Core Audio

(Legacy)

Audio

2008-10-15

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

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction

Important: The information in this document is obsolete and should not be used for new development.

This document has been replaced by *Core Audio Overview* and other audio documents in the ADC Reference Library. For iPhone OS development, refer to *Getting Started with Audio & Video*. For Mac OS X development, refer to *Getting Started with Audio*.

About Core Audio

Core Audio presents a multitiered set of API services that developers can take advantage of in their applications. These range from low-level access to particular audio devices to sequencing and software-synthesis. The MIDI services present the capabilities of a MIDI device, which allow an application to interface to a device and manage and manipulate the MIDI data flow around the system.

These API services in Core Audio are presented in frameworks. A framework is a type of bundle that packages a dynamic shared library with the resources that the library requires, including header files. A framework bundle has an extension of `.framework`. Inside the bundle there can be multiple major versions of the framework.

The executable code in a framework is a dynamic shared library. Multiple, concurrently running programs can share the code in this library without requiring their own copy. As a packaging mechanism used by Mac OS X, frameworks present the runtime library that your application can run against, and the header files that you can use to link to.

Frameworks are implemented in C and C++ and present a C-based function API. There is also a Java API for these audio system services. The Java API primarily presents a corresponding C function or structure as a method on a Java class. There is as little overhead as possible in the interface of Java code to the underlying C implementation. Everything in the C interface you can accomplish using Java.

Because the Java API so closely follows the C API, if you are a Java developer, you need to understand the overall design of these frameworks in order to effectively use the provided services. The language choice is up to you, depending upon your development needs and requirements.

Additional Resources

Apple provides a number of resources available to assist developers. These include:

- The coreaudio-api mailing list: http://lists.apple.com/
- The developer website: http://developer.apple.com/audio
- Core Audio SDK: http://connect.apple.com

- Mac OS X development resources: http://developer.apple.com/macosx/

Core Audio Overview

This chapter will provide you with an understanding of the architecture of Core Audio, and how the various pieces fit together functionally.

Apple's Objectives

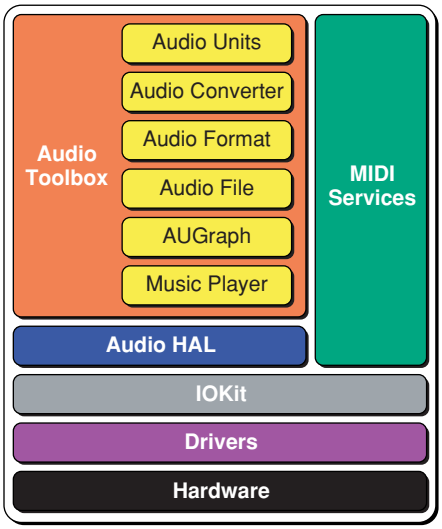
In creating Core Audio, Apple's objective in the audio space has been twofold. The primary goal is to deliver a high-quality, superior audio experience for Macintosh users. The second objective reflects a shift in emphasis from developers having to establish their own audio and MIDI protocols in their applications to Apple moving ahead to assume responsibility for these services on the Macintosh platform.

Some of the key features of the Core Audio architecture available in Mac OS X include:

- A flexible audio format
- Multichannel audio I/O
- Support for both PCM and non-PCM formats
- 32-bit floating point native-endian PCM as the canonical format
- Fully specifiable sample rates
- Multiple application usage of audio devices
- Application determined latency
- Ubiquity of timing information
- Both C and Java APIs

Figure 2-1 illustrates the Core Audio architecture in Mac OS X and its various building blocks.

Figure 2-1 The Core Audio Architecture



The theory of operation behind the Core Audio architecture is discussed in subsequent chapters of this document.

Introduction to Core Audio

Hardware Abstraction Layer (HAL)

Note: In its preliminary form, this document does not yet contain documentation for the Hardware Abstraction Layer. The final document will contain information on this technology.

The Hardware Abstraction Layer (HAL) is presented in the Core Audio framework and defines the lowest level of audio hardware access to the application. It presents the global properties of the system, such as the list of available audio devices. It also contains an `Audio Device` object that allows the application to read input data and write output data to an audio device that is represented by this object. It also provides the means to manipulate and control the device through a property mechanism.

The service allows for devices that use PCM encoded data. For PCM devices, the generic format is 32-bit floating point, maintaining a high resolution of the audio data regardless of the actual physical format of the device. This is also the generic format of PCM data streams throughout the Core Audio API.

An audio stream object represents n-channels of interleaved samples that correspond to a particular I/O end-point of the device itself. Some devices (for example, a card that has both digital and analog I/O) may present more than one audio stream.

