Core Audio Overview

Audio > Core Audio

2007-01-08

Contents

Figures and Tables

6

Introduction

Core Audio is a set of services that developers use to implement audio and music features in Mac OS X applications. Its services handle all aspects of audio, from recording, editing, and playback, compression and decompression, to MIDI (Musical Instrument Digital Interface) processing, signal processing, and audio synthesis. You can use it to write standalone applications or modular plug-ins that work with existing products.

Core Audio is available to all versions of Mac OS X, although older versions may not contain particular features. This document describes Core Audio features available as of Mac OS X v10.4.

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

This document is for all developers interested in creating audio software in Mac OS X. You should have basic knowledge of audio, digital audio, and MIDI terminology, as well as some familiarity with Mac OS X.

Organization of This Document

This document is organized into the following chapters:

- "What is Core Audio?" (page 9) describes basic features of Core Audio and their relation to other audio and recording technologies.
- "Core Audio Programming Interfaces" (page 17) describes the various programming interfaces available in Core Audio.
- "An Overview of Common Tasks" (page 29) describes at a high level how you might use Core Audio to accomplish common audio-related tasks.
- "Core Audio Frameworks" (page 39) lists the various frameworks and headers that define Core Audio.
- "System-Supplied Audio Units" (page 43) lists the audio units that ship with Mac OS X v10.4, along with their Component Manager types and subtypes.
- "Supported Audio File and Data Formats" (page 47) describes the audio file and data formats that Core Audio supports by default.

See Also

For more information about audio and Core Audio on Mac OS X, see the following resources:

- *Audio Unit Programming Guide*, which contains detailed information about creating audio units.
- *Apple Core Audio Format Specification 1.0*, which describes Apple's Core Audio File (CAF) format.

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

What is Core Audio?

Core Audio is designed to handle all audio needs in Mac OS X. You can use Core Audio to generate, record, mix, edit, process, and play audio. Using Core Audio, you can also generate, record, process, and play MIDI data, interfacing with both hardware and software MIDI instruments.

Core Audio combines C programming interfaces with tight system integration, resulting in a flexible programming environment that still maintains low latency through the signal chain. Some of Core Audio's benefits include:

■ Plug-in interfaces for audio synthesis and audio digital signal processing (DSP)

■ Built in support for reading and writing a wide variety of audio file and data formats

■ Plug-in interfaces for handling custom file and data formats

■ A modular approach for constructing signal chains

■ Scalable multichannel input and output

■ Easy synchronization of audio and MIDI data during recording or playback

■ A standardized interface to all built-in and external hardware devices, regardless of connection type (USB, Firewire, PCI, and so on)

> **Note:** Although Core Audio uses C interfaces, you can call Core Audio functions from Cocoa applications and Objective-C code.

Core Audio in Mac OS X

Core Audio is tightly integrated into Mac OS X. The majority of Core Audio services are layered on top of the Hardware Abstraction Layer (HAL), and Core MIDI as shown in Figure 1-1. Audio signals pass to and from hardware through the HAL, while Core MIDI provides a similar interface for MIDI data and MIDI devices.

Figure 1-1   Core Audio architecture
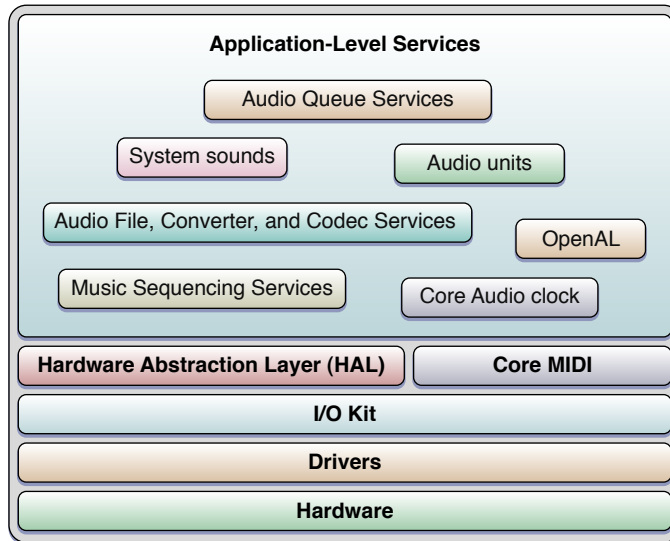
The sections that follow describe some of the essential features of Core Audio:

A Little About Digital Audio and Linear PCM

As you might expect, Core Audio handles audio data in a digital format. Most Core Audio constructs manipulate audio data in linear pulse-code-modulated (**linear PCM**) format, which is the most common uncompressed data format for digital audio. Pulse code modulation relies on measuring an audio signal's magnitude at regular intervals (the **sampling rate**) and converting each sample into a numerical value. This value varies linearly with the signal amplitude. For example, standard CD audio has a sampling rate of 44.1 kHz and uses 16 bits to represent the signal amplitude (65,536 possible values). Core Audio's data structures can describe linear PCM at any sample rate and bit depth, supporting both integer and floating-point sample values.

Core Audio generally expects audio data to be in native-endian 32-bit floating-point linear PCM format. However, you can create audio converters 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 supplies codecs to translate most common digital audio formats (though it does not supply an encoder for converting to MP3).

Core Audio also supports most common file formats for storing audio data.

Audio Units

Audio units are plug-ins that handle audio data.

Within a Mac OS X application, almost all processing and manipulation of audio data can be done through audio units (though this is not a requirement). Some units are strictly available to simplify common tasks for developers (such as splitting a signal or interfacing with hardware), while others appear onscreen as signal processors that users can insert into the signal path. For example, software-based effect processors often

mimic their real-world counterparts (such as distortion boxes) onscreen. Some audio units generate signals themselves, whether programmatically or in response to MIDI input. Some examples of audio unit implementations include the following:

■ A signal processor (for example, high-pass filter, flanger, compressor, or distortion box). An effect unit performs audio digital signal processing (DSP) and is analogous to an effects box or outboard signal processor.

■ A musical instrument or software synthesizer. These instrument units (sometimes called music device audio units) typically generate musical tones in response to MIDI input). An instrument unit can interpret MIDI data from a file or an external MIDI device.

■ A signal source. A generator unit lets you implement an audio unit as a signal generator. 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 over a network.

■ An interface to hardware input or output. For more information, see "The Hardware Abstraction Layer" (page 12) and "Interfacing with Hardware Devices" (page 30).

■ A signal converter, which uses an audio converter to convert between various linear PCM variants. See "Audio Converters and Codecs" (page 21) for more details.

■ A mixer or splitter. For example, a mixer unit can mix down tracks or apply stereo or 3D panning effects. A splitter unit might transform a mono signal into simulated stereo by splitting the signal into two channels and applying comb filtering.

■ An offline effect unit. Offline effects are either too processor-intensive or simply impossible to apply in real time. For example, a effect that reverses the samples in a file (resulting in the music being played backward) must be applied offline.

Because audio units are modular, you can mix and match them in whatever permutations you or the end user requires. Figure 1-2 shows a simple chain of audio units. This chain uses an instrument unit to generate an audio signal, which is then passed through effect units to apply bandpass filtering and distortion.

