropouts, discontinuities, or pauses—depending on the vagaries of the network you get the stream from.

Audio File Stream Services lets your application work with streams and all their complexities. It takes care of the parsing.

To use Audio File Stream Services, you create an audio file stream object, of type `AudioFileStreamID`. This object serves as a proxy for the stream itself. This object also lets your application know what's going on with the stream by way of properties (see "Properties, Scopes, and Elements" (page 22)). For example, when Audio File Stream Services has determined the bit rate for a stream, it sets the `kAudioFileStreamProperty_BitRate` property on your audio file stream object.

Because Audio File Stream Services performs the parsing, it becomes your application's role to respond to being given sets of audio data and other information. You make your application responsive in this way by defining two callback functions.

First, you need a callback for property changes in your audio file stream object. At a minimum, you write this callback to respond to changes in the `kAudioFileStreamProperty_ReadyToProducePackets` property. The typical scenario for using this property is as follows:

1. A user presses Play, or otherwise requests that a stream start playing.
2. Audio File Stream Services starts parsing the stream.
3. When enough audio data packets are parsed to send them along to your application, Audio File Stream Services sets the `kAudioFileStreamProperty_ReadyToProducePackets` property to `true` (actually, to a value of `1`) in your audio file stream object.
4. Audio File Stream Services invokes your application's property listener callback, with a property ID value of `kAudioFileStreamProperty_ReadyToProducePackets`.
5. Your property listener callback takes appropriate action, such as setting up an audio queue object for playback of the stream.

Second, you need a callback for the audio data. Audio File Stream Services calls this callback whenever it has collected a set of complete audio data packets. You define this callback to handle the received audio. Typically, you play it back immediately by sending it along to Audio Queue Services. For more on playback, see the next section, "Recording and Playback using Audio Queue Services" (page 42).

Audio Sessions: Cooperating with Core Audio

In iOS, your application runs on a device that sometimes has more important things to do, such as take a phone call. If your application is playing sound and a call comes in, the iPhone needs to do the right thing.

For a first pass, this "right thing" means meeting user expectations. For a second pass, it also means that the iPhone needs to consider the current state of each running application when resolving competing requests.

An **audio session** is an intermediary between your application and iOS. Each iPhone application has exactly one audio session. You configure it to express your application's audio intentions. To start, you answer some questions about how you want your application to behave:

- How do you want your application to respond to interruptions, such as a phone call?
- Do you intend to mix your application's sounds with those from other running applications, or do you intend to silence them?
- How should your application respond to an audio route change, for example, when a user plugs in or unplugs a headset?

With answers in hand, you employ the audio session interface (declared in `AudioToolbox/AudioServices.h`) to configure your audio session and your application. Table 2-2 describes the three programmatic features provided by this interface.

Table 2-2 Features provided by the audio session interface

Audio session feature	Description
Categories	A category is a key that identifies a set of audio behaviors for your application. By setting a category, you indicate your audio intentions to iOS, such as whether your audio should continue when the screen locks.
Interruptions and route changes	Your audio session posts notifications when your audio is interrupted, when an interruption ends, and when the hardware audio route changes. These notifications let you respond to changes in the larger audio environment—such as an interruption due to in an incoming phone call—gracefully.

Audio session feature	Description
Hardware characteristics	You can query the audio session to discover characteristics of the device your application is running on, such as hardware sample rate, number of hardware channels, and whether audio input is available.

Audio Session Default Behavior

An audio session comes with some default behavior. Specifically:

- When the user moves the Ring/Silent switch to silent, your audio is silenced.
- When the user presses the Sleep/Wake button to lock the screen, or when the Auto-Lock period expires, your audio is silenced.
- When your audio starts, other audio on the device—such as iPod audio that was already playing—is silenced.

This set of behaviors is specified by the default audio session category, namely `kAudioSessionCategory_SoloAmbientSound`. iOS provides categories for a wide range of audio needs, ranging from user-interface sound effects to simultaneous audio input and output, as you would use for a VOIP (voice over Internet protocol) application. You can specify the category you want at launch and while your application runs.

Audio session default behavior is enough to get you started in iPhone audio development. Except for certain special cases, however, the default behavior is unsuitable for a shipping application, as described next.

Interruptions: Deactivation and Activation

One feature conspicuously absent from a default audio session is the ability to reactivate itself following an interruption. An audio session has two primary states: active and inactive. Audio can work in your application only when the audio session is active.

Upon launch, your default audio session is active. However, if a phone call comes in, your session is immediately deactivated and your audio stops. This is called an *interruption*. If the user chooses to ignore the phone call, your application continues running. But with your audio session inactive, audio does not work.

If you use OpenAL, the I/O audio unit, or Audio Queue Services for audio in your application, you must write an interruption listener callback function and explicitly reactivate your audio session when an interruption ends. *Audio Session Programming Guide* provides details and code examples.

If you use the `AVAudioPlayer` class, the class takes care of audio session reactivation for you.

Determining if Audio Input is Available

A recording application on an iOS-based device can record only if hardware audio input is available. To test this, you use the audio session `kAudioSessionProperty_AudioInputAvailable` property. This is important when your application is running on devices like the iPod touch (2nd generation), which gain audio input only when appropriate accessory hardware is attached. Listing 2-10 shows how to perform the test.

Listing 2-10 Determining if a mobile device supports audio recording

```
UInt32 audioInputIsAvailable;
UInt32 propertySize = sizeof (audioInputIsAvailable);

AudioSessionGetProperty (
    kAudioSessionProperty_AudioInputAvailable,
    &propertySize,
    &audioInputIsAvailable // A nonzero value on output means that
                           // audio input is available
);
```

Using Your Audio Session

Your application has just one audio session category at a time—so, at a given time, all of your audio obeys the characteristics of the active category. (The one exception to this rule is audio that you play using System Sound Services—the API for alerts and user-interface sound effects. Such audio always uses the lowest priority audio session category.) "Audio Session Categories" in *Audio Session Programming Guide* describes all the categories.

When you add audio session support to your application, you can still run your app in the Simulator for development and testing. However, the Simulator does not simulate session behavior. To test the behavior of your audio session code, you need to run on a device.

Note: Ignoring Audio Session Services will not prevent your application from running, but your app may not behave the way you want it to. In most cases, you should not ship an iPhone or iPod touch application that uses audio without using this interface, as described earlier in this section.

Playback using the AVAudioPlayer Class

The `AVAudioPlayer` class provides a simple Objective-C interface for audio playback. If your application does not require stereo positioning or precise synchronization, and if you are not playing audio captured from a network stream, Apple recommends that you use this class for playback.

Using an audio player you can:

- Play sounds of any duration
- Play sounds from files or memory buffers
- Loop sounds
- Play multiple sounds simultaneously
- Control relative playback level for each sound you are playing
- Seek to a particular point in a sound file, which supports such application features as fast forward and rewind
- Obtain data that you can use for audio level metering

The `AVAudioPlayer` class lets you play sound in any audio format available in iOS. For a complete description of this class's interface, see *AVAudioPlayer Class Reference*.

Unlike OpenAL, the I/O audio unit, and Audio Queue Services, the `AVAudioPlayer` class does not require that you use Audio Session Services. An audio player reactivates itself following an interruption. If you want the behavior specified by the default audio session category (see "Audio Sessions: Cooperating with Core Audio" (page 37)), such as having your audio stop when the screen locks, you can successfully use the default audio session with an audio player.

To configure an audio player for playback, you assign a sound file to it, prepare it to play, and designate a delegate object. The code in Listing 2-11 would typically go into an initialization method of the controller class for your application.

Listing 2-11 Configuring an AVAudioPlayer object

```
NSString *soundFilePath =
```

```
                [[NSBundle mainBundle] pathForResource: @"sound"
                                                ofType: @"wav"];

NSURL *fileURL = [[NSURL alloc] initFileURLWithPath: soundFilePath];

AVAudioPlayer *newPlayer =
                [[AVAudioPlayer alloc] initWithContentsOfURL: fileURL
                                                       error: nil];
[fileURL release];

self.player = newPlayer;
[newPlayer release];

[self.player prepareToPlay];
[self.player setDelegate: self];
```

You use a delegate object (which can be your controller object) to handle interruptions and to update the user interface when a sound has finished playing. The delegate methods for the `AVAudioPlayer` class are described in *AVAudioPlayerDelegate Protocol Reference* . Listing 2-12 shows a simple implementation of one delegate method. This code updates the title of a Play/Pause toggle button when a sound has finished playing.

Listing 2-12 Implementing an AVAudioPlayer delegate method

```
- (void) audioPlayerDidFinishPlaying: (AVAudioPlayer *) player
                        successfully: (BOOL) flag {
    if (flag == YES) {
        [self.button setTitle: @"Play" forState: UIControlStateNormal];
    }
}
```

To play, pause, or stop an `AVAudioPlayer` object, call one of its playback control methods. You can test whether or not playback is in progress by using the `playing` property. Listing 2-13 shows a basic play/pause toggle method that controls playback and updates the title of a `UIButton` object.

Listing 2-13 Controlling an AVAudioPlayer object

```
- (IBAction) playOrPause: (id) sender {
```

```
    // if already playing, then pause
    if (self.player.playing) {
        [self.button setTitle: @"Play" forState: UIControlStateHighlighted];
        [self.button setTitle: @"Play" forState: UIControlStateNormal];
        [self.player pause];

    // if stopped or paused, start playing
    } else {
        [self.button setTitle: @"Pause" forState: UIControlStateHighlighted];
        [self.button setTitle: @"Pause" forState: UIControlStateNormal];
        [self.player play];
    }
}
```

The `AVAudioPlayer` class uses the Objective-C declared properties feature for managing information about a sound—such as the playback point within the sound's timeline, and for accessing playback options—such as volume and looping. For example, you set the playback volume for an audio player as shown here:

```
[self.player setVolume: 1.0];    // available range is 0.0 through 1.0
```

For more information on the `AVAudioPlayer` class, see *AVAudioPlayer Class Reference*.

Recording and Playback using Audio Queue Services

Audio Queue Services provides a straightforward, low overhead way to record and play audio. It lets your application use hardware recording and playback devices (such as microphones and loudspeakers) without knowledge of the hardware interface. It also lets you use sophisticated codecs without knowledge of how the codecs work.

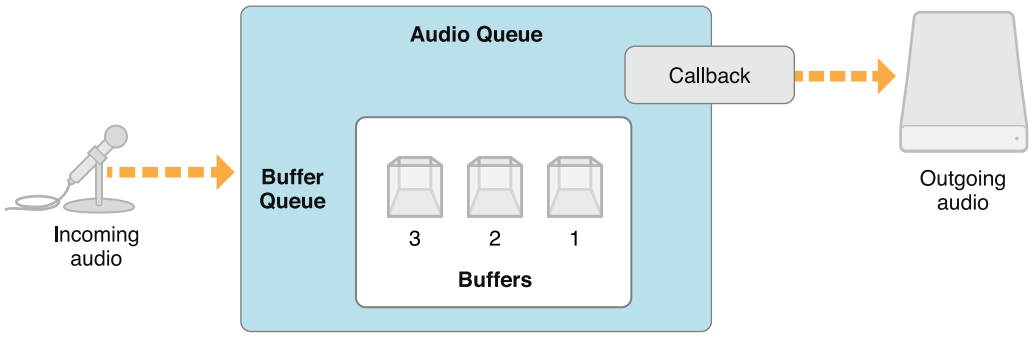
Although it is a high-level interface, Audio Queue Services supports some advanced features. It provides fine-grained timing control to support scheduled playback and synchronization. You can use it to synchronize playback of multiple audio queues, to play sounds simultaneously, to independently control playback level of multiple sounds, and to perform looping. Audio Queue Services and the `AVAudioPlayer` class (see "Playback using the AVAudioPlayer Class" (page 40)) are the only ways to play compressed audio on an iPhone or iPod touch.

You typically use Audio Queue Services in conjunction with Audio File Services (as described in "Sound Files" (page 32)) or with Audio File Stream Services (as described in "Sound Streams" (page 36)).

Audio Queue Callback Functions for Recording and Playback

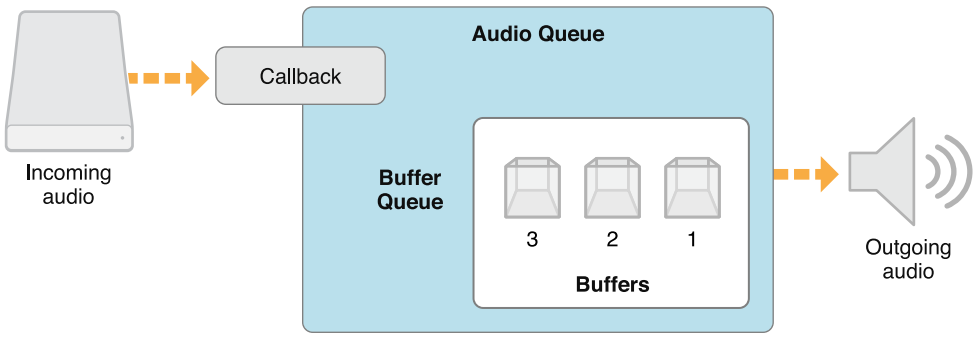
As is the case with Audio File Stream Services, you interact with audio queue objects using callbacks and properties. For recording, you implement a callback function that accepts audio data buffers—provided by your recording audio queue object—and writes them to disk. Your audio queue object invokes your callback when there is a new buffer of incoming data to record. Figure 2-2 illustrates a simplified view of this process.