The service provides the scheduling and user/kernel transitions required to both deliver and produce audio data to and from the audio device. Timing information is an essential component of this service; time stamps are ubiquitous throughout both the audio and MIDI system. This provides the capability to know the state of any particular sample (that is, “sample accurate timing”) of the device.

Audio Unit

An audio unit is a single processing unit that either is a source of audio data (for example, a software synthesizer), a destination of audio data (for example an audio unit that wraps an audio device), or both a source and destination (for example a DSP unit, such as a reverb, that takes audio data and processes or transforms this data).

The Audio Unit API uses a similar property mechanism as the Core Audio framework and use the same structures for both the buffers of audio data and timing information. Audio unit also provides real-time control capabilities, called parameters, that can be scheduled, allowing for changes in the audio rendering to be scheduled to a particular sample offset within any given "slice" of an audio unit's rendering process.

An application can use an `AudioOutputUnit` to interface to a device. The `DefaultOutputAudioUnit` tracks the selection of a device by the user as the "default" output for audio, and provides additional services such as sample rate conversion, to provide a simpler means of interfacing to an output device.

Audio Codec

Audio codecs are the encoders and decoders available to the system for audio compression and decompression. Using the Audio Codec API for conversion between audio formats is deprecated in favor of using the Audio Converter API, described in the "Audio Toolbox" (page 45) section.

Deploying an audio codec is performed by subclassing either the `ACBaseCodec` class or the `ACSimpleCodec` class, both provided in the Core Audio SDK. Once subclassed, the abstract methods (those set equal to zero) need to be provided, and the methods designated as virtual may be overridden as needed.

Audio Toolbox

This framework currently provides five primary services:

1. Audio Converter provides format conversion services. When encoding or decoding audio data, Audio Converter should be utilized, as it allows for many different type and format conversions. It also allows for conversions between linear PCM data and compressed audio data.

2. Audio Format is provided to help handle information about different audio formats. It is able to inspect `AudioStreamBasicDescription` instances to provide information about various aspects of an audio stream. Also, the Audio Format API can provide information about the encoders and decoders available on the system.

3. Audio File provides file services for dealing with creating, opening, modifying, and saving audio files. It features file-creation and format-specification capabilities, as well as reading and writing mechanisms and the ability to open files in the file system. Audio File uses a property system to keeps track of a file's file format, data format, channel layout, and more.

4. AUGraph allows for the construction and management of a signal processing graph of Audio Units, managing the connections and run-time state of the units that comprise a particular graph, including run-time management of inserting or removing nodes. The ubiquitous timing information in the signal chain deals with both feedback and fanning.

5. Music Sequence services provide a sequence object made up of one or more tracks of music events (both system-provided and user-defined). Track data can be edited while a sequence is playing, and its data can be iterated over. A music sequence typically addresses a graph of audio units, where tracks can be addressed to different nodes (audio units) of its graph, or a MIDI endpoint. A music player is responsible for the playing of a sequence.

MIDI Services

Note: In its preliminary form, this document does not yet contain documentation for MIDI Services. Please consult the Core Audio SDK, available from http://developer.apple.com/audio, for more information on developing MIDI Services.

This framework provides the representation of MIDI hardware and the interapplication communication of MIDI data to an application. The `MIDIDevice` object presents a MIDI-capable piece of hardware. A discrete MIDI source or destination (16 channels of MIDI data) is represented by the `MIDIEndpoint` object. This may be a real device or another application that is presented to your application as a virtual `MIDIEndpoint`, thus providing the interapplication communication of MIDI data.

The framework provides the I/O service and hosts the drivers that are supplied by both Apple and third-party companies to represent that hardware within the system.

Core Audio Types

Core Audio utilizes a series of structures and constants to encapsulate various pieces of information. `CoreAudioTypes.h` includes these structures, used consistently throughout Core Audio:

- `AudioBufferList` encapsulates buffer data.
- `AudioStreamBasicDescription` encapsulates formatting information.
- `AudioTimeStamp` holds time stamp information.
- `AudioChannelLayout` specifies the layout of an audio sample's layout.

In addition, many constants are declared, including channel layout constants, used in identifying the layout of audio sources, and format ID constants, useful when specifying the format of the audio data.