Figure 1-2   A simple audio unit chain

Instrument unit   Bandpass filter unit   Distortion unit
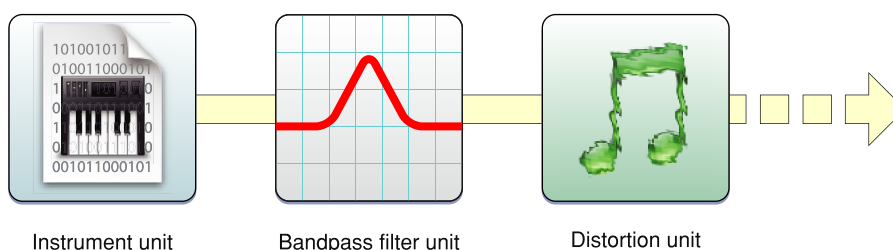
If you develop audio DSP code that you want to make available to a wide variety of applications, you should package them as audio units.

If you are an audio application developer, supporting audio units lets you leverage the library of existing audio units (both third-party and Apple-supplied) to extend the capabilities of your application.

Apple ships a number of audio units to accomplish common tasks, such as filtering, delay, reverberation, and mixing, as well as units to represent input and output devices (for example, units that allow audio data to be transmitted and received over a network). See "System-Supplied Audio Units" (page 43) for a listing of the audio units that ship with Mac OS X v10.4.

For a simple way to experiment with audio units, see the AU Lab application, available in the Xcode Tools install at `/Developer/Applications/Audio`. AU Lab allows you to mix and match various audio units. In this way, you can build a signal chain from an audio source through an output device.

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, you do not even have to interact directly with the HAL. Apple supplies a special audio unit, called the AUHAL, which allows you to pass audio data directly from another audio unit to a hardware device. Similarly, input coming from a hardware device is also routed through the AUHAL to be made available to subsequent audio units, as shown in Figure 1-3.

Figure 1-3   Hardware input through the HAL and the AUHAL unit

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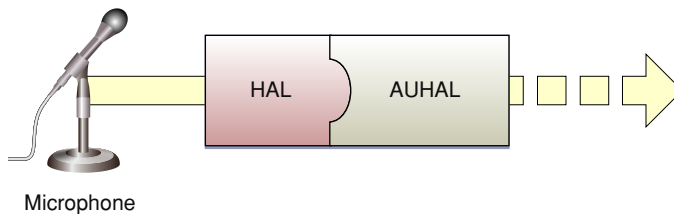
The AUHAL also takes care of any data conversion or channel mapping required to translate audio data between audio units and hardware.

MIDI Support

Core MIDI is the part of Core Audio that supports the MIDI protocol. Core MIDI allows applications to communicate with MIDI devices such as keyboards and guitars. Input from MIDI devices can be either stored as MIDI data or translated through an instrument unit into an audio signal. Applications can also output information 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 I/O. Core Audio also supports a music player programming interface that you can use to play MIDI data.

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

The Audio MIDI Setup Application

When using audio in Mac OS X, both users and developers can use the Audio MIDI Setup application to configure audio and MIDI settings. You can use Audio MIDI Setup to:

■ Specify the default audio input and output devices.

■ Configure properties for input and output devices, such as the 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 32).)

■ Configure MIDI networks and MIDI devices.

You can find the Audio MIDI Setup application in the `/Applications/Utilities` folder in Mac OS X v10.2 and later.

A Core Audio Recording Studio

Although Core Audio encompasses much more than recording and playing back audio, it can be useful to compare its capabilities to elements in a recording studio. A simple hardware-based recording studio may have a few instruments with some effect units feeding into a mixing deck, along with audio from a MIDI keyboard, as shown in Figure 1-4. The mixer can output the signal to studio monitors as well as a recording device, such as a tape deck (or perhaps a hard drive).

Figure 1-4 A simple recording studio
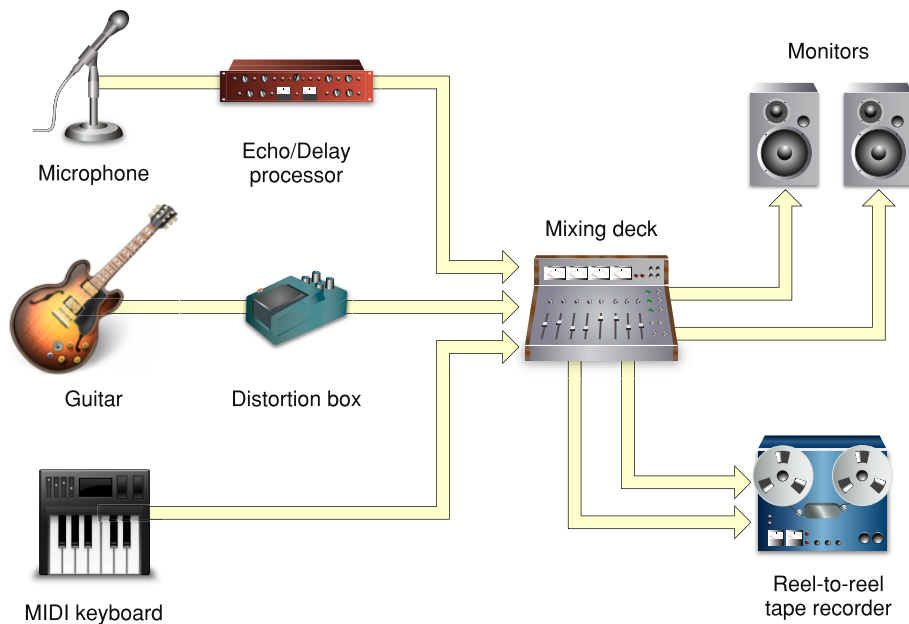
Much of the hardware used in a studio can be replaced by software-based equivalents. Specialized music studio applications can record, synthesize, edit, mix, process, and play back audio. They can also record, edit, process, and play back MIDI data, interfacing with both hardware and software MIDI instruments. In Mac OS X, applications rely on Core Audio services to handle all of these tasks, as shown in Figure 1-5.

Figure 1-5   A Core Audio "recording studio"
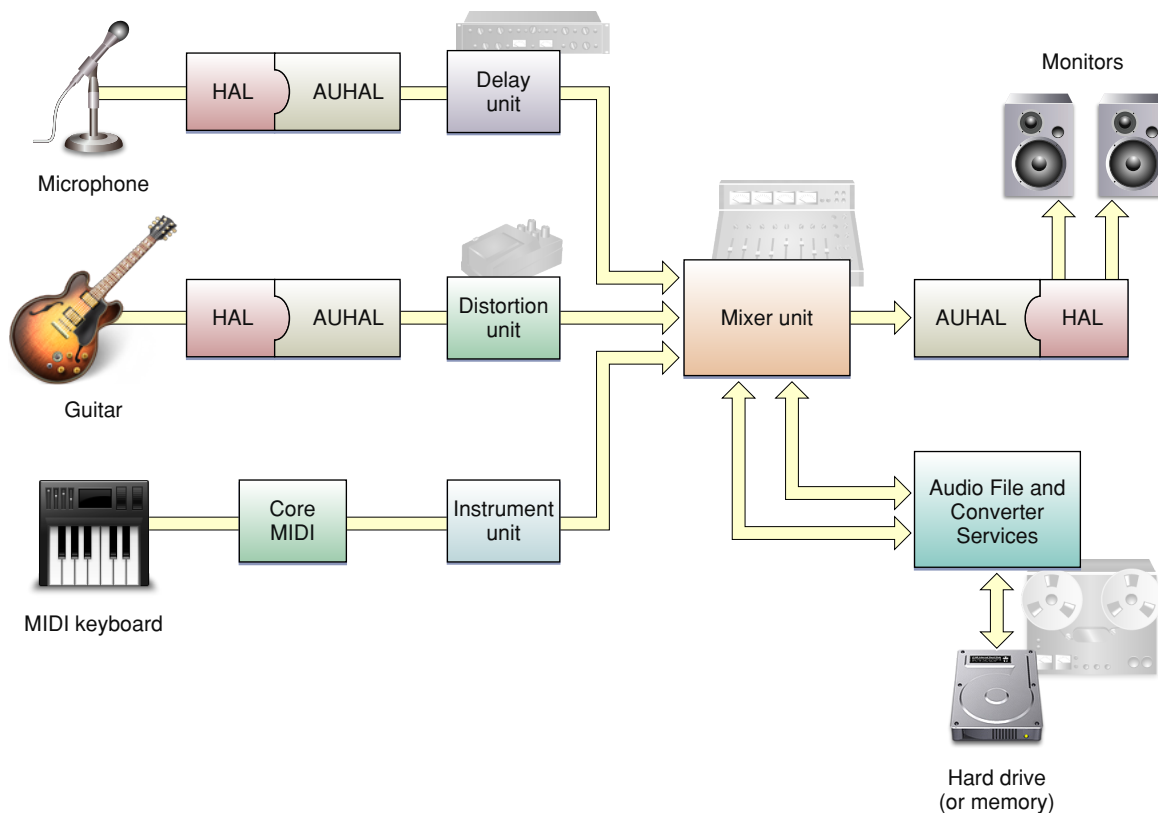
As you can see, audio units make up much of the signal chain. Other Core Audio interfaces provide application-level support, allowing applications to obtain audio or MIDI data in various formats and output it to files or output devices. "Core Audio Programming Interfaces" (page 17) discusses the constituent interfaces of Core Audio in more detail.