Figure 2-2 Recording with an audio queue object



For playback, your audio callback has a converse role. It gets called when your playback audio queue object needs another buffer's worth of audio to play. Your callback then reads a given number of audio packets from disk and hands them off to one of the audio queue object's buffers. The audio queue object plays that buffer in turn. Figure 2-3 illustrates this.

Figure 2-3 Playing back using an audio queue object



Creating an Audio Queue Object

To use Audio Queue Services, you first create an audio queue object—of which there are two flavors, although both have the data type `AudioQueueRef`.

- You create a recording audio queue object using the `AudioQueueNewInput` function.
- You create a playback audio queue object using the `AudioQueueNewOutput` function.

To create an audio queue object for playback, perform these three steps:

1. Create a data structure to manage information needed by the audio queue, such as the audio format for the data you want to play.
2. Define a callback function for managing audio queue buffers. The callback uses Audio File Services to read the file you want to play.
3. Instantiate the playback audio queue using the `AudioQueueNewOutput` function.

Listing 2-14 illustrates these steps:

Listing 2-14 Creating an audio queue object

```
static const int kNumberBuffers = 3;
// Create a data structure to manage information needed by the audio queue
struct myAQStruct {
    AudioFileID                   mAudioFile;
    CAStreamBasicDescription      mDataFormat;
    AudioQueueRef                 mQueue;
    AudioQueueBufferRef           mBuffers[kNumberBuffers];
    SInt64                        mCurrentPacket;
    UInt32                        mNumPacketsToRead;
    AudioStreamPacketDescription  *mPacketDescs;
    bool                          mDone;
};
// Define a playback audio queue callback function
static void AQTestBufferCallback(
    void                    *inUserData,
    AudioQueueRef           inAQ,
    AudioQueueBufferRef     inCompleteAQBuffer
) {
    myAQStruct *myInfo = (myAQStruct *)inUserData;
    if (myInfo->mDone) return;
    UInt32 numBytes;
```

```
    UInt32 nPackets = myInfo->mNumPacketsToRead;

    AudioFileReadPackets (
        myInfo->mAudioFile,
        false,
        &numBytes,
        myInfo->mPacketDescs,
        myInfo->mCurrentPacket,
        &nPackets,
        inCompleteAQBuffer->mAudioData
    );
    if (nPackets > 0) {
        inCompleteAQBuffer->mAudioDataByteSize = numBytes;
        AudioQueueEnqueueBuffer (
            inAQ,
            inCompleteAQBuffer,
            (myInfo->mPacketDescs ? nPackets : 0),
            myInfo->mPacketDescs
        );
        myInfo->mCurrentPacket += nPackets;
    } else {
        AudioQueueStop (
            myInfo->mQueue,
            false
        );
        myInfo->mDone = true;
    }
}
// Instantiate an audio queue object
AudioQueueNewOutput (
    &myInfo.mDataFormat,
    AQTestBufferCallback,
    &myInfo,
    CFRunLoopGetCurrent(),
    kCFRunLoopCommonModes,
```

```
    0,
    &myInfo.mQueue
);
```

Controlling Audio Queue Playback Level

Audio queue objects give you two ways to control playback level.

To set playback level directly, use the `AudioQueueSetParameter` function with the `kAudioQueueParam_Volume` parameter, as shown in Listing 2-15. Level change takes effect immediately.

Listing 2-15 Setting playback level directly

```
Float32 volume = 1;
AudioQueueSetParameter (
    myAQstruct.audioQueueObject,
    kAudioQueueParam_Volume,
    volume
);
```

You can also set playback level for an audio queue buffer, using the `AudioQueueEnqueueBufferWithParameters` function. This lets you assign audio queue settings that are, in effect, carried by an audio queue buffer as you enqueue it. Such changes take effect when the audio queue buffer begins playing.

In both cases, level changes for an audio queue remain in effect until you change them again.

Indicating Audio Queue Playback Level

You can obtain the current playback level from an audio queue object by querying its `kAudioQueueProperty_CurrentLevelMeterDB` property. The value of this property is an array of `AudioQueueLevelMeterState` structures, one per channel. Listing 2-16 shows this structure:

Listing 2-16 The `AudioQueueLevelMeterState` structure

```
typedef struct AudioQueueLevelMeterState {
    Float32     mAveragePower;
    Float32     mPeakPower;
```

```
};  AudioQueueLevelMeterState;
```

Playing Multiple Sounds Simultaneously

To play multiple sounds simultaneously, create one playback audio queue object for each sound. For each audio queue, schedule the first buffer of audio to start at the same time using the `AudioQueueEnqueueBufferWithParameters` function.

Audio format is critical when you play sounds simultaneously on iPhone or iPod touch. This is because playback of certain compressed formats in iOS employs an efficient hardware codec. Only a single instance of one of the following formats can play on the device at a time:

- AAC
- ALAC (Apple Lossless)
- MP3

To play high quality, simultaneous sounds, use linear PCM or IMA4 audio.

The following list describes how iOS supports audio formats for individual or multiple playback:

- **Linear PCM and IMA/ADPCM (IMA4) audio** You can play multiple linear PCM or IMA4 format sounds simultaneously in iOS without incurring CPU resource problems.
- **AAC, MP3, and Apple Lossless (ALAC) audio** Playback for AAC, MP3, and Apple Lossless (ALAC) sounds uses efficient hardware-based decoding on iPhone and iPod touch. You can play only one such sound at a time.

For a complete description of how to use Audio Queue Services for recording and playback, including code samples, see *Audio Queue Services Programming Guide*.

Playback with Positioning Using OpenAL

The open-sourced OpenAL audio API—available in the OpenAL framework and built on top of Core Audio—is optimized for positioning sounds during playback. OpenAL makes it easy to play, position, mix, and move sounds, using an interface modeled after OpenGL. OpenAL and OpenGL share a common coordinate system, enabling you to synchronize the movements of sounds and on-screen graphical objects. OpenAL uses Core Audio's I/O audio unit (see "Audio Units" (page 50)) directly, resulting in lowest-latency playback.

For all of these reasons, OpenAL is your best choice for playing sound effects in game applications on iPhone and iPod touch. OpenAL is also a good choice for general iOS application playback needs.

OpenAL 1.1 in iOS does not support audio capture. For recording in iOS, use Audio Queue Services.

OpenAL in OS X provides an implementation of the OpenAL 1.1. specification, plus extensions. OpenAL in iOS has two Apple extensions:

- `alBufferDataStaticProcPtr` follows the use pattern for `alBufferData` but eliminates a buffer data copy.
- `alcMacOSXMixerOutputRateProcPtr` lets you control the sample rate of the underlying mixer.

For a Mac example of using OpenAL in Core Audio, see `Services/OpenALExample` in the Core Audio SDK. For OpenAL documentation, see the OpenAL website at http://openal.org.

System Sounds: Alerts and Sound Effects

To play short sound files when you do not need level, positioning, audio session, or other control, use System Sound Services, declared in the `AudioServices.h` header file in the Audio Toolbox framework. This interface is ideal when you just want to play a short sound in the simplest way possible. In iOS, sounds played using System Sound Services are limited to a maximum duration of 30 seconds.

In iOS, you call the `AudioServicesPlaySystemSound` function to immediately play a sound effect file that you designate. Alternatively, you can call `AudioServicesPlayAlertSound` when you need to alert the user. Each of these functions will invoke vibration on an iPhone if the user has set the silence switch. (Using the iOS Simulator or on the desktop, of course, there is no vibration or buzzing.)

You can explicitly invoke vibration on an iPhone by using the `kSystemSoundID_Vibrate` constant when you call the `AudioServicesPlaySystemSound` function.

To play a sound with the `AudioServicesPlaySystemSound` function, you first register your sound effect file as a system sound by creating a sound ID object. You can then play the sound. In typical use, which includes playing a sound occasionally or repeatedly, retain the sound ID object until your application quits. If you know that you will use a sound only once—for example, in the case of a startup sound—you can destroy the sound ID object immediately after playing the sound, freeing memory.

Listing 2-17 shows a minimal program that uses the interfaces in System Sound Services to play a sound. The sound completion callback, and the call that installs it, are primarily useful when you want to free memory after playing a sound in cases where you know you will not be playing the sound again.

Listing 2-17 Playing a short sound

```
#include <AudioToolbox/AudioToolbox.h>
#include <CoreFoundation/CoreFoundation.h>
```

```
// Define a callback to be called when the sound is finished
// playing. Useful when you need to free memory after playing.
static void MyCompletionCallback (
    SystemSoundID  mySSID,
    void * myURLRef
) {
        AudioServicesDisposeSystemSoundID (mySSID);
        CFRelease (myURLRef);
        CFRunLoopStop (CFRunLoopGetCurrent());
}

int main (int argc, const char * argv[]) {
    // Set up the pieces needed to play a sound.
    SystemSoundID    mySSID;
    CFURLRef        myURLRef;
    myURLRef = CFURLCreateWithFileSystemPath (
        kCFAllocatorDefault,
        CFSTR ("../../ComedyHorns.aif"),
        kCFURLPOSIXPathStyle,
        FALSE
    );

    // create a system sound ID to represent the sound file
    OSStatus error = AudioServicesCreateSystemSoundID (myURLRef, &mySSID);

    // Register the sound completion callback.
    // Again, useful when you need to free memory after playing.
    AudioServicesAddSystemSoundCompletion (
        mySSID,
        NULL,
        NULL,
        MyCompletionCallback,
        (void *) myURLRef
    );
```

```
    // Play the sound file.
    AudioServicesPlaySystemSound (mySSID);

    // Invoke a run loop on the current thread to keep the application
    // running long enough for the sound to play; the sound completion
    // callback later stops this run loop.
    CFRunLoopRun ();
    return 0;
}
```

Core Audio Plug-ins: Audio Units and Codecs

Core Audio has a plug-in mechanism for processing audio data. In iOS, the system supplies these plug-ins. In OS X, you have built-in plug-ins and can also create your own.

Multiple host application can each use multiple instances of an audio unit or codec.

Audio Units

In iOS, audio units provide the mechanism for applications to achieve low-latency input and output. They also provide certain DSP features, as described here.

In OS X, audio units are most prominent as add-ons to audio applications like GarageBand and Logic Studio. In this role, an audio unit has its own user interface, or **view**. Audio units in iOS do not have views.

On either platform, you find the programmatic names of the built-in audio units in the `AUComponent.h` header file in the Audio Unit framework.

iOS audio units use 8.24-bit fixed point linear PCM audio data for input and output. The one exception is data format converter units, as described in the following list. The iOS audio units are:

- **3D mixer unit**—Allows any number of mono inputs, each of which can be 8-bit or 16-bit linear PCM. Provides one stereo output in 8.24-bit fixed-point PCM. The 3D mixer unit performs sample rate conversion on its inputs and provides a great deal of control over each input channel. This control includes volume, muting, panning, distance attenuation, and rate control for these changes. Programmatically, this is the `kAudioUnitSubType_AU3DMixerEmbedded` unit.

- **Multichannel mixer unit**—Allows any number of mono or stereo inputs, each of which can be 16-bit linear or 8.24-bit fixed-point PCM. Provides one stereo output in 8.24-bit fixed-point PCM. Your application can mute and unmute each input channel as well as control its volume. Programmatically, this is the `kAudioUnitSubType_MultiChannelMixer` unit.
- **Converter unit**—Provides sample rate, bit depth, and bit format (linear to fixed-point) conversions. The iPhone converter unit's canonical data format is 8.24-bit fixed-point PCM. It converts to or from this format. Programmatically, this is the `kAudioUnitSubType_AUConverter` unit.
- **I/O unit**—Provides real-time audio input and output and performs sample rate conversion as needed. Programmatically, this is the `kAudioUnitSubType_RemoteIO` unit.
- **iPod EQ unit**—Provides a simple equalizer you can use in your application, with the same presets available in the built-in iPhone iPod application. The iPod EQ unit uses 8.24-bit fixed-point PCM input and output. Programmatically, this is the `kAudioUnitSubType_AUiPodEQ` unit.

The special category of audio unit known as the *I/O unit* is described further in "Audio Processing Graphs" (page 53), because of the important role that such audio units play there.

OS X provides more than 40 audio units, all of which you can find listed in "System-Supplied Audio Units in OS X" (page 85). You can also create your own audio units, either as building blocks for your application or to provide to customers as products in their own right. Refer to *Audio Unit Programming Guide* for more information.