However, Core Audio lets you do much more than mimic a recording studio on a computer. You can use Core Audio for everything from playing simple system sounds to creating compressed audio files to providing an immersive sonic experience for game players.

Development Using the Core Audio SDK

To assist audio developers, Apple supplies a software development kit (SDK) for Core Audio. 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`)

- Host applications that load and manipulate audio units. (`AudioUnitHosting`.) Note that for the actual testing of audio units, you should use the AU Lab application mentioned in "Audio Units" (page 10).

- 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. 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++ framework as sample code to assist audio unit development. Feel free to modify 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`.

Core Audio Programming Interfaces

Core Audio is a comprehensive set of services for handling all audio tasks in Mac OS X, and as such it contains many constituent parts. This chapter describes the various programming interfaces of Core Audio.

For the purposes of this document, an **API** refers to a programming interface defined by a single header file, while a **service** is an interface defined by several header files.

For a complete list of Core Audio frameworks and the headers they contain, see "Core Audio Frameworks" (page 39).

Audio Unit Services

Audio Unit Services allows you to create and manipulate audio units. This interface consists of the functions, data types, and constants found in the following header files in `AudioUnit.framework` and `AudioToolbox.framework`:

- `AudioUnit.h`
- `AUComponent.h`
- `AudioOutputUnit.h`
- `AudioUnitParameters.h`
- `AudioUnitProperties.h`
- `AudioUnitCarbonView.h`
- `AUCocoaUIView.h`
- `MusicDevice.h`
- `AudioUnitUtilities.h` (in `AudioToolbox.framework`)

Audio units are plug-ins, specifically Component Manager components, for handling or generating audio signals. Multiple instances of the same audio unit can appear in the same host application. They can appear virtually anywhere in an audio signal chain.

An audio unit must support the noninterleaved 32-bit floating-point linear PCM format to ensure compatibility with units from other vendors. It may also support other linear PCM variants. Currently audio units do not support audio formats other than linear PCM. To convert audio data of a different format to linear PCM, you can use an audio converter (see "Audio Converters and Codecs" (page 21).

> **Note:** Audio File and Converter Services uses Component Manager components to handle custom file formats or data conversions. However, these components are not audio units.

Host applications must use Component Manager calls to discover and load audio units. Each audio unit is uniquely identified by a combination of the Component Manager type, subtype, and manufacturer's code. The type indicates the general purpose of the unit (effect unit, generator unit, and so on). The subtype is an arbitrary value that uniquely identifies an audio unit of a given type by a particular manufacturer. For example, if your company supplies several different effect units, each must have a distinct subtype to distinguish them from each other. Apple defines the standard audio unit types, but you are free to create any subtypes you wish.

Audio units describe their capabilities and configuration information using **properties**. Properties are key-value pairs that describe non-time varying characteristics, such as the number of channels in an audio unit, the audio data stream format it supports, the sampling rate it accepts, and whether or not the unit supports a custom Cocoa view. Each audio unit type has several required properties, as defined by Apple, but you are free to define additional properties based on your unit's needs. Host applications can use property information to create user interfaces for a unit, but in many cases, more sophisticated audio units supply their own custom user interfaces.

Audio units also contain various **parameters**, the types of which depend on the capabilities of the audio unit. Parameters typically represent settings that are adjustable in real time, often by the end user. For example, a parametric filter audio unit may have parameters determining the center frequency and the width of the filter response, which may be set using the user interface. An instrument unit, on the other hand, uses parameters to represent the current state of MIDI or event data.

A signal chain composed of audio units typically ends in an output unit. An output unit often interfaces with hardware (the AUHAL is such an output unit, for example), but this is not a requirement. Output units differ from other audio units in that they are the only units that can start and stop data flow independently. Standard audio units rely on a "pull" mechanism to obtain data. Each audio unit registers a callback with its successor in the audio chain. When an output unit starts the data flow (triggered by the host application), its render function calls back to the previous unit in the chain to ask for data, which in turn calls its predecessor, and so on.

Host applications can combine audio units in an audio processing graph to create larger signal processing modules. Combining units in a processing graph automatically creates the callback links to allow data flow through the chain. See "Audio Processing Graph API" (page 19) for more information.

To monitor changes in the state of an audio unit, applications can register callbacks ("listeners") that are invoked when particular audio unit events occur. For example, an application might want to know when a parameter changes value or when the 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 more detailed information about building audio units using the SDK, see the *Audio Unit Programming Guide*.

Audio Processing Graph API

The Audio Processing Graph API lets audio unit host application developers create and manipulate audio processing graphs. The Audio Processing Graph API consists of the functions, data types, and constants defined in the header file `AUGraph.h` in `AudioToolbox.framework`.

An audio processing graph (sometimes called an AUGraph) defines a collection of audio units strung together to perform a particular task. This arrangement lets you create modules of common processing tasks that you can easily add to and remove from your signal chain. For example, a graph could connect several audio units together to distort a signal, compress it, and then pan it to a particular location in the soundstage. You can end a graph with the AUHAL to transmit the sound to a hardware device (such as an amplifier/speaker). Audio processing graphs are useful for applications that primarily handle signal processing by connecting audio units rather than implementing the processing themselves. Figure 2-1 shows a simple audio processing graph.

Figure 2-1   A simple audio processing graph
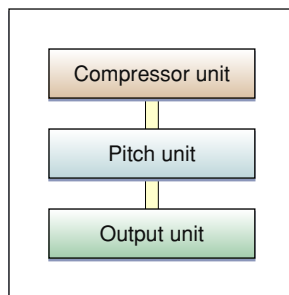
Each audio unit in an audio processing graph is called a **node**. You make a connection by attaching an output from one node to the input of another. You can't connect an output from one audio unit to more than one audio unit input unless you use a splitter unit, as shown in Figure 2-2. However, an audio unit may contain multiple outputs or inputs, depending on its type.

Figure 2-2   Incorrect and correct ways to fan out a connection
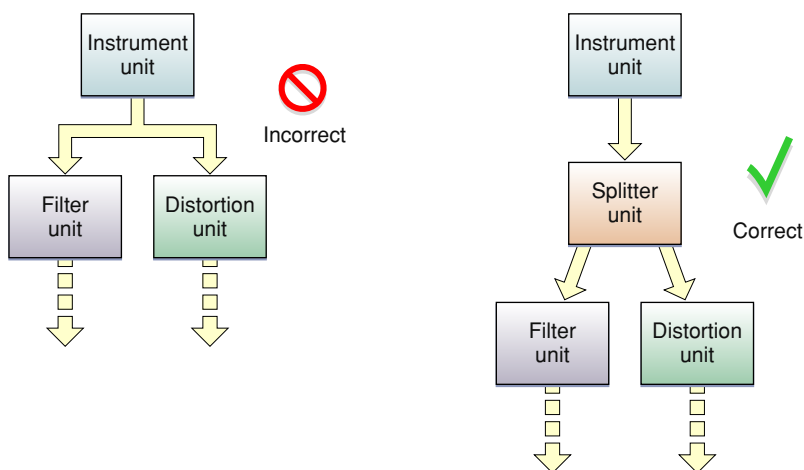
You can use the Audio Processing Graph API to combine subgraphs into a larger graph, where a subgraph appears as a single node in the larger graph, as shown in Figure 2-3.

Figure 2-3   A subgraph within a larger audio processing graph
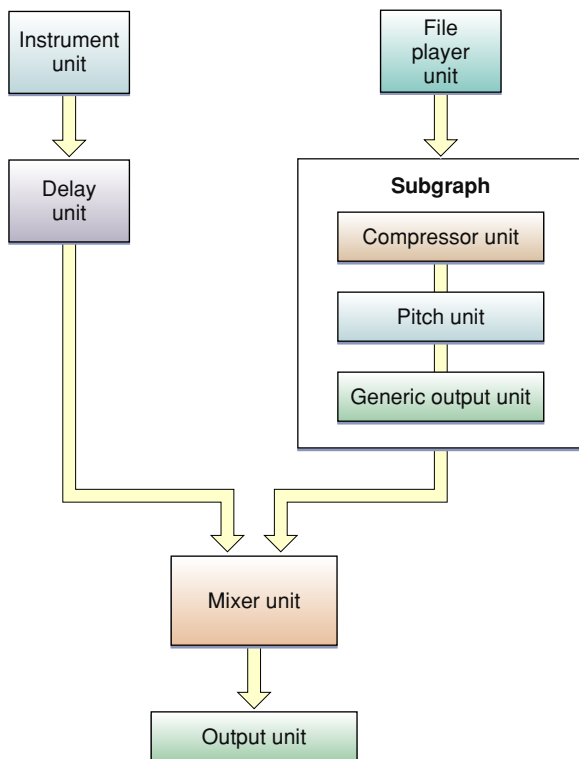
Each graph or subgraph must end in an output audio unit. In the case of a subgraph, the signal path should end with the generic output unit, which does not connect to any hardware.

While it is possible to link audio units programmatically without using an audio processing graph, you can modify a graph dynamically, allowing you to change the signal path while processing audio data. In addition, because a graph represents simply an interconnection of audio units, you can create and modify a graph without having to instantiate the audio units it references.

Audio File and Converter Services

Audio File and Converter Services lets you read or write audio data, either to a file or to a buffer, and allows you to convert the data between any number of different formats. This service consists of the functions, data types, and constants defined in the following header files in `AudioToolbox.framework` and `AudioUnit.framework`:

- `ExtendedAudioFile.h`
- `AudioFile.h`
- `AudioFormat.h`
- `AudioConverter.h`

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

- `CAFFile.h`

In many cases, you use the Extended Audio File API, 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" (page 47) 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, and an AC-3 file can be decoded only to a stereo data stream (not 5.1 surround).

If you need more control over the file reading, file writing, or data conversion process, you can access the Audio File and Audio Converter APIs directly (in `AudioFile.h` and `AudioConverter.h`). When using the Audio File API, the audio data source (as represented by an audio file object) can be either an actual file or a buffer in memory. In addition, if your application reads and writes proprietary file formats, you can handle the format translation using custom Component Manager components that the Audio File API can discover and load. For example, if your file format incorporates DRM, you would want to create a custom component to handle that process.

Audio Converters and Codecs

An audio converter lets you convert audio data from one format to another. For example, you can make simple conversions such as changing the sample rate and interleaving or deinterleaving audio data streams, to more complex operations such as compressing or decompressing audio. Three types of conversions are possible:

- Decoding an audio format (such as AAC (Advanced Audio Coding)) to linear PCM format.

- Encoding linear PCM data into a different audio format.

- Translating between different variants of linear PCM (for example, converting 16-bit signed integer linear PCM to 32-bit floating point linear PCM).

The Audio Converter API lets you create and manipulate audio converters. You can use the API with many built-in converters to handle most common audio formats. You can instantiate more than one converter at a time, and specify the converter to use when calling a conversion function. Each audio converter has properties that describe characteristics of the converter. For example, a channel mapping property also allows you to specify how the input channels should map to the output channels.

You convert data by calling 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.

An audio codec is a Component Manager component loaded by an audio converter to encode or decode a specific audio format. Typically a codec would decode to or encode from linear PCM. The Audio Codec API provides the Component Manager interface necessary for implementing an audio codec. After you create a custom codec, then you can use an audio converter to access it. "Supported Audio File and Data Formats" (page 47) lists standard Core Audio codecs for translating between compressed formats and Linear PCM.

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.

File Format Information

In addition to reading, writing, and conversion, Audio File and Converter Services can obtain useful information about file types and the audio data a file contains. For example, you can obtain data such as the following using the Audio File API:

■   File types that Core Audio can read or write

■   Data formats that the Audio File API can read or write

■   The name of a given file type

■   The file extension(s) for a given file type

The Audio File API also allows you to set or read properties associated with a file. Examples of properties include the data format stored in the file and a CFDictionary containing metadata such as the genre, artist, and copyright information.

Audio Metadata

When handling audio data, you often need specific information about the data so you know how to best process it. The Audio Format API (in `AudioFormat.h`) allows you to query information stored in various audio structures. For example, you might want to know some of the following characteristics:

■   Information associated with a particular channel layout (number of channels, channel names, input to output mapping).

■   Panning matrix information, which you can use for mapping between channel layouts.

■   Sampling rate, bit rate, and other basic information.

In addition to this information, you can also use the Audio Format API to obtain specific information about the system related to Core Audio, such as the audio codecs that are currently available.

Core Audio File Format

Although technically not a part of the Core Audio programming interface, the Core Audio File format (CAF) is a powerful and flexible file format , defined by Apple, for storing audio data. CAF files have no size restrictions (unlike AIFF, AIFF-C, and WAVE files) and can support a wide range of metadata, such as channel information and text annotations. The CAF format is flexible enough to contain any audio data format, even formats that do not exist yet. For detailed information about the Core Audio File format, see *Apple Core Audio Format Specification 1.0*.

Hardware Abstraction Layer (HAL) Services

Core Audio 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 defined 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 and output units, see "Interfacing with Hardware Devices" (page 30).

Music Player API

The Music Player API allows you to arrange and play a collection of music tracks. It consists of the functions, data types, and constants defined in the header file `MusicPlayer.h` in `AudioToolbox.framework`.