OS X audio units use noninterleaved 32-bit floating point linear PCM data for input and output—except in the case of an audio unit that is a data format converter, which converts to or from this format. OS X audio units may also support other linear PCM variants—but no formats other than linear PCM. This ensures compatibility with audio units from Apple and other vendors. To convert audio data of a different format to linear PCM, you can use an audio converter (see "Data Format Conversion" (page 32)).

In iOS and in OS X, host applications use functions in Audio Unit Services to discover and load audio units. Each audio unit is uniquely identified by a three-element combination of type, subtype, and manufacturer code. The type code, specified by Apple, indicates the general purpose of the unit (effect, generator, and so on). The subtype describes more precisely what an audio unit does, but is not programmatically significant for audio units. If your company provides more than one effect unit, however, each must have a distinct subtype to distinguish it from the others. The manufacturer code identifies you as the developer of your audio units. Apple expects you to register a manufacturer code, as a "creator code," on the Data Type Registration page.

Audio units describe their capabilities and configuration information using properties, as described in "Properties, Scopes, and Elements" (page 22). Each audio unit type has some required properties, as defined by Apple, but you are free to add additional properties based on your audio unit's needs. In OS X, host applications can use property information to create a generic view for an audio unit, and you can provide a custom view as an integral piece of your audio unit.

Audio units also use a **parameter** mechanism for settings that are adjustable in real time, often by the end user. These parameters are not arguments to functions, but rather a mechanism that supports user adjustments to audio unit behavior. For example, a parametric filter unit may have parameters for center frequency and width of filter response, settable through a view.

To monitor changes in the state of an audio unit in OS X, applications can register callback functions (sometimes called *listeners*) that are invoked when particular audio unit events occur. For example, a Mac app might want to know when a user moves a slider in an audio unit's view, or when audio data flow is interrupted. See Technical Note TN2104: Handling Audio Unit Events for more details.

The Core Audio SDK (in its `AudioUnits` folder) provides templates for common audio unit types (for example, effect units and instrument units) along with a C++ framework that implements most of the Component Manager plug-in interface for you. For detailed information about building audio units for OS X using the SDK, see *Audio Unit Programming Guide*.

Codecs

The recording and playback codecs available in iOS are chosen to balance audio quality, flexibility in application development, hardware features, and battery life.

There are two groups of playback codecs in iOS. The first group, listed in Table 2-3, includes highly efficient formats that you can use without restriction. That is, you can play more than one instance of each of these formats at the same time.

For information on audio file formats supported in iOS, see "iPhone Audio File Formats" (page 35).

Table 2-3 iOS: unrestricted playback audio formats

iLBC (internet Low Bitrate Codec, also a speech codec)
IMA/ADPCM (also known as IMA-4)
Linear PCM
µLaw and aLaw

The second group of iOS playback codecs all share a single hardware path. Consequently, only a single instance of one of these formats can play at a time:

Table 2-4 iOS: restricted playback audio formats

AAC

Apple Lossless
MP3

iOS contains the recording codecs listed in Table 2-5. As you can see, neither MP3 nor AAC recording is available. This is due to the high CPU overhead, and consequent battery drain, of these formats.

Table 2-5 iOS: recording audio formats

Apple Lossless
iLBC (internet Low Bitrate Codec, a speech codec)
IMA/ADPCM (also known as IMA-4)
Linear PCM
µLaw and aLaw

In OS X, you find a wide array of codecs and supported formats, described in "Supported Audio File and Data Formats in OS X" (page 89).

OS X also provides interfaces for using and creating audio data codecs. These interfaces are declared in the `AudioConverter.h` header file (in the Audio Toolbox framework) and `AudioCodec.h` (in the Audio Unit framework). "Common Tasks in OS X" (page 61) describes how you might use these services.

Audio Processing Graphs

An **audio processing graph** (sometimes called an *AUGraph*) defines a collection of audio units strung together to perform a complex task. For example, a graph could distort a signal, compress it, and then pan it to a particular location in the soundstage. When you define a graph, you have a reusable processing module that you can add to and remove from a signal chain in your application.

A processing graph typically ends in an I/O unit (sometimes called an *output unit*). An I/O unit often interfaces (indirectly—via low-level services) with hardware, but this is not a requirement. An I/O unit can send its output, instead, back to your application.

You may also see an I/O unit referred to as the **head node** in a processing graph. I/O units are the only ones that can start and stop data flow in a graph. This is an essential aspect of the so-called **pull** mechanism that audio units use to obtain data. Each audio unit in a graph registers a rendering callback with its successor,

allowing that successor to request audio data. When an I/O unit starts the data flow (triggered, in turn, by your application), its render method calls back to the preceding audio unit in the chain to ask for data, which in turn calls its predecessor, and so on.

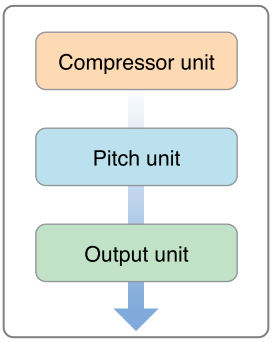
iOS has a single I/O unit, as described in "Audio Units" (page 50). The story on OS X is a bit more complex.

- The **AUHAL unit** is for dedicated input and output to a specific hardware device. See Technical Note TN2091, *Device input using the HAL Output Audio Unit*.
- The **Generic I/O unit** lets you connect the output of an audio processing graph to your application. You can also use a generic I/O unit as the head node of a subgraph, as you can see in Figure 2-6 (page 56).
- The **System I/O unit** is for alerts and user interface sound effects.
- The **Default I/O unit** is for all other audio input and output.

The Audio MIDI Setup application (Mac only; in the Utilities folder) lets a user direct the connections of the System I/O and Default I/O units separately.

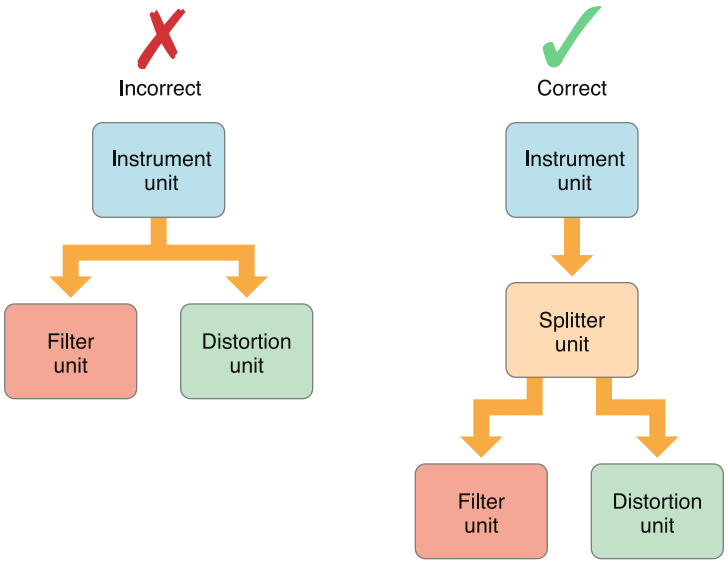
Figure 2-4 shows a simple audio processing graph, in which signal flows from the top to the bottom.

Figure 2-4 A simple audio processing graph



Each audio unit in an audio processing graph can be called a **node**. You make a processing graph connection by attaching an output from one node to the input of another. You cannot connect a single output to more than one input unless you use an intervening splitter unit, as shown in Figure 2-5.

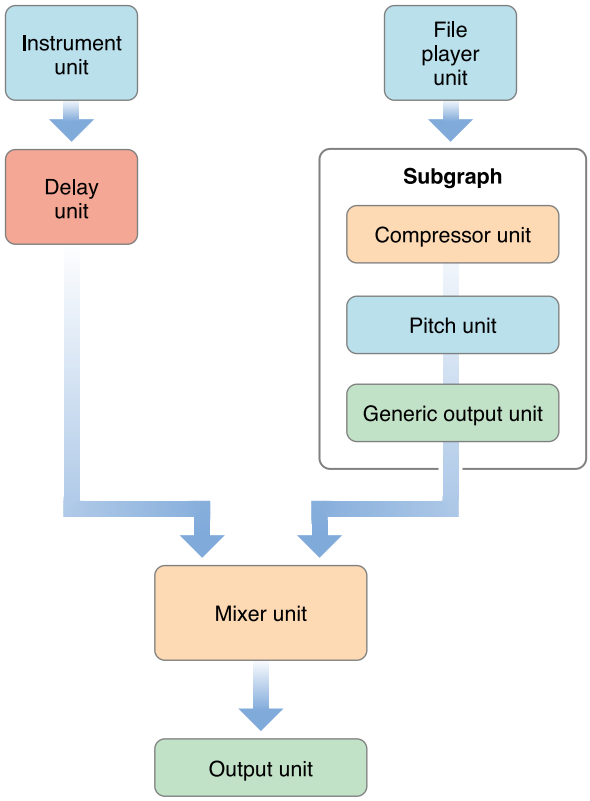
Figure 2-5 How to fan out an audio unit connection



However, an audio unit may be designed to provide multiple outputs or inputs, depending on its type. A splitter unit does this, for example.

You can use Audio Processing Graph Services to combine subgraphs into a larger graph, where a subgraph appears as a single node in the larger graph. Figure 2-6 illustrates this.

Figure 2-6 A subgraph connection



Each graph or subgraph must end in an I/O unit. A subgraph, or a graph whose output feeds your application, should end with the generic I/O unit, which does not connect to hardware.

While you can link audio units programmatically without using an audio processing graph, it's rarely a good idea. Graphs offer important advantages. You can modify a graph dynamically, allowing you to change the signal path while processing data. In addition, because a graph encapsulates an interconnection of audio units, you instantiate a graph in one step instead of explicitly instantiating each of the audio units it references.

MIDI Services in OS X

Core Audio uses Core MIDI Services for MIDI support. These services consist of the functions, data types, and constants declared in the following header files in `CoreMIDI.framework`:

- `MIDIServices.h`
- `MIDISetup.h`

- `MIDIThruConnection.h`
- `MIDIDriver.h`

Core MIDI Services defines an interface that applications and audio units can use to communicate with MIDI devices. It uses a number of abstractions that allow an application to interact with a MIDI network.

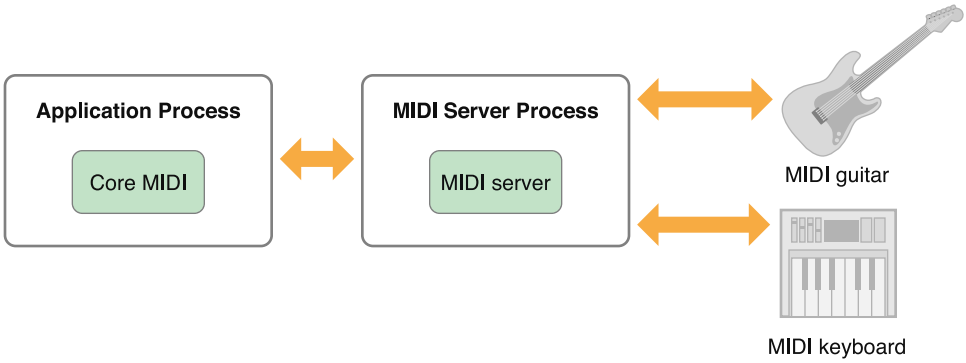
A **MIDI endpoint** (defined by an opaque type `MIDIEndpointRef`) represents a source or destination for a standard 16-channel MIDI data stream. You can associate endpoints with tracks used by Music Player Services, allowing you to record or play back MIDI data. A MIDI endpoint is a proxy object for a standard MIDI cable connection. MIDI endpoints do not necessarily have to correspond to a physical device, however; an application can set itself up as a virtual source or destination to send or receive MIDI data.

MIDI drivers often combine multiple endpoints into logical groups, called **MIDI entities** (`MIDIEntityRef`). For example, it would be reasonable to group a MIDI-in endpoint and a MIDI-out endpoint as a MIDI entity, which can then be easily referenced for bidirectional communication with a device or application.

Each physical MIDI device (not a single MIDI connection) is represented by a Core MIDI device object (`MIDIDeviceRef`). Each device object may contain one or more MIDI entities.

Core MIDI communicates with the MIDI Server, which does the actual job of passing MIDI data between applications and devices. The MIDI Server runs in its own process, independent of any application. Figure 2-7 shows the relationship between Core MIDI and MIDI Server.