A particular stream of MIDI or event data is a **track** (represented by the `MusicTrack` type). Tracks contain a series of time-based events, which can be MIDI data, Core Audio event data, or your own custom event messages. A collection of tracks is a **sequence** (type `MusicSequence`). A sequence always contains an additional tempo track, which synchronizes the playback of all tracks in the sequence. Your application can add, delete, or edit tracks in a sequence dynamically. Each sequence must be assigned to a corresponding **music player** object (type `MusicPlayer`), which acts as the overall controller for all the tracks in the sequence.

A track is analogous to sheet music for an instrument, indicating which notes to play and for how long. A sequence is similar to a musical score, which contains notes for multiple instruments. Instrument units or external MIDI devices represent the musical instruments, while the music player is similar to the conductor who keeps all the musicians coordinated.

Track data played by a music player can be sent to an audio processing graph, an external MIDI device, or a combination of the two. The audio processing graph receives the track data through one or more instrument units, which convert the event (or MIDI) data into actual audio signals. The music player automatically communicates with the graph's output audio unit or Core MIDI to ensure that the audio output is properly synchronized.

Track data does not have to represent musical information. For example, special Core Audio events can represent changes in audio unit parameter values. A track assigned to a panner audio unit might send parameter events to alter the position of a sound source in the soundstage over time. Tracks can also contain proprietary user events that trigger an application-defined callback.

For more information about using the Music Player API to play MIDI data, see "Handling MIDI Data" (page 35).

Core MIDI Services and MIDI Server Services

Core Audio uses Core MIDI Services for MIDI support. These services consist of the functions, data types, and constants defined 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, and it is the primary conduit for interacting with Core Audio services. For example, you can associate endpoints with tracks used by the Music Player API, allowing you to record or play back MIDI data. A MIDI endpoint is a logical representation of 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 2-4 shows the relationship between Core MIDI and MIDI Server.

Figure 2-4   Core MIDI and Core MIDI Server
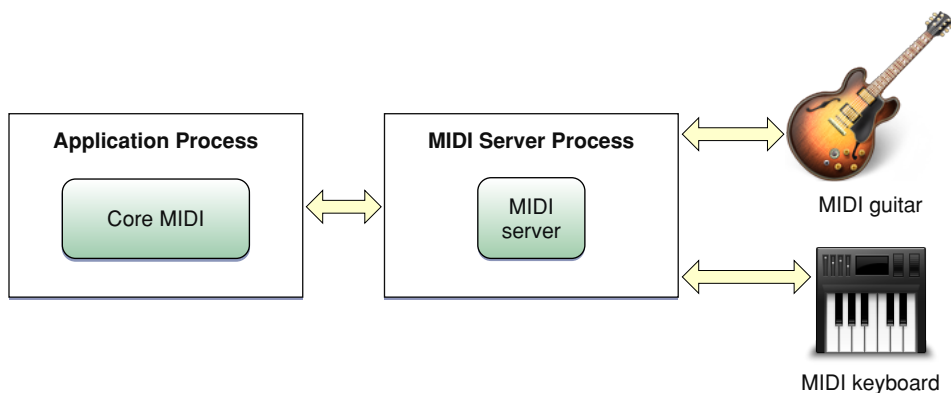
In addition to providing an application-agnostic base for MIDI communications, MIDI Server also handles any MIDI thru connections, which allows device-to device chaining without involving the host application.

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 2-5 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 2-5   MIDI Server interface with I/O Kit
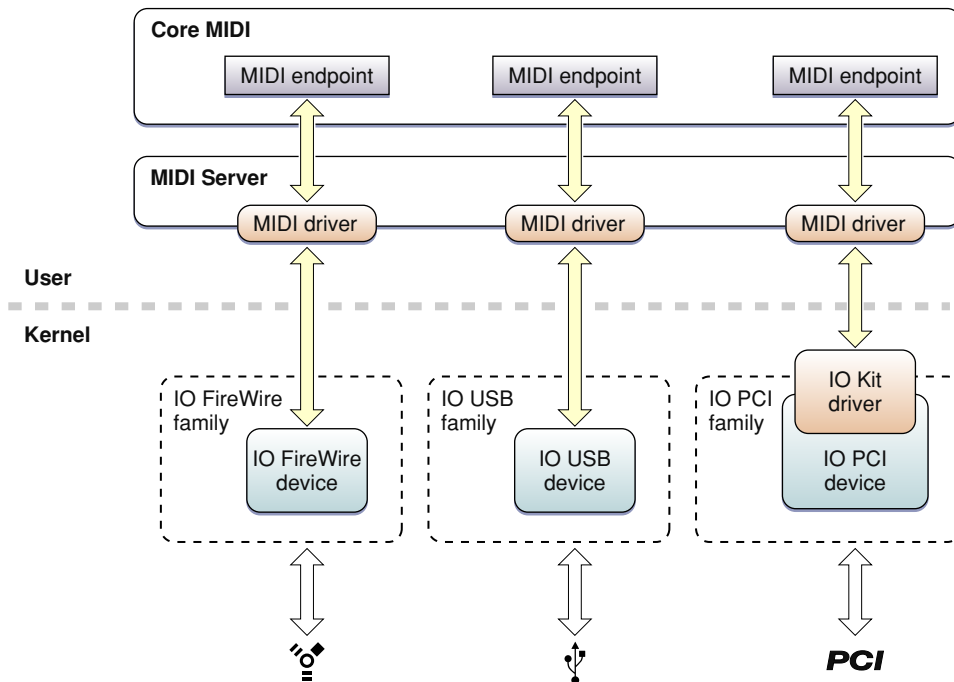
The drivers for each MIDI device generally exist outside the kernel, running in the MIDI Server process. These drivers interact with the default I/O Kit drivers for the underlying protocols (such as USB and FireWire). The MIDI drivers are responsible for presenting the raw device data to Core MIDI in 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.

Core Audio Clock API

The Core Audio Clock API, as defined in the header file `CoreAudioClock.h` in `AudioToolbox.framework`, provides a reference clock that you can use to synchronize applications or 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. The Core Audio Clock API also contains utilit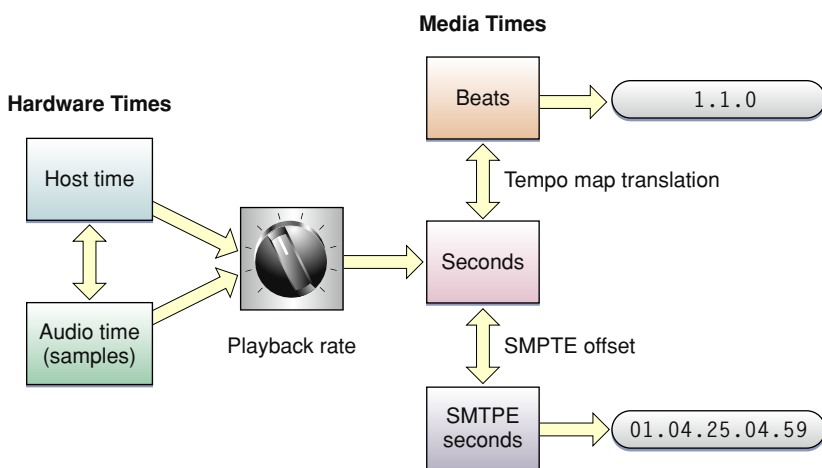y functions that convert one time format to another and that display bar-beat or SMPTE times. Figure 2-6 shows the interrelationship between various Core Audio Clock formats.