Figure 2-7 Core MIDI and Core MIDI Server

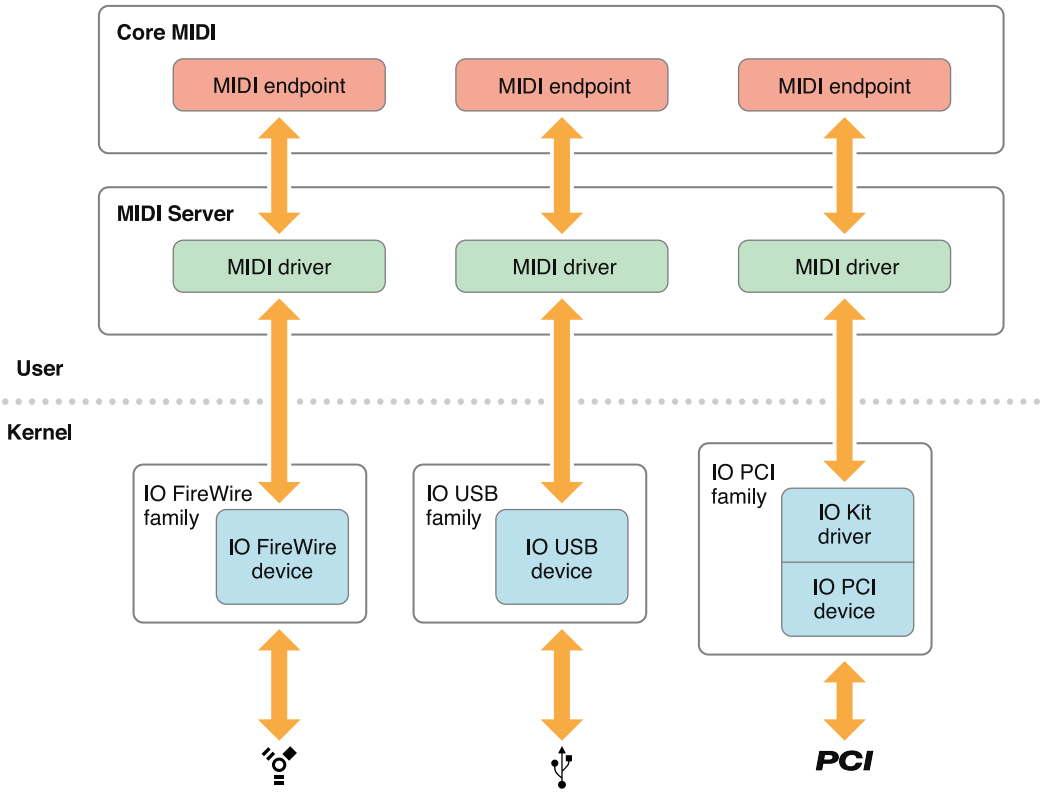


In addition to providing an application-agnostic base for MIDI communications, MIDI Server also handles any MIDI thru connections, which allows device-to device chaining without involving the host application.

If you are a MIDI device manufacturer, you may need to supply a CFPlugin plug-in for the MIDI Server packaged in a CFBundle to interact with the kernel-level I/O Kit drivers. Figure 2-8 shows how Core MIDI and Core MIDI Server interact with the underlying hardware.

Note: If you create a USB MIDI class-compliant device, you do not have to write your own driver, because Apple's supplied USB driver will support your hardware.

Figure 2-8 MIDI Server interface with I/O Kit



The drivers for each MIDI device generally exist outside the kernel, running in the MIDI Server process. These drivers interact with the default I/O Kit drivers for the underlying protocols (such as USB and FireWire). The MIDI drivers are responsible for presenting the raw device data to Core MIDI in a usable format. Core MIDI then passes the MIDI information to your application through the designated MIDI endpoints, which are the abstract representations of the MIDI ports on the external devices.

MIDI devices on PCI cards, however, cannot be controlled entirely through a user-space driver. For PCI cards, you must create a kernel extension to provide a custom user client. This client must either control the PCI device itself (providing a simple message queue for the user-space driver) or map the address range of the PCI device into the address of the MIDI server when requested to do so by the user-space driver. Doing so allows the user-space driver to control the PCI device directly.

For an example of implementing a user-space MIDI driver, see `MIDI/SampleUSBDriver` in the Core Audio SDK.

Music Player Services in OS X

Music Player Services allows you to arrange and play a collection of MIDI music tracks.

A **track** is a stream of MIDI or event data (represented by the `MusicTrack` data type). Tracks contain a series of time-based events, which can be MIDI data, Core Audio event data, or custom event messages. You can think of a track as sheet music for one instrument.

A **sequence** is a collection of tracks (represented by the `MusicSequence` data type). A sequence always contains an separate tempo track, used for synchronizing all its tracks. You can think of a sequence as a musical score, which collects the sheet music for multiple instruments. Your Mac app can add, delete, or edit tracks in a sequence dynamically.

To play a sequence you assign it to a **music player** object (of type `MusicPlayer`), which acts as the conductor, controlling the playback of the sequence. To produce sound, you send each track to an instrument unit or to an external MIDI device.

Track data does not have to represent musical information. You could instead use a track to implement audio unit automation. For example, a track assigned to a panner unit could control the apparent position of a sound source in the soundstage. Tracks can also contain proprietary user events that trigger an application-defined callback.

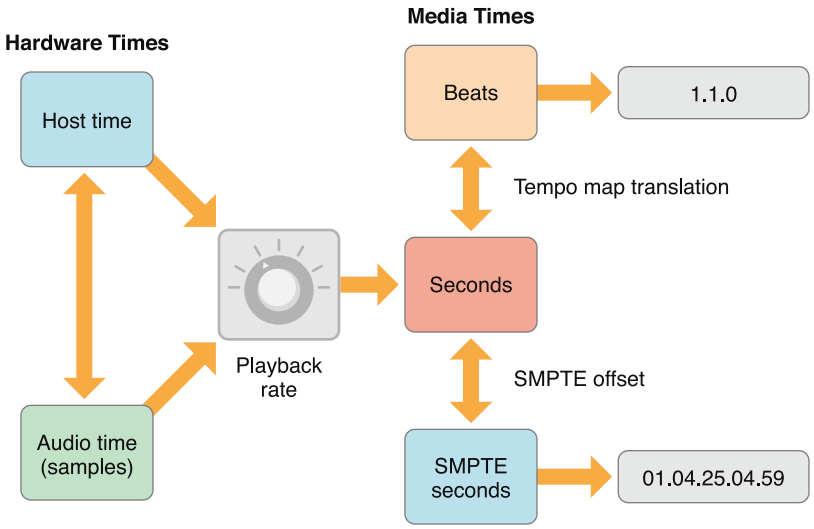
For more information about using Music Player Services to play MIDI data, see "Handling MIDI Data" (page 69).

Timing Services in OS X

Core Audio Clock Services provides a reference clock that you can use to synchronize applications and devices. This clock may be a standalone timing source or it can be synchronized with an external trigger, such as a MIDI beat clock or MIDI time code. You can start and stop the clock yourself, or you can set the clock to activate or deactivate in response to certain events.

You can obtain the generated clock time in a number of formats, including seconds, beats, SMPTE time, audio sample time, and bar-beat time. The latter describes the time in a manner that is easy to display onscreen in terms of musical bars, beats, and subbeats. Core Audio Clock Services also contains utility functions that convert one time format to another and that display bar-beat or SMPTE times. Figure 2-9 shows the interrelationship between various Core Audio Clock formats.

Figure 2-9 Some Core Audio Clock formats



The hardware times represent absolute time values from either the host time (the system clock) or an audio time obtained from an external audio device (represented by an `AudioDevice` object in the HAL). You determine the current host time by calling `mach_absolute_time` or `UpTime`. The audio time is the audio device's current time represented by a sample number. The sample number's rate of change depends on the audio device's sampling rate.

The media times represent common timing methods for audio data. The canonical representation is in seconds, expressed as a double-precision floating point value. However, you can use a tempo map to translate seconds into musical bar-beat time, or apply a SMPTE offset to convert seconds to SMPTE seconds.

Media times do not have to correspond to real time. For example, an audio file that is 10 seconds long will take only 5 seconds to play if you double the playback rate. The knob in "Services Available in iOS Only" (page 82) indicates that you can adjust the correlation between the absolute ("real") times and the media-based times. For example, bar-beat notation indicates the rhythm of a musical line and what notes to play when, but does not indicate how long it takes to play. To determine that, you need to know the playback rate (say, in beats per second). Similarly, the correspondence of SMPTE time to actual time depends on such factors as the frame rate and whether frames are dropped or not.

Common Tasks in OS X

This chapter describes how you can combine parts of Core Audio to accomplish some common tasks in OS X.

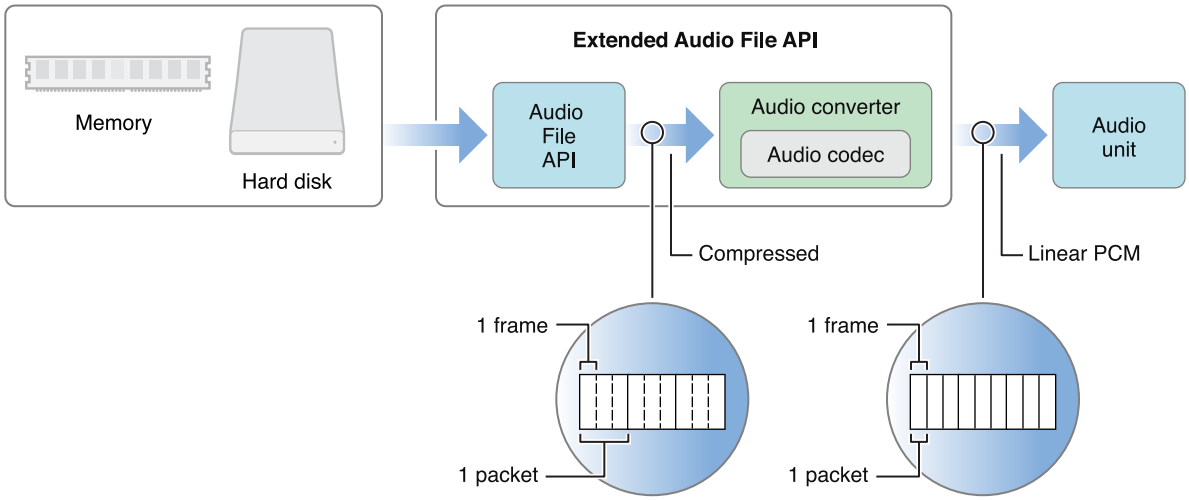
Core Audio is extremely modular, with few restrictions on how to use its various parts. In other words, you can often do things in more than one way. For example, your Mac app can call Audio File Services to read a compressed sound file from disk, Audio Converter Services to convert the data to linear PCM, and Audio Queue Services to play it. Alternatively, you could load the built-in File Player audio unit into your application. Each approach has its advantages and capabilities.

Reading and Writing Audio Data

Many applications that handle audio need to read and write the data, either to disk or to a buffer. You typically want to read the file's contained data and convert it to linear PCM. You can do so in one step using Extended Audio File Services.

As shown in Figure 3-1, Extended Audio File Services uses Audio File Services to read the audio data and then calls Audio Converter Services to convert it to linear PCM (assuming the data is not already in that format).

Figure 3-1 Reading audio data



If you need more control over the file reading and conversion procedure, you can call Audio File or Audio Converter functions directly. You use Audio File Services to read the file from disk or a buffer. This data may be in a compressed format, in which case it can be converted to linear PCM using an audio converter. You can also use an audio converter to handle changes in bit depth, sampling rate, and so on within the linear PCM format. You handle conversions by using Audio Converter Services to create an audio converter object, specifying the input and output formats you desire. Each format is defined in an ASBD (see "Universal Data Types in Core Audio" (page 25)).

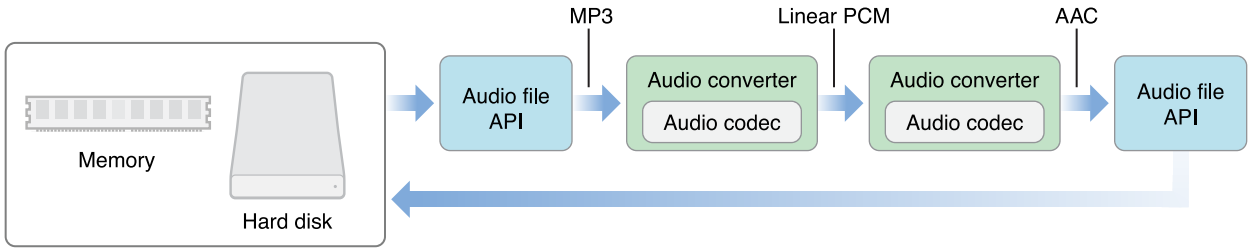
Once converted to linear PCM, the data is ready to process—such as by an audio unit. To use an audio unit, you register a callback with the audio unit's input. You design your callback to provide buffers of PCM data when called. When the audio unit needs more data to process, it invokes your callback.

If you want to output the audio data, you send it to an I/O unit. An I/O unit can accept only linear PCM format data. Such an audio unit is usually a proxy for a hardware device, but this is not a requirement.

Converting Audio Data Formats

Core Audio uses linear PCM as an intermediate format, which permits many permutations of conversions. To determine whether a particular format conversion is possible, you need to make sure that both a decoder (format A to linear PCM) and an encoder (linear PCM to format B) are available. For example, if you wanted to convert data from MP3 to AAC, you would need two audio converters: one to convert from MP3 to linear PCM, and another to convert linear PCM to AAC, as shown in Figure 3-2.

Figure 3-2 Converting audio data using two converters



For examples of using the Audio File and Audio Converter APIs, see the `SimpleSDK/ConvertFile` and `Services/AudioFileTools` samples in the Core Audio SDK. If you are interested in writing a custom audio converter codec, see the samples in the `AudioCodec` folder.

Interfacing with Hardware

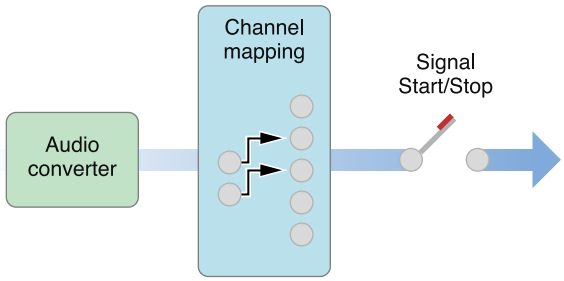
Most audio applications connect with external hardware, either to output sound (for example, to an amplifier and loudspeaker) or to obtain it (such as through a microphone). Core Audio's lower layer makes these connections by talking to the I/O Kit and drivers. OS X provides an interface to these connections, called the *audio hardware abstraction layer*, or audio HAL. You find the audio HAL's interfaces declared in the `AudioHardware.h` header file in the Core Audio framework.

However, in many cases, applications do not need to use the audio HAL directly. Apple provides three standard audio units that should address most hardware needs: the default output unit, the system output unit, and the AUHAL unit. Your application must explicitly load these audio units before you can use them.

Default and System I/O Units

The default I/O unit and system I/O unit send audio data to the default output (as selected by the user) or system output (where alerts and other system sounds are played) respectively. If you connect an audio unit output to one of these I/O units—such as in an audio processing graph—the audio unit's render callback function is called when the output needs data. The I/O unit routes the data through the HAL to the appropriate output device, automatically handling the following tasks, as shown in Figure 3-3.

Figure 3-3 Inside an I/O unit



- Any required linear PCM data conversion. The output unit contains an audio converter that can translate your audio data to the linear PCM variant required by the hardware.
- Any required channel mapping. For example, if your unit is supplying two-channel data but the output device can handle five, you will probably want to map which channels go to which. You can do so by specifying a channel map using the `kAudioOutputUnitProperty_ChannelMap` property on the output unit. If you don't supply a channel map, the default is to map the first audio channel to the first device channel, the second to the second, and so on. The actual output heard is then determined by how the user has configured the device speakers in the Audio MIDI Setup application.
- Signal Start/Stop. Output units are the only audio units that can control the flow of audio data in the signal chain.

For an example of using the default output unit to play audio, see `SimpleSDK/DefaultOutputUnit` in the Core Audio SDK.

The AUHAL Unit

If you need to connect to an input device, or a hardware device other than the default output device, you need to use the AUHAL. Although designated as an output device, you can configure the AUHAL to accept input as well by setting the `kAudioOutputUnitProperty_EnableIO` property on the input. For more specifics, see Technical Note TN2091: Device Input Using the HAL Output Audio Unit. When accepting input, the AUHAL supports input channel mapping and uses an audio converter (if necessary) to translate incoming data to linear PCM format.

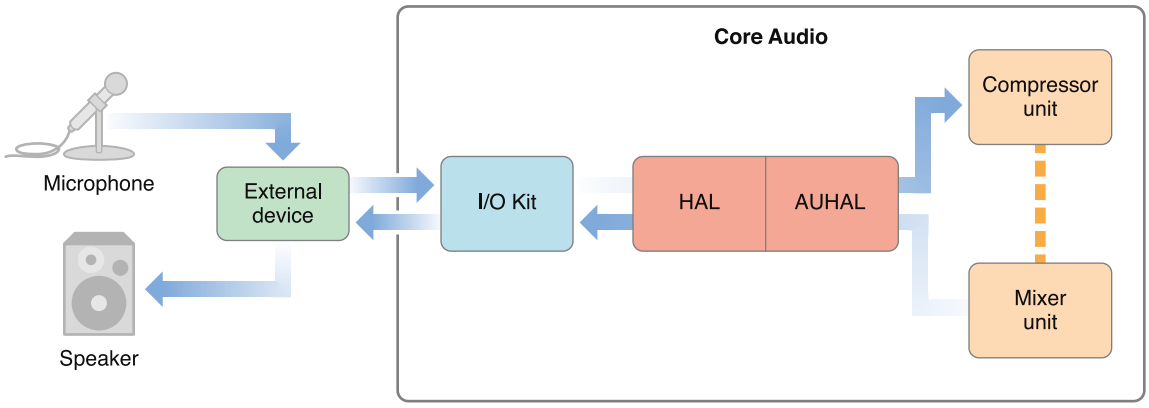
The AUHAL is a more generalized version of the default output unit. In addition to the audio converter and channel mapping capabilities, you can specify the device to connect to by setting the `kAudioOutputUnitProperty_CurrentDevice` property to the ID of an `AudioDevice` object in the HAL. Once connected, you can also manipulate properties associated with the `AudioDevice` object by addressing the AUHAL; the AUHAL automatically passes along any property calls meant for the audio device.

An AUHAL instance can connect to only one device at a time, so you can enable both input and output only if the device can accept both. For example, the built-in audio for PowerPC-based Macintosh computers is configured as a single device that can both accept input audio data (through the Mic in) and output audio (through the speaker).

Note: Some audio hardware, including USB audio devices and built-in audio on the current line of Intel-based Macintosh computers, are represented by separate audio devices for input and output. See "Using Aggregate Devices" (page 65) for information about how you can combine these separate devices into a single `AudioDevice` object.

For the purposes of signal flow, the AUHAL configured for both input and output behaves as two audio units. For example, when output is enabled, the AUHAL invokes the previous audio unit's render callback. If an audio unit needs input data from the device, it invokes the AUHAL's render callback. Figure 3-4 shows the AUHAL used for both input and output.

Figure 3-4 The AUHAL used for input and output



An audio signal coming in through the external device is translated into an audio data stream and passed to the AUHAL, which can then send it on to another audio unit. After processing the data (for example, adding effects, or mixing with other audio data), the output is sent back to the AUHAL, which can then output the audio through the same external device.

For examples of using the AUHAL for input and output, see the *CAPlayThrough* code sample. `CAPlayThrough` shows how to implement input and output using an AUHAL for input and the default output unit for output.

Using Aggregate Devices

When interfacing with hardware audio devices, Core Audio allows you to add an additional level of abstraction, creating aggregate devices which combine the inputs and outputs of multiple devices to appear as a single device. For example, if you need to accommodate five channels of audio output, you can assign two channels of output to one device and the other three to another device. Core Audio automatically routes the data flow

to both devices, while your application can interact with the output as if it were a single device. Core Audio also works on your behalf to ensure proper audio synchronization and to minimize latency, allowing you to focus on details specific to your application or plug-in.

Users can create aggregate devices in the Audio MIDI Setup application by selecting the Audio > Open Aggregate Device Editor menu item. After selecting the sub devices to combine as an aggregate device, the user can configure the device's input and output channels like any other hardware device. The user also needs to indicate which sub device's clock should act as the master for synchronization purposes.

Any aggregate devices the user creates are global to the system. You can create aggregate devices that are local to the application process programmatically using HAL Services function calls. An aggregate device appears as an `AudioAggregateDevice` object (a subclass of `AudioDevice`) in the HAL.

> **Note:** Aggregate devices can be used to hide implementation details. For example, USB audio devices normally require separate drivers for input and output, which appear as separate `AudioDevice` objects. However, by creating a global aggregate device, the HAL can represent the drivers as a single `AudioDevice` object.

An aggregate device retains knowledge of its sub devices. If the user removes a sub device (or configures it in an incompatible manner), those channels disappear from the aggregate, but those channels will reappear when the sub device is reattached or reconfigured.

Aggregate devices have some limitations:

- All the sub devices that make up the aggregate device must be running at the same sampling rate, and their data streams must be mixable.
- They don't provide any configurable controls, such as volume, mute, or input source selection.
- You cannot specify an aggregate device to be a default input or output device unless all of its sub devices can be a default device. Otherwise, applications must explicitly select an aggregate device in order to use it.
- Currently only devices represented by an `IOAudio` family (that is, kernel-level) driver can be added to an aggregate device.

Creating Audio Units

For detailed information about creating audio units, see *Audio Unit Programming Guide*.

Hosting Audio Units

Audio units, being plug-ins, require a host application to load and control them.

Because audio units are Component Manager components, a host application must call the Component Manager to load them. The host application can find and instantiate audio units if they are installed in one of the following folders:

- `~/Library/Audio/Plug-Ins/Components`. Audio units installed here may only be used by the owner of the home folder.
- `/Library/Audio/Plug-Ins/Components`. Audio units installed here are available to all users.
- `/System/Library/Components`. The default location for Apple-supplied audio units.

If you need to obtain a list of available audio units (to display to the user, for example), you need to call the Component Manager function `CountComponents` to determine how many audio units of a particular type are available, then iterate using `FindNextComponent` to obtain information about each unit. A `ComponentDescription` structure contains the identifiers for each audio unit (its type, subtype, and manufacturer codes). See "System-Supplied Audio Units in OS X" (page 85) for a list of Component Manager types and subtypes for Apple-supplied audio units. The host can also open each unit (by calling `OpenComponent`) so it can query it for various property information, such as the audio unit's default input and output data formats, which the host can then cache and present to the user.

In most cases, an audio processing graph is the simplest way to connect audio units. One advantage of using a processing graph is that the API takes care of making individual Component Manager calls to instantiate or destroy audio units. To create a graph, call `NewAUGraph`, which returns a new graph object. Then you can add audio units to the graph by calling `AUGraphNewNode`. A graph must end in an output unit, either a hardware interface (such as the default output unit or the AUHAL) or the generic output unit.

After adding the units that will make up the processing graph, call `AUGraphOpen`. This function is equivalent to calling `OpenComponent` on each of the audio units in the graph. At this time, you can set audio unit properties such as the channel layout, sampling rate, or properties specific to a particular unit (such as the number of inputs and outputs it contains).

To make individual connections between audio units, call `AUGraphConnectNodeInput`, specifying the output and input to connect. The audio unit chain must end in an output unit; otherwise the host application has no way to start and stop audio processing.

If the audio unit has a user interface, the host application is responsible for displaying it. Audio units may supply a Cocoa or a Carbon-based interface (or both). Code for the user interface is typically bundled along with the audio unit.

- If the interface is Cocoa-based, the host application must query the unit property `kAudioUnitProperty_CocoaUI` to find the custom class that implements the interface (a subclass of `NSView`) and create an instance of that class.
- If the interface is Carbon-based, the user interface is stored as one or more Component Manager components. You can obtain the component identifiers (type, subtype, manufacturer) by querying the `kAudioUnitProperty_GetUIComponentList` property. The host application can then instantiate the user interface by calling `AudioUnitCarbonViewCreate` on a given component, which displays its interface in a window as an HIView.

After building the signal chain, you can initialize the audio units by calling `AUGraphInitialize`. Doing so invokes the initialization function for each audio unit, allowing it to allocate memory for rendering, configure channel information, and so on. Then you can call `AUGraphStart`, which initiates processing. The output unit then requests audio data from the previous unit in the chain (by means of a callback), which then calls its predecessor, and so on. The source of the audio may be an audio unit (such as a generator unit or AUHAL) or the host application may supply audio data itself by registering a callback with the first audio unit in the signal chain (by setting the unit's `kAudioUnitProperty_SetRenderCallback` property).

While an audio unit is instantiated, the host application may want to know about changes to parameter or property values; it can register a listener object to be notified when changes occur. For details on how to implement such a listener, see Technical Note TN2104: Handling Audio Unit Events.

When the host wants to stop signal processing, it calls `AUGraphStop`.

To uninitialize all the audio units in a graph, call `AUGraphUninitialize`. When back in the uninitialized state, you can still modify audio unit properties and make or change connections. If you call `AUGraphClose`, each audio unit in the graph is deallocated by a `CloseComponent` call. However, the graph still retains the nodal information regarding which units it contains.

To dispose of a processing graph, call `DisposeAUGraph`. Disposing of a graph automatically disposes of any instantiated audio units it contains.

For examples of hosting audio units, see the `Services/AudioUnitHosting` and `Services/CocoaAUHost` examples in the Core Audio SDK.

For an example of implementing an audio unit user interface, see the `AudioUnits/CarbonGenericView` example in the Core Audio SDK. You can use this example with any audio unit containing user-adjustable parameters.

For more information about using the Component Manager, see the following documentation:

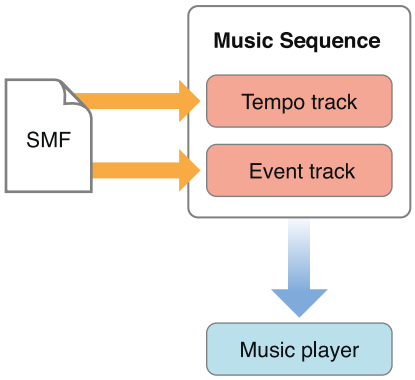
- *Component Manager Reference*
- *Component Manager for QuickTime*

- Component Manager documentation in Inside Macintosh: More Macintosh Toolbox. Although this is a legacy document, it provides a good conceptual overview of the Component Manager.