Figure 2-6   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 Figure 2-6 (page 26) 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.

OpenAL (Open Audio Library)

Core Audio includes a Mac OS X implementation of the open-source OpenAL specification. OpenAL is a cross-platform API used to position and manipulate sounds in a simulated three-dimensional space. For example, you can use OpenAL for positioning and moving sound effects in a game, or creating a sound space for multichannel audio. In addition to simple positioning sound around a listener, you can also add distancing effects through a medium (such as fog or water), doppler effects, and so on.

The OpenAL coding conventions and syntax were designed to mimic OpenGL (only controlling sound rather than light), so OpenGL programmers should find many concepts familiar.

For an example of using OpenAL in Core Audio, see `Services/OpenALExample` in the Core Audio SDK. For more details about OpenAL, including programming information and API references, see openal.org.

System Sound API

The System Sound API provides a simple way to play standard system sounds in your application. Its header, `SystemSound.h` is the only Core Audio header located in a non-Core Audio framework. It is located in the `CoreServices/OSServices` framework.

For more details about using the System Sound API, see Technical Note 2102: The System Sound APIs for Mac OS X v10.2, 10.3, and Later.

An Overview of Common Tasks

This chapter describes some basic scenarios in which you can use Core Audio. These examples show how you would combine parts of Core Audio to accomplish some common tasks.

Note that Core Audio is extremely modular, with few restrictions on how to use its various parts, so you may choose to implement certain capabilities in other ways than those shown here. For example, your application can call Audio File and Converter Services functions directly to read data from a file and convert it to linear PCM, or you can choose to encapsulate that capability as a standalone generator audio unit that your application can discover and load.

Reading and Writing Audio Data

Many applications that handle audio need to read and write the data, either to disk or to a buffer. In most cases, you will want to read the file data and convert it to linear PCM (for use in audio units and so on). You can do so in one step using the Extended Audio File API.

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

Figure 3-1   Reading audio data
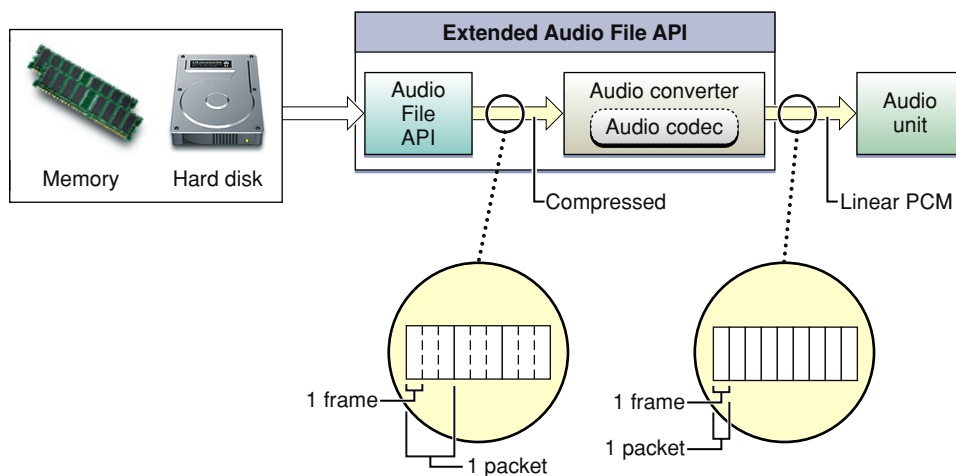
If you need more control over the file reading and conversion procedure, you can call Audio File or Audio Converter functions directly. You use the Audio File API to read the file from disk or a buffer. This data may be in a compressed format, in which case it can be converted to linear PCM using an 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 the Audio Converter API to create an audio converter object, specifying the input and output formats you desire. Each format is defined in a

`AudioStreamBasicDescription` structure, which is a fundamental data type in Core Audio for describing audio data. As stated earlier, if you are converting to linear PCM, you can use Extended Audio File API calls to go from audio file data to linear PCM in one step, without having to create your own audio converter.

Once converted to linear PCM, the data is ready to process by an audio unit. To connect with an audio unit, you must register a callback with a particular audio unit input. When the audio unit needs input, it will invoke your callback, which can then supply the necessary audio data.

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

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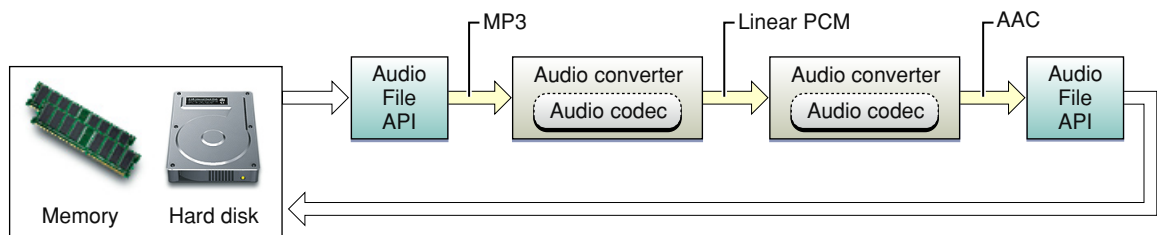
For examples of using the Audio File and Audio Converter APIs, see the `SimpleSDK/ConvertFile` and `Services/AudioFileTools` samples in the Core Audio SDK. If you are interested in writing a custom audio converter codec, see the samples in the `AudioCodec` folder.

Interfacing with Hardware Devices

Most audio applications have to connect with external hardware, either to output audio data (for example, to an amplifier/speaker) or to obtain it (such as through a microphone). These operations must go through the hardware abstraction layer (HAL). Fortunately, in most cases you do not need to write custom code to access the HAL. Apple provides three standard audio units that should address most hardware needs: the default output unit, the system output unit, and the AUHAL. Your application must call the Component Manager to discover and load these units before you can use them.

Default and System Output Units

The default output unit and system output 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 to one of these output devices (such as in an audio processing graph), your unit's callback function (sometimes called the render callback) is called when the output needs data. The output 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 output unit
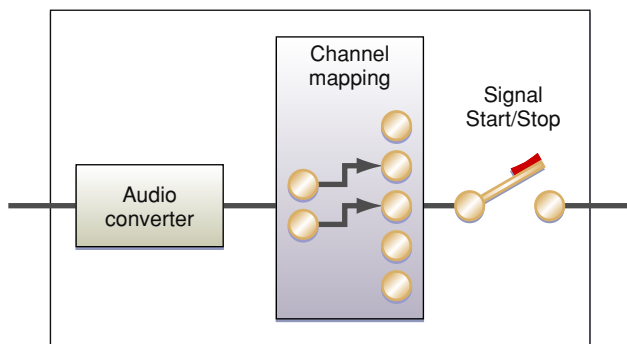
■ Any required linear PCM data conversion. The output unit contains an audio converter that can translate your audio data to the linear PCM variant required by the hardware.