Handling MIDI Data

When working with MIDI data, an application might need to load track data from a standard MIDI file (SMF). You can invoke a Music Player function (`MusicSequenceLoadSMFWithFlags` or `MusicSequenceLoadSMFDataWithFlags`) to read in data in the Standard MIDI Format, as shown in Figure 3-5.

Figure 3-5 Reading a standard MIDI file

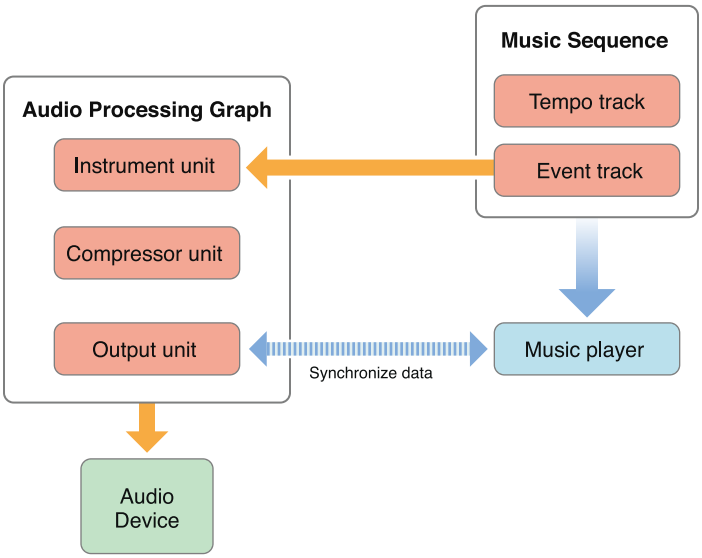


Depending on the type of MIDI file and the flags set when loading a file, you can store all the MIDI data in single track, or store each MIDI channel as a separate track in the sequence. By default, each MIDI channel is mapped sequentially to a new track in the sequence. For example, if the MIDI data contains channels 1, 3, and, 4, three new tracks are added to the sequence, containing data for channels 1, 3, and 4 respectively. These tracks are appended to the sequence at the end of any existing tracks. Each track in a sequence is assigned a zero-based index value.

Timing information (that is, tempo events) goes to the tempo track.

Once you have loaded MIDI data into the sequence, you can assign a music player instance to play it, as shown in Figure 3-6.

Figure 3-6 Playing MIDI data

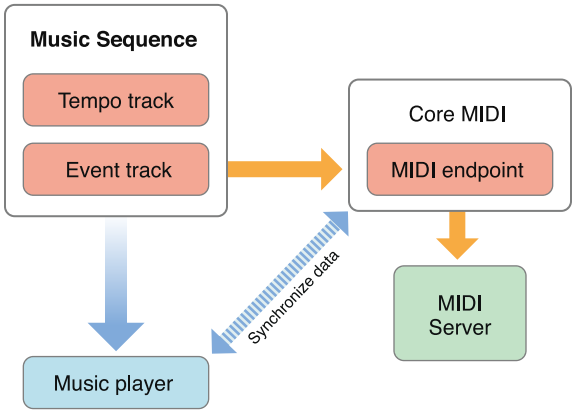


The sequence must be associated with a particular audio processing graph, and the tracks in the sequence can be assigned to one or more instrument units in the graph. (If you don't specify a track mapping, the music player sends all the MIDI data to the first instrument unit it finds in the graph.) The music player assigned to the sequence automatically communicates with the graph's output unit to make sure the outgoing audio data is properly synchronized. The compressor unit, while not required, is useful for ensuring that the dynamic range of the instrument unit's output stays consistent.

MIDI data in a sequence can also go to external MIDI hardware (or software configured as a virtual MIDI destination), as shown in Figure 3-7.

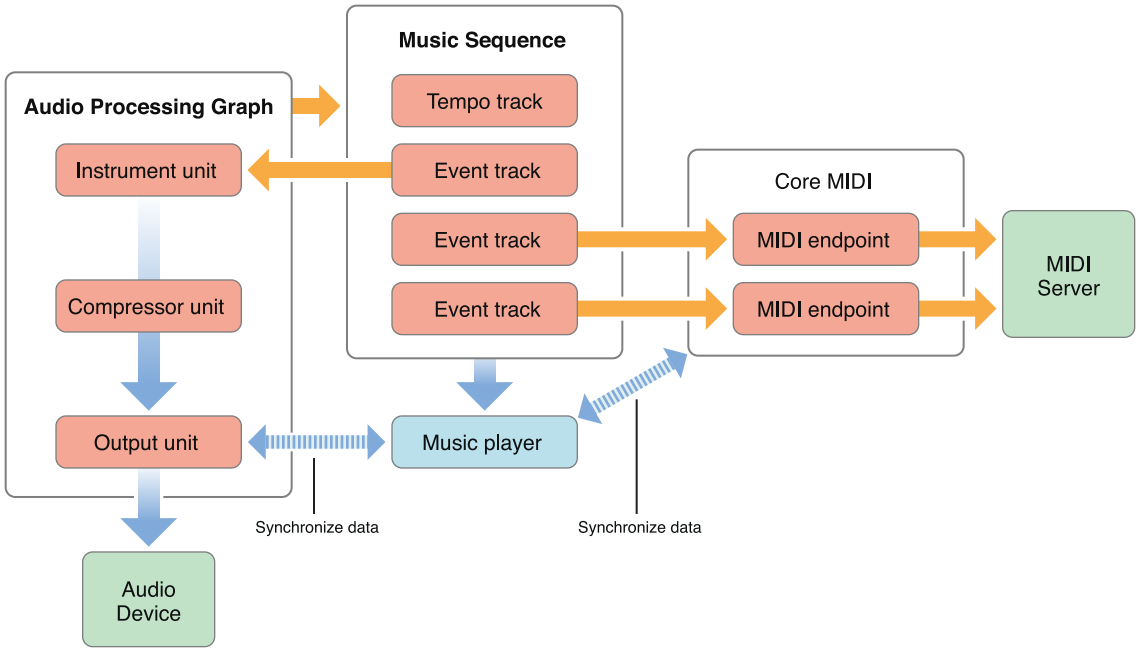
Tracks destined for MIDI output must be assigned a MIDI endpoint. The music player communicates with Core MIDI to ensure that the data flow to the MIDI device is properly synchronized. Core MIDI then coordinates with the MIDI Server to transmit the data to the MIDI instrument.

Figure 3-7 Sending MIDI data to a MIDI device



A sequence of tracks can be assigned to a combination of instrument units and MIDI devices. For example, you can assign some of the tracks to play through an instrument unit, while other tracks go through Core MIDI to play through external MIDI devices, as shown in Figure 3-8.

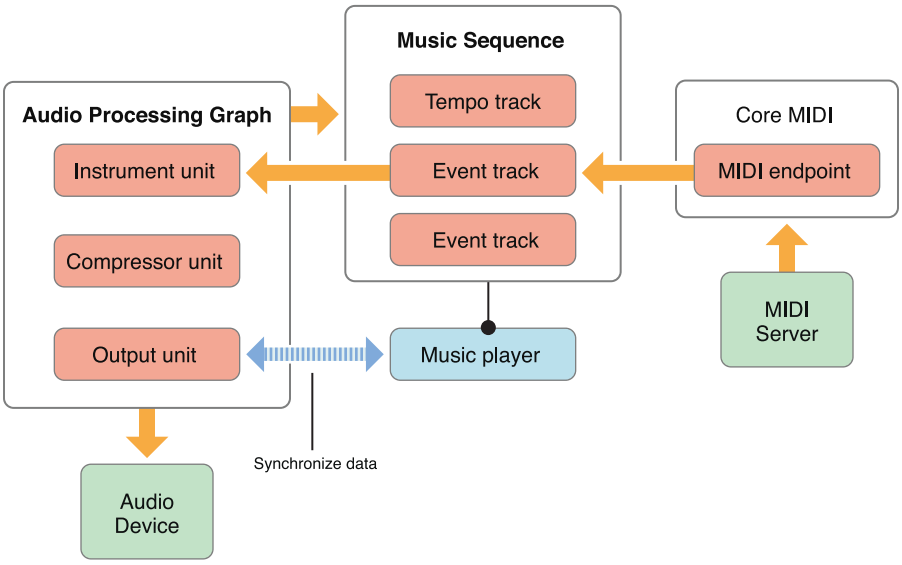
Figure 3-8 Playing both MIDI devices and a virtual instrument



The music player automatically coordinates between the audio processing graph's output unit and Core MIDI to ensure that the outputs are synchronized.

Another common scenario is to play back already existing track data while accepting new MIDI input, as shown in Figure 3-9.

Figure 3-9 Accepting new track input



The playback of existing data is handled as usual through the audio processing graph, which sends audio data to the output unit. New data from an external MIDI device enters through Core MIDI and is transferred through the assigned endpoint. Your application must iterate through this incoming data and write the MIDI events to a new or existing track. The Music Player API contains functions to add new tracks to a sequence, and to write time-stamped MIDI events or other messages to a track.

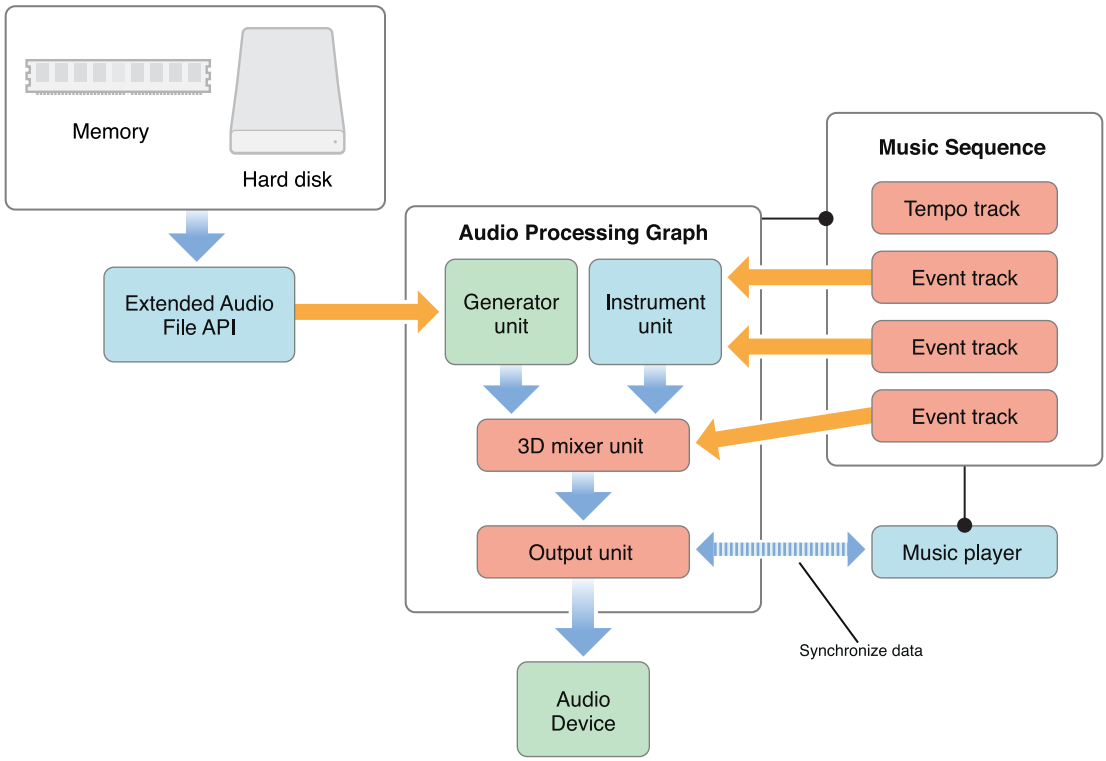
For examples of handling and playing MIDI data, see the following examples in the Core Audio SDK:

- `MIDI/SampleTools`, which shows a simple way to send and receive MIDI data.
- `SimpleSDK/PlaySoftMIDI`, which sends MIDI data to a simple processing graph consisting of an instrument unit and an output unit.
- `SimpleSDK/PlaySequence`, which reads in a MIDI file into a sequence and uses a music player to play it.

Handling Audio and MIDI Data Together

Sometimes you want to combine audio data with audio synthesized from MIDI data and play back the result. For example, the audio for many games consists of background music, which is stored as an audio file on disk, along with noises triggered by events (footsteps, gunfire, and so on), which are generated as MIDI data. Figure 3-10 shows how you can use Core Audio to combine the two.

Figure 3-10 Combining audio and MIDI data



The soundtrack audio data is retrieved from disk or memory and converted to linear PCM using the Extended Audio File API. The MIDI data, stored as tracks in a music sequence, is sent to a virtual instrument unit. The output from the virtual instrument unit is in linear PCM format and can then be combined with the soundtrack data. This example uses a 3D mixer unit, which can position audio sources in a three-dimensional space. One of the tracks in the sequence is sending event data to the mixer unit, which alters the positioning parameters, making the sound appear to move over time. The application would have to monitor the player's movements and add events to the special movement track as necessary.