■ 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

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 32) 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
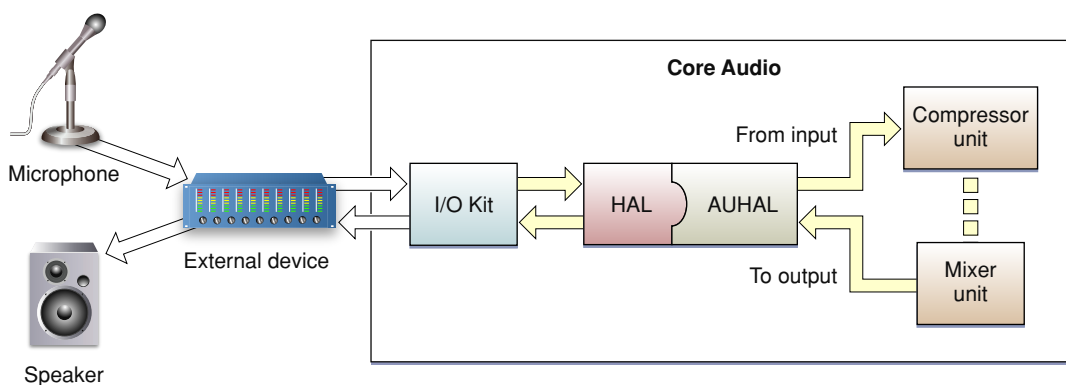
An audio signal coming in through the external device is translated into an audio data stream and passed to the AUHAL, which can then send it on to another audio unit. After processing the data (for example, adding effects, or mixing with other audio data), the output is sent back to the AUHAL, which can then output the audio through the same external device.

For examples of using the AUHAL for input and output, see the *SimplePlayThru* and *CAPlayThrough* code samples in the ADC Reference Library. `SimplePlayThru` shows how to handle input and output through a single AUHAL instance. `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 subdevices 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 subdevice'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 subdevices. If the user removes a subdevice (or configures it in an incompatible manner), those channels disappear from the aggregate, but those channels will reappear when the subdevice is reattached or reconfigured.

Aggregate devices have some limitations:

■ All the subdevices 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 subdevices 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" (page 43) 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 `AUGraphDispose`. 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
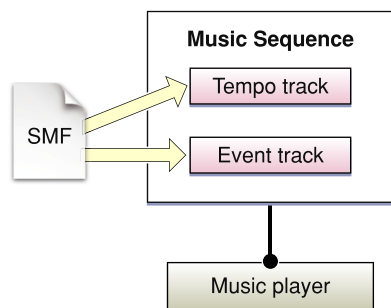
Depending on the type of MIDI file and the flags set when loading a file, you can store all the MIDI data in single track, or store each MIDI channel as a separate track in the sequence. By default, each MIDI channel is mapped sequentially to a new track in the sequence. For example, if the MIDI data 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
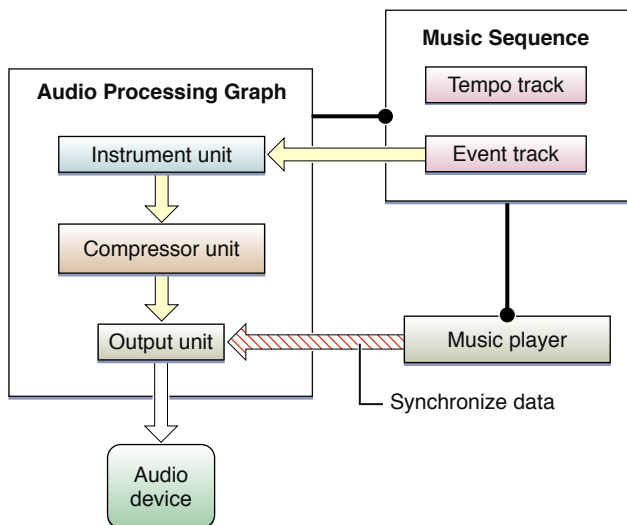
The sequence must be associated with a particular audio processing graph, and the tracks in the sequence can be assigned to one or more instrument units in the graph. (If you don't specify a track mapping, the music player sends all the MIDI data to the first instrument unit it finds in the graph.) The 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
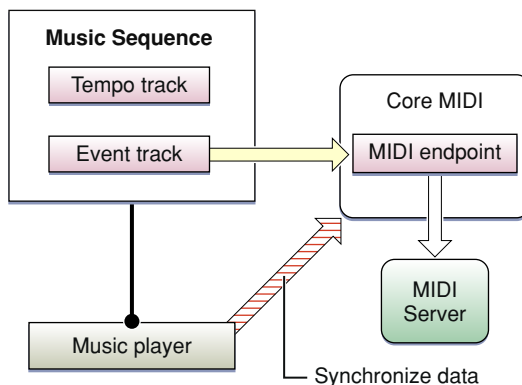
A sequence of tracks can be assigned to a combination of instrument units and MIDI devices. For example, you can assign some of the tracks to play through an instrument unit, while other tracks go through Core MIDI to play through external MIDI devices, as shown in Figure 3-8.

Figure 3-8 Playing both MIDI devices and a virtual instrument
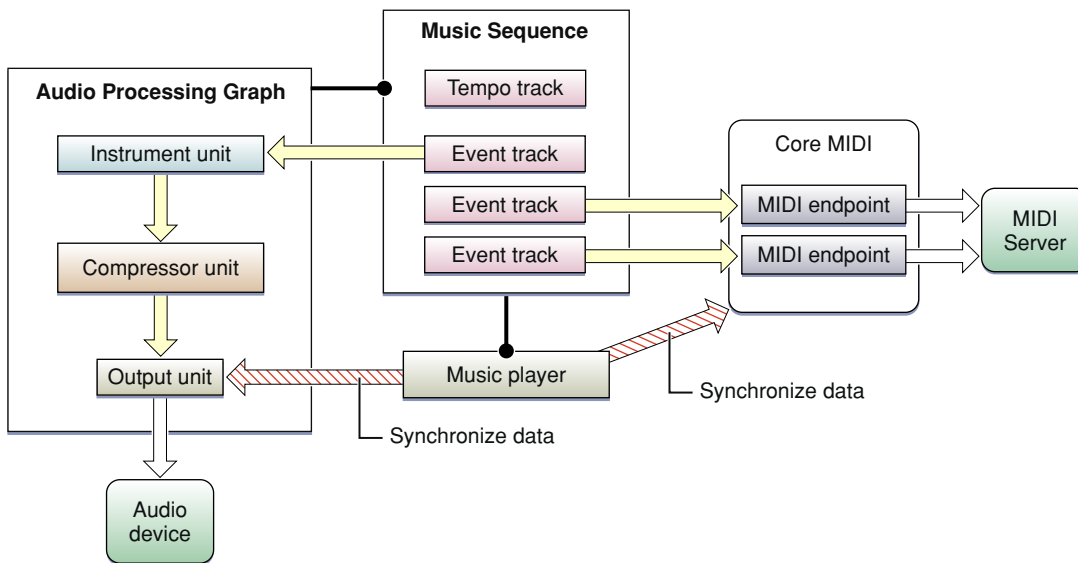
The music player automatically coordinates between the audio processing graph's output unit and Core MIDI to ensure that the outputs are synchronized.