For an example of loading and playing file-based audio data, see `SimpleSDK/PlayFile` in the Core Audio SDK.

Core Audio Frameworks

Core Audio consists of a number of separate frameworks, which you can find in `/System/Library/Frameworks`. These frameworks are not grouped under an umbrella framework, so finding particular headers can sometimes be tricky. This appendix describes each of the Core Audio frameworks and their associated header files.

Frameworks Available in iOS and OS X

The frameworks listed in this section are available in iOS 2.0 and OS X v10.5.

AudioToolbox.framework

The Audio Toolbox framework contains the APIs that provide application-level services. The Audio Toolbox framework includes these header files:

- `AudioConverter.h`: Audio Converter API. Defines the interface used to create and use audio converters.
- `AudioFile.h`: Defines an interface for reading and writing audio data in files.
- `AudioFileStream.h`: Defines an interface for parsing audio file streams.
- `AudioFormat.h`: Defines the interface used to assign and read audio format metadata in audio files.
- `AudioQueue.h`: Defines an interface for playing and recording audio.
- `AudioServices.h`: Defines three interfaces. System Sound Services lets you play short sounds and alerts. Audio Hardware Services provides a lightweight interface for interacting with audio hardware. Audio Session Services lets iPhone and iPod touch applications manage audio sessions.
- `AudioToolbox.h`: Top-level include file for the Audio Toolbox framework.
- `AUGraph.h`: Defines the interface used to create and use audio processing graphs.
- `ExtendedAudioFile.h`: Defines the interface used to translate audio data from files directly into linear PCM, and vice versa.

In OS X you have these additional header files:

- `AudioFileComponents.h`: Defines the interface for Audio File Component Manager components. You use an audio file component to implement reading and writing a custom file format.

- `AudioUnitUtilities.h`: Utility functions for interacting with audio units. Includes audio unit parameter conversion functions, and audio unit event functions to create listener objects, which invoke a callback when specified audio unit parameters have changed.
- `CAFFile.h`: Defines the Core Audio Format audio file format. See *Apple Core Audio Format Specification 1.0* for more information.
- `CoreAudioClock.h`: Lets you designate a timing source for synchronizing applications or devices.
- `MusicPlayer.h`: Defines the interface used to manage and play event tracks in music sequences.
- `AUMIDIController.h`: Deprecated: Do not use. An interface to allow audio units to receive data from a designated MIDI source. Standard MIDI messages are translated into audio unit parameter values. This interface is superseded by functions in the Music Player API.
- `DefaultAudioOutput.h`: Deprecated: Do not use. Defines an older interface for accessing the default output unit (deprecated in OS X v10.3 and later).

AudioUnit.framework

The Audio Unit framework contains the APIs used for managing plug-ins in Core Audio. Except as noted, the Audio Unit framework includes these header files:

- `AUComponent.h`: Defines the audio unit types.
- `AudioComponent.h`: (iOS only) Defines the interface for using audio components.
- `AudioOutputUnit.h`: Defines the interface used to turn an output unit on or off.
- `AudioUnit.h`: Include file for the Audio Unit framework.
- `AudioUnitParameters.h`: Predefined parameter constants used by Apple's audio units. Third parties can also use these constants for their own audio units.
- `AudioUnitProperties.h`: Predefined audio unit properties for common audio unit types as well as Apple's audio units.

In OS X you have these additional header files:

- `AUCocoaUIView.h`: Defines the protocol for a custom Cocoa view you can use to hold your audio unit's user interface. See also `CoreAudioKit.framework/AUGenericView.h`.
- `AudioCodec.h`: Defines the interface used specifically for creating audio codec components..
- `AudioUnitCarbonView.h`: Defines the interface for loading and interacting with a Carbon-based audio unit user interface. A Carbon interface is packaged as a Component Manager component and appears as an HIView.
- `AUNTComponent.h`: Deprecated: Do not use. Defines the interface for older "v1" audio units. Deprecated in OS X v10.3 and later. Replaced by `AUComponent.h`.

- `LogicAUProperties.h`: An interface for audio units that run in the Logic Node environment of the Logic Studio application.
- `MusicDevice.h`: An interface for creating instrument units (that is, software-based music synthesizers).

CoreAudio.framework

The Core Audio framework contains data types common to all Core Audio services, as well as lower-level APIs used to interact with hardware. In OS X, this framework contains the interfaces for Hardware Abstraction Layer (HAL) Services.

This framework includes this header file:

- `CoreAudioTypes.h`: Defines data types used by all of Core Audio.

In OS X you have these additional header files:

- `AudioDriverPlugin.h`: Defines the interface used to communicate with an audio driver plug-in.
- `AudioHardware.h`: Defines the interface for interacting with audio device objects. An audio device object represents an external device in the hardware abstraction layer (HAL).
- `AudioHardwarePlugin.h`: Defines the CFPlugin interface required for a HAL plug-in. An instance of a plug-in appears as an audio device object in the HAL.
- `CoreAudio.h`: Top-level include file for the Core Audio framework.
- `HostTime.h`: Contains functions to obtain and convert the host's time base.

OpenAL.framework

The OpenAL framework provides an implementation of the the OpenAL specification. This framework includes these two header files:

- al.h
- alc.h

In iOS you have these additional header files:

- `oalMacOSX_OALExtensions.h`
- `oalStaticBufferExtension.h`

In OS X you have this additional header file:

- `MacOSX_OALExtensions.h`

Frameworks Available in iOS Only

The frameworks listed in this section are available in iOS only.

AVFoundation.framework

The AV Foundation framework provides an Objective-C interface for playing back audio with the control needed by most applications. The AV Foundation framework in iOS includes one header file:

- `AVAudioPlayer.h`: Defines an interface for playing audio from a file or from memory.

Frameworks Available in OS X Only

The frameworks listed in this section are available in OS X only.

CoreAudioKit.framework

The Core Audio Kit framework contains APIs used for creating a Cocoa user interface for an audio unit.

- `CoreAudioKit.h`: Top-level include file for the Core Audio Kit framework.
- `AUGenericView.h`: Defines a generic Cocoa view class for use with audio units. This is the bare-bones user interface that is displayed if an audio unit doesn't create its own custom interface.
- `AUPannerView.h`: Defines and instantiates a generic view for use with panner audio units.

CoreMIDI.framework

The Core MIDI framework contains all Core MIDI Services APIs used to implement MIDI support in applications.

- `CoreMIDI.h`: Top-level include file for the Core MIDI framework.
- `MIDIServices.h`: Defines the interface used to set up and configure an application to communicate with MIDI devices (through MIDI endpoints, notifications, and so on).
- `MIDISetup.h`: Defines the interface used to configure or customize the global state of the MIDI system (that is, available MIDI devices, MIDI endpoints, and so on).
- `MIDIThruConnection.h`: Defines functions to create MIDI play-through connections between MIDI sources and destinations. A MIDI thru connection allows you to daisy-chain MIDI devices, allowing input to one device to pass through to another device as well.

CoreMIDIServer.framework

The Core MIDI Server framework contains interfaces for MIDI drivers.

- `CoreMIDIServer.h`: Top-level include file for the Core MIDI Server framework.
- `MIDIDriver.h`: Defines the CFPlugin interface used by MIDI drivers to interact with the Core MIDI Server.

Core Audio Services

This chapter lists the services available in Core Audio. In iOS, you find these services arranged into the following frameworks:

- **Audio Toolbox**—Application-level services: files, streams, alerts, playback and recording. In iOS, includes audio session services.
- **Audio Unit**—Audio unit and audio codec services.
- **AV Foundation**—An Objective-C audio playback interface.
- **Core Audio**—Data types and, in OS X, hardware services.
- **OpenAL**—Positional and low-latency audio services.

Core Audio in OS X includes those four frameworks and adds three more:

- **Core Audio Kit**—Audio unit user-interface services.
- **Core MIDI**—Application-level MIDI support.
- **Core MIDI Server**—MIDI server and driver support.

For a framework-focused view of the header files in Core Audio, see the Appendix "Core Audio Frameworks" (page 74).

The rest of this chapter presents a services-focused view of Core Audio—starting with services available in both iOS and OS X.

Services Available in iOS and OS X

The services listed in this section are available in iOS 2.0 and OS X v10.5.

Audio Converter Services

Audio Converter Services allows you to convert data between formats. This interface consists of the functions, data types, and constants declared in the `AudioConverter.h` header file in `AudioToolbox.framework`.

Audio File Services

Audio File services lets you read or write audio data to and from a file or buffer. You use it in conjunction with Audio Queue Services to record or play audio. In iOS and OS X, Audio File Services consists of the functions, data types, and constants declared in the `AudioFile.h` header file in `AudioToolbox.framework`.

Audio File Stream Services

Audio File Stream services lets you parse audio file streams—that is, audio data for which you don't necessarily have access to the entire file. You can also use it to parse file data from disk, although Audio File Services is designed for that purpose.

Audio File Stream services returns audio data and metadata to your application via callbacks. which you typically then play back using Audio Queue Services. In iOS and OS X, Audio File Stream Services consists of the functions, data types, and constants declared in the `AudioFileStream.h` header file in `AudioToolbox.framework`.

Audio Format Services

Audio Format Services lets you work with audio data format information. Other services, such as Audio File Services, have functions for this use as well. You use Audio Format Services when all you want to do is obtain audio data format information. In OS X, you can also use this service to get system characteristics such as the available sample rates for encoding. Audio Format Services consists of the functions, data types, and constants declared in the `AudioFormat.h` header file in `AudioToolbox.framework`.

Audio Processing Graph Services

Audio Processing Graph Services lets you create and manipulate audio processing graphs in your application. In iOS and in OS X, it consists of the functions, data types, and constants declared in `AUGraph.h` header file in `AudioToolbox.framework`.

Audio Queue Services

Audio Queue Services lets you play or record audio. It also lets you pause and resume playback, perform looping, and synchronize multiple channels of audio. In iOS and OS X, Audio Queue Services consists of the functions, data types, and constants declared in the `AudioQueue.h` header file in `AudioToolbox.framework`.

Audio Unit Services

Audio Unit Services lets you load and use audio units in your application.

In iOS, Audio Unit Services comprises the functions, data types, and constants declared in the following header files in `AudioUnit.framework`:

- `AUComponent.h`
- `AudioComponent`.h (iOS only)
- `AudioOutputUnit.h`
- `AudioUnitParameters.h`
- `AudioUnitProperties.h`

OS X adds to these the following header files from `AudioUnit.framework` and `AudioToolbox.framework`:

- `AUCocoaUIView.h`
- `AudioUnitCarbonView.h`
- `AudioUnitUtilities.h` (in `AudioToolbox.framework`)
- `LogicAUProperties`.h
- `MusicDevice.h`

OpenAL

OpenAL is an open-source positional audio technology designed for use in game applications. iOS and OS X have implementations of the OpenAL 1.1 specification. You find its headers in the following header files in the OpenAL framework:

- al.h
- alc.h

In iOS you have these additional header files:

- `oalMacOSX_OALExtensions.h`
- `oalStaticBufferExtension.h`

In OS X you have this additional header file:

- `MacOSX_OALExtensions.h`

System Sound Services

System Sound Services lets you play short sounds and alerts. On the iPhone, it lets you invoke vibration. System Sound Services consists of a subset of the functions, data types, and constants declared in the `AudioServices.h` header file in `AudioToolbox.framework`.

Services Available in iOS Only

The services listed in this section are available in iOS only.

Audio Session Services

Audio Session Services lets you manage audio sessions in your application—coordinating the audio behavior in your application with background applications on an iPhone or iPod touch. Audio Session Services consists of a subset of the functions, data types, and constants declared in the `AudioServices.h` header file in `AudioToolbox.framework`.

The AVAudioPlayer Class

The `AVAudioPlayer` class provides a simple Objective-C interface for playing sounds. If your application does not require stereo positioning or precise synchronization, and if you are not playing audio captured from a network stream, Apple recommends that you use this class for playback. This class is declared in the `AVAudioPlayer.h` header file in `AVFoundation.framework`.

Services Available in OS X Only

The services listed in this section are available in OS X only.

Audio Codec Services

Audio Codec Services allows you to convert data between formats. This interface consists of the functions, data types, and constants declared in the `AudioCodec.h` header file in `AudioUnit.framework`.

- `AudioCodec.h` (located in `AudioUnit.framework`).

Audio Hardware Services

Audio Hardware Services provides a small, lightweight interface to some of the important features of the Audio HAL (hardware abstraction layer). Audio Hardware Services consists of a subset of the functions, data types, and constants declared in the `AudioServices.h` header file in `AudioToolbox.framework`.

Core Audio Clock Services

Core Audio Clock Services provides a reference clock that you can use to synchronize applications and devices. This service consists of the functions, data types, and constants declared in the `CoreAudioClock.h` header file in `AudioToolbox.framework`.

Core MIDI Services

Core Audio in OS X supports MIDI through Core MIDI Services, which consists of the functions, data types, and constants declared in the following header files in `CoreMIDI.framework`:

- `MIDIServices.h`
- `MIDISetup.h`
- `MIDIThruConnection.h`
- `MIDIDriver.h`

Core MIDI Server Services

Core MIDI Server Services lets MIDI drivers communicate with the OS X MIDI server. This interface consists of the functions, data types, and constants declared in the following header files in `CoreMIDIServer.framework`:

- `CoreMIDIServer.h`
- `MIDIDriver.h`

Extended Audio File Services

about extended audio file services

In many cases, you use Extended Audio File Services, which provides the simplest interface for reading and writing audio data. Files read using this API are automatically uncompressed and/or converted into linear PCM format, which is the native format for audio units. Similarly, you can use one function call to write linear PCM audio data to a file in a compressed or converted format. "Supported Audio File and Data Formats in OS X" (page 89) lists the file formats that Core Audio supports by default. Some formats have restrictions; for example, by default, Core Audio can read, but not write, MP3 files.

Hardware Abstraction Layer (HAL) Services

Core Audio in OS X uses a hardware abstraction layer (HAL) to provide a consistent and predictable interface for applications to deal with hardware. Each piece of hardware is represented by an audio device object (type `AudioDevice`) in the HAL. Applications can query the audio device object to obtain timing information that can be used for synchronization or to adjust for latency.

HAL Services consists of the functions, data types, and constants declared in the following header files in `CoreAudio.framework`:

- `AudioDriverPlugin.h`
- `AudioHardware.h`
- `AudioHardwarePlugin.h`
- `CoreAudioTypes.h` (Contains data types and constants used by all Core Audio interfaces)
- `HostTime.h`

Most developers will find that Apple's AUHAL unit serves their hardware interface needs, so they don't have to interact directly with the HAL Services. The AUHAL is responsible for transmitting audio data, including any required channel mapping, to the specified audio device object. For more information about using the AUHAL unit and other output units, see "Interfacing with Hardware" (page 63).

Music Player Services

Music Player Services, available in OS X, allows you to arrange and play a collection of MIDI music tracks. It consists of the functions, data types, and constants declared in the header file `MusicPlayer.h` in `AudioToolbox.framework`.

System-Supplied Audio Units in OS X

The tables in this appendix list the audio units that ship with OS X v10.5, grouped by Component Manager type. The Component Manager manufacturer identifier for all these units is `kAudioUnitManufacturer_Apple`.

Table C-1 System-supplied effect units (`kAudioUnitType_Effect`)

Effect Units	Subtype	Description
AUBandpass	`kAudioUnitSubType_BandPassFilter`	A single-band bandpass filter.
AUDynamicsProcessor	`kAudioUnitSubType_DynamicsProcessor`	A dynamics processor that lets you set parameters such as headroom, the amount of compression, attack and release times, and so on.
AUDelay	`kAudioUnitSubType_Delay`	A delay unit.
AUFilter	`kAudioUnitSubType_AUFilter`	A five-band filter, allowing for low and high frequency cutoffs as well as three bandpass filters.
AUGraphicEQ	`kAudioUnitSubType_GraphicEQ`	A 10-band or 31-band graphic equalizer.
AUHiPass	`kAudioUnitSubType_HighPassFilter`	A high-pass filter with an adjustable resonance peak.
AUHighShelfFilter	`kAudioUnitSubType_HighShelfFilter`	A filter that allows you to boost or cut high frequencies by a fixed amount.
AUPeakLimiter	`kAudioUnitSubType_PeakLimiter`	A peak limiter.
AULowPass	`kAudioUnitSubType_LowPassFilter`	A low-pass filter with an adjustable resonance peak.
AULowShelfFilter	`kAudioUnitSubType_LowShelfFilter`	A filter that allows you to boost or cut low frequencies by a fixed amount.

Effect Units	Subtype	Description
AUMultibandCompressor	`kAudioUnitSubType_MultiBandCompressor`	A four-band compressor.
AUMatrixReverb	`kAudioUnitSubType_MatrixReverb`	A reverberation unit that allows you to specify spatial characteristics, such as size of room, material absorption characteristics, and so on.
AUNetSend	`kAudioUnitSubType_NetSend`	A unit that streams audio data over a network. Used in conjunction with the AUNetReceive generator audio unit.
AUParametricEQ	`kAudioUnitSubType_ParametricEQ`	A parametric equalizer.
AUSampleDelay	`kAudioUnitSubType_SampleDelay`	A delay unit that allows you to set the delay by number of samples rather than by time.
AUPitch	`kAudioUnitSubType_Pitch`	An effect unit that lets you alter the pitch of the sound without changing the speed of playback.

Table C-2 System-supplied instrument unit (`kAudioUnitType_MusicDevice`)

Instrument Unit	Subtype	Description
DLSMusicDevice	`kAudioUnitSubType_DLSSynth`	A virtual instrument unit that lets you play MIDI data using sound banks in the SoundFont or Downloadable Sounds (DLS) format. Sound banks must be stored in the `/Library/Audio/Sounds/Banks` folder of either your home or system directory.

Table C-3 System-supplied mixer units (`kAudioUnitType_Mixer`)

Mixer Units	Subtype	Description
AUMixer3D	`kAudioUnitSubType_3DMixer`	A special mixing unit that can take several different signals and mix them so they appear to be positioned in a three-dimensional space. For details on using this unit, see Technical Note TN2112: Using the 3DMixer Audio Unit.

Mixer Units	Subtype	Description
AUMatrixMixer	`kAudioUnitSubType_MatrixMixer`	A unit that mixes an arbitrary number of inputs to an arbitrary number of outputs.
AUMixer	`kAudioUnitSubType_StereoMixer`	A unit that mixes an arbitrary number of mono or stereo inputs to a single stereo output.

Table C-4 System-supplied converter units (`kAudioUnitType_FormatConverter`)

Converter Unit	Subtype	Description
AUConverter	`kAudioUnitSubType_AUConverter`	A generic converter to handle data conversions within the linear PCM format. That is, it can handle sample rate conversions, integer to floating point conversions (and vice versa), interleaving, and so on. This audio unit is essentially a wrapper around an audio converter.
AUDeferredRenderer	`kAudioUnitSubType_DeferredRenderer`	An audio unit that obtains its input from one thread and sends its output to another; you can use this unit to divide your audio processing chain among multiple threads.
AUMerger	`kAudioUnitSubType_Merger`	An unit that combines two separate audio inputs.
AUSplitter	`kAudioUnitSubType_Splitter`	A unit that splits an audio input into two separate audio outputs.
AUTimePitch	`kAudioUnitSubType_TimePitch`	A unit that lets you change the speed of playback without altering the pitch, or vice versa.
AUVarispeed	`kAudioUnitSubType_Varispeed`	A unit that lets you change the speed of playback (and consequently the pitch as well).

Table C-5 System-supplied output units (`kAudioUnitType_Output`)

Output Unit	Subtype	Description
AudioDeviceOutput	`kAudioUnitSubType_HALOutput`	A unit that interfaces with an audio device using the hardware abstraction layer. Also called the AUHAL. Despite its name, the AudioDeviceOutput unit can also be configured to accept device input. See "Interfacing with Hardware" (page 63) for more details.

Output Unit	Subtype	Description
DefaultOutputUnit	`kAudioUnitSubType_DefaultOutput`	An output unit that sends its input data to the user-designated default output (such as the computer's speaker).
GenericOutput	`kAudioUnitSubType_GenericOutput`	A generic output unit that contains the signal format control and conversion features of an output unit, but doesn't interface with an output device. Typically used for the output of an audio processing subgraph. See "Audio Processing Graphs" (page 53).
SystemOutputUnit	`kAudioUnitSubType_SystemOutput`	An output unit that sends its input data to the standard system output. System output is the output designated for system sounds and effects, which the user can set in the Sound Effects tab of the Sound preference panel.

Table C-6 System-supplied generator units (`kAudioUnitType_Generator`)

Generator Unit	Subtype	Description
AUAudioFilePlayer	`kAudioUnitSubType_AudioFilePlayer`	A unit that obtains and plays audio data from a file.
AUNetReceive	`kAudioUnitSubType_NetReceive`	A unit that receives streamed audio data from a network. Used in conjunction with the AUNetSend audio unit.
AUScheduledSoundPlayer	`kAudioUnitSubType_ScheduledSoundPlayer`	A unit that plays audio data from one or more buffers in memory.

Supported Audio File and Data Formats in OS X

This appendix describes the audio data and file formats supported in Core Audio in OS X v10.5.

Each audio file type lists the data formats supported for that type. That is, a converter exists to convert data from the particular file format to any of the listed data formats. Some data formats (such as AC3) cannot be converted to a linear PCM format and therefore cannot be handled by standard audio units.

A Core Audio Format (CAF) file can contain audio data of any format. Any application that supports the CAF file format can write audio data to the file or extract the data it contains. However, the ability to encode or decode the audio data contained within it is dependent on the audio codecs available on the system.

Table D-1 Allowable data formats for each file format.

File Format	Data Formats
AAC (`.aac`, `.adts`)	`'aac '`
AC3 (`.ac3`)	`'ac-3'`
AIFC (`.aif`, `.aiff`,`.aifc`)	`BEI8`, `BEI16`, `BEI24`, `BEI32`, `BEF32`, `BEF64`, `'ulaw'`, `'alaw'`, `'MAC3'`, `'MAC6'`, `'ima4' `, `'QDMC'`, `'QDM2'`, `'Qclp'`, `'agsm'`
AIFF (`.aiff`)	`BEI8`, `BEI16`, `BEI24`, `BEI32`
Apple Core Audio Format (`.caf`)	`'.mp3'`, `'MAC3'`, `'MAC6'`, `'QDM2'`, `'QDMC'`, `'Qclp'`, `'Qclq'`, `'aac '`, `'agsm'`, `'alac'`, `'alaw'`, `'drms'`, `'dvi '`, `'ima4'`, `'lpc '`, `BEI8`, `BEI16`, `BEI24`, `BEI32`, `BEF32`, `BEF64`, `LEI16`, `LEI24`, `LEI32`, `LEF32`, `LEF64`, `'ms\x00\x02'`, `'ms\x00\x11'`, `'ms\x001'`, `'ms\x00U'`, `'ms \x00'`, `'samr'`, `'ulaw'`
MPEG Layer 3 (`.mp3`)	`'.mp3'`
MPEG 4 Audio (`.mp4`)	`'aac '`
MPEG 4 Audio (`.m4a`)	`'aac ', alac'`
NeXT/Sun Audio (`.snd`, `.au`)	`BEI8`, `BEI16`, `BEI24`, `BEI32`, `BEF32`, `BEF64`, `'ulaw'`
Sound Designer II (`.sd2`)	`BEI8`, `BEI16`, `BEI24`, `BEI32`
WAVE (`.wav`)	`LEUI8`, `LEI16`, `LEI24`, `LEI32`, `LEF32`, `LEF64`, `'ulaw'`, `'alaw'`

Key for linear PCM formats. For example, BEF32 = Big Endian linear PCM 32 bit floating point.

Table D-2 Key for linear PCM formats

LE	Little Endian
BE	Big Endian
F	Floating point
I	Integer
UI	Unsigned integer
8/16/24/32/64	Number of bits

Core Audio includes a number of audio codecs that translate audio data to and from Linear PCM. Codecs for the following audio data type are available in OS X v10.4. Audio applications may install additional encoders and decoders.

Audio data type	Encode from linear PCM?	Decode to linear PCM?
MPEG Layer 3 (`'.mp3'`)	No	Yes
MACE 3:1 (`'MAC3'`)	Yes	Yes
MACE 6:1 (`'MAC6'`)	Yes	Yes
QDesign Music 2 (`'QDM2'`)	Yes	Yes
QDesign (`'QDMC'`)	No	Yes
Qualcomm PureVoice (`'Qclp'`)	Yes	Yes
Qualcomm QCELP (`'qclq'`)	No	Yes
AAC (`'aac '`)	Yes	Yes
Apple Lossless (`'alac'`)	Yes	Yes
Apple GSM 10:1 (`'agsm'`)	No	Yes
ALaw 2:1 `'alaw'`)	Yes	Yes
Apple DRM Audio Decoder (`'drms'`)	No	Yes
AC-3	No	No

Audio data type	Encode from linear PCM?	Decode to linear PCM?
DVI 4:1 (`'dvi '`)	No	Yes
Apple IMA 4:1 (`'ima4'`)	Yes	Yes
LPC 23:1 (`'lpc '`)	No	Yes
Microsoft ADPCM	No	Yes
DVI ADPCM	Yes	Yes
GSM610	No	Yes
AMR Narrowband (`'samr'`)	Yes	Yes
µLaw 2:1 (`'ulaw'`)	Yes	Yes

Document Revision History

This table describes the changes to *Core Audio Overview*.

Date	Notes
2008-11-13	Updated for iOS 2.2.
2008-07-08	Updated for iOS 2.0 and OS X v10.5.
2007-01-08	Minor corrections and wording changes.
2006-08-07	New document that introduces the basic concepts and architecture of the Core Audio frameworks.

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