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
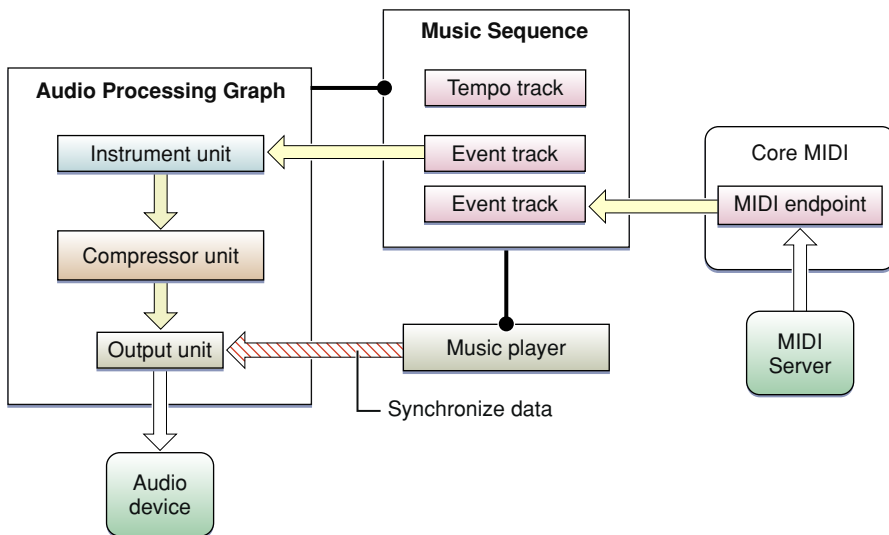
The playback of existing data is handled as usual through the audio processing graph, which sends audio data to the output unit. New data from an external MIDI device enters through Core MIDI and is transferred through the assigned endpoint. Your application must iterate through this incoming data and 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 Both Audio and MIDI Data

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
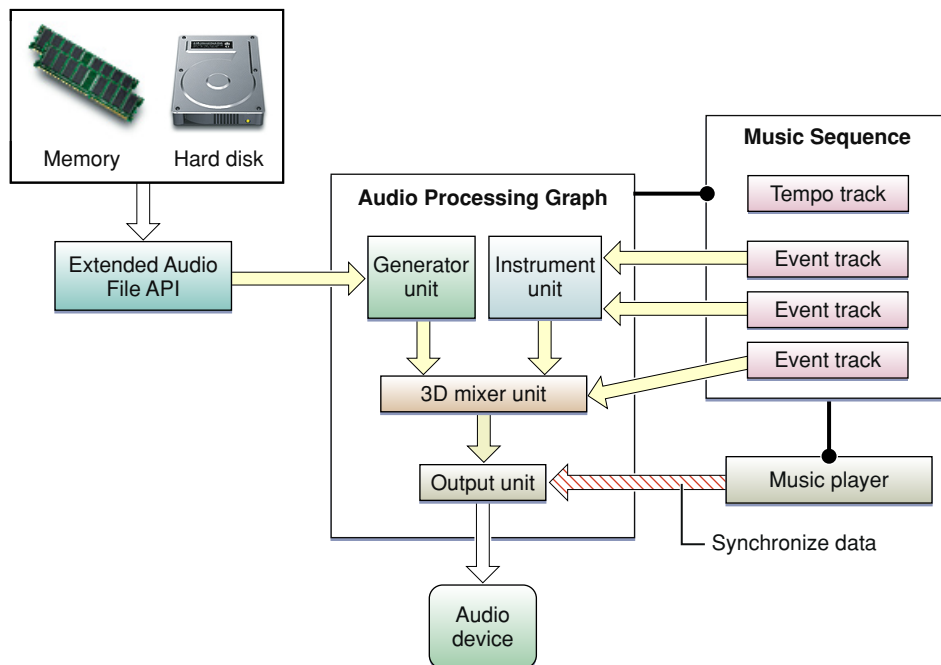
The soundtrack audio data is retrieved from disk or memory and converted to linear PCM using the Extended Audio File API. The MIDI data, stored as tracks in a music sequence, is sent to a virtual instrument unit. The output from the virtual instrument unit is in linear PCM format and can then be combined 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.

AudioToolbox.framework

The Audio Toolbox framework contains the APIs that provide application-level services:

- `AudioToolbox.h`: Top-level include file for the Audio Toolbox framework.
- `AudioConverter.h`: Audio Converter API. Defines the interface used to create and use audio converters.
- `AudioFile.h`: Audio File API. Defines the interface used to read and write audio data in files.
- `ExtendedAudioFile.h`: Extended Audio File API. Defines the interface used to translate audio data from files directly into linear PCM, and vice versa.
- `AudioFormat.h`: Audio Format API. Defines the interface used to assign and read audio format metadata in audio files.
- `CoreAudioClock.h`: The Core Audio Clock interface lets you designate a timing source for synchronizing applications or devices.
- `MusicPlayer.h`: Music Player API. Defines the interface used to manage and play event tracks in music sequences.
- `AUGraph.h`: Audio Processing Graph API. Defines the interface used to create and use audio processing graphs.
- `DefaultAudioOutput.h`: Deprecated: Do not use. Defines an older interface for accessing the default output unit (deprecated in Mac OS X v10.3 and later).
- `AudioUnitUtilities.h`: Some 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.
- `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.
- `CAFFile.h`: Defines the Core Audio file format, which provides many advantages over other audio file formats. See *Apple Core Audio Format Specification 1.0* for more information.
- `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.

AudioUnit.framework

The Audio Unit framework contains the APIs used specifically for audio units and audio codecs.

- `AudioUnit.h`: Top-level include file for the Audio Unit framework.
- `AUComponent.h`: Defines the basic interface for audio unit components.
- `AudioCodec.h`: Defines the interface used specifically for creating audio codec components..
- `AudioOutputUnit.h`: Defines the interface used to turn an output unit on or off.
- `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.
- `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.
- `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`.
- `AUNTComponent.h`: Deprecated: Do not use. Defines the interface for older "v1" audio units. Deprecated in Mac OS X v10.3 and later. Replaced by `AUComponent.h`.
- `MusicDevice.h`: An interface for creating instrument units (that is, software-based music synthesizers).

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.

CoreAudio.framework

The Core Audio framework contains lower-level APIs used to interact with hardware, as well as data types common to all Core Audio services. This framework contains all the APIs that make up Hardware Abstraction Layer (HAL) Services.

- `CoreAudio.h`: Top-level include file for the Core Audio framework.
- `CoreAudioTypes.h`: Defines data types used by all of Core Audio.
- `HostTime.h`: Contains functions to obtain and convert the host's time base.
- `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.

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.

OpenAL.framework

The OpenAL framework provides the Mac OS X implementation of the the OpenAL specification. For more details about OpenAL APIs, see openal.org.

- `al.h`
- `alc.h`
- `alctypes.h`
- `altypes.h`
- `alut.h`

System-Supplied Audio Units

The tables in this appendix list the audio units that ship with Mac OS X v10.4, grouped by Component Manager type. The Component Manager manufacturer identifier for all these units is `kAudioUnitManufacturer_Apple`.

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

Effect Units	Component Manager 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.
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.

Effect Units	Component Manager Subtype	Description
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 B-2   System-supplied instrument unit (`kAudioUnitType_MusicDevice`)

Instrument Unit	Component Manager 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 B-3   System-supplied mixer units (`kAudioUnitType_Mixer`)

Mixer Units	Component Manager 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.
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 B-4   System-supplied converter units (`kAudioUnitType_FormatConverter`)

Converter Unit	Component Manager 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, as described in "Audio Converters and Codecs" (page 21).
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 B-5   System-supplied output units (`kAudioUnitType_Output`)

Output Unit	Component Manager 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 Devices" (page 30) for more details.
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 Graph API" (page 19).

45

Output Unit	Component Manager Subtype	Description
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 B-6   System-supplied generator units (`kAudioUnitType_Generator`)

Generator Unit	Component Manager 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

This appendix describes the audio data and file formats supported in Core Audio as of Mac OS X v10.4.

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 C-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 C-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 Mac 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
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

Audio data type	Encode from linear PCM?	Decode to linear PCM?
µLaw 2:1 (`ulaw`)	Yes	Yes

APPENDIX C

Supported Audio File and Data Formats

Document Revision History

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

Date	Notes
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.