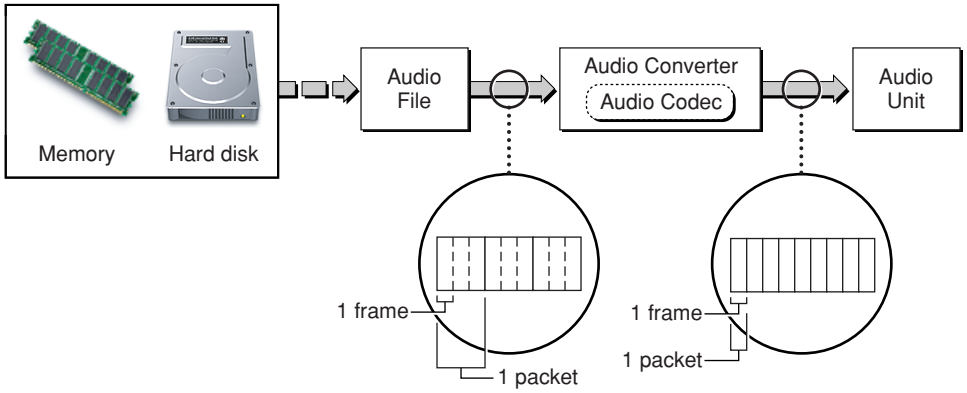
Using Core Audio

There are many tasks that you can accomplish with Core Audio. This section will outline the architecture of Core Audio, highlighting the various uses of Mac OS X's audio technology.

Audio Data Operations

One of the main functions of Core Audio is to work with and manipulate audio data, that is either stored on disk or already in memory. Effects can be applied to the data, and data sources mixed. Beyond that, Core Audio is also responsible for pulling data from input devices, and outputting data back out. Finally, data can be put back out to disk as a file, and may be converted to another format.

Figure 2-2 Reading in an audio file



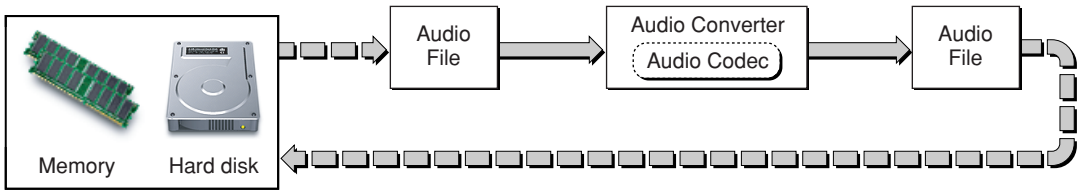
In order to use audio data from a file, it first must be read in. The Audio File API is provided for this purpose. An audio file instance can be created to act as a proxy for the file on disk, or for a buffer in memory (using callbacks).

Once the audio file has been created and bound to a file or memory, its data can be read in. If the data in the file is encoded, an audio converter is needed to convert the data into 32 bit floating-point Pulse Code Modulated (PCM) native-endian data, also known as the canonical format. Once the data is in this format, it is ready to be used in an audio unit or by another portion of Core Audio.

It is worth noting that an audio converter instance inherently uses the audio codecs available on the system. Using an audio codec directly for this kind of data conversion is discouraged, since the Audio Converter API takes care of the actual buffering and other considerations that need to be considered during a conversion.

To write a file back out to disk, simply reverse this process. Data output in the canonical format can be converted to an encoded format with an audio converter, and then saved to disk or memory via the Audio File API.

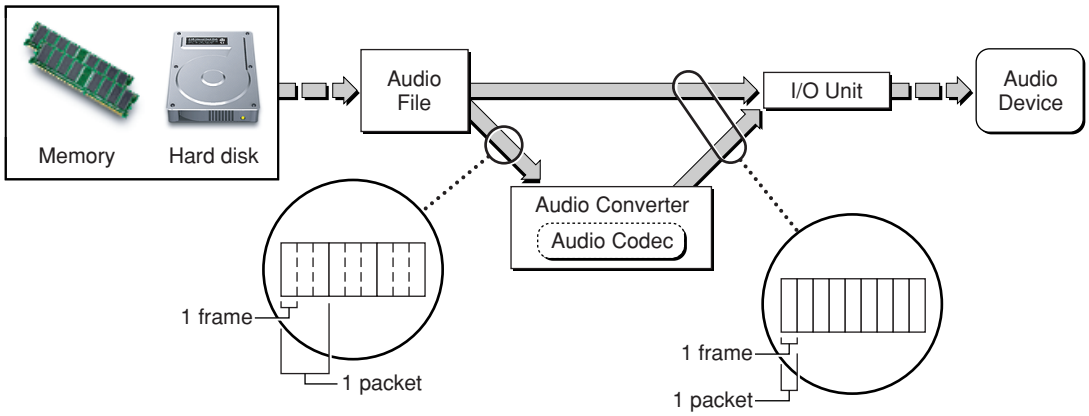
Figure 2-3 Converting audio files



Converting a file uses a process similar to the previous example. The Audio File API is used to open the file off of disk, an audio converter takes the incoming data and converts it to the desired format, and another audio file instance is used to save the data out to disk. Again, the codecs needed to decode the incoming

data and encode the outgoing data are used automatically by the converter; for instance, it is not necessary for you to read in the encoded data, convert it to the canonical format, and then encode it in the resulting format before writing it out. This service provided by the Audio Converter API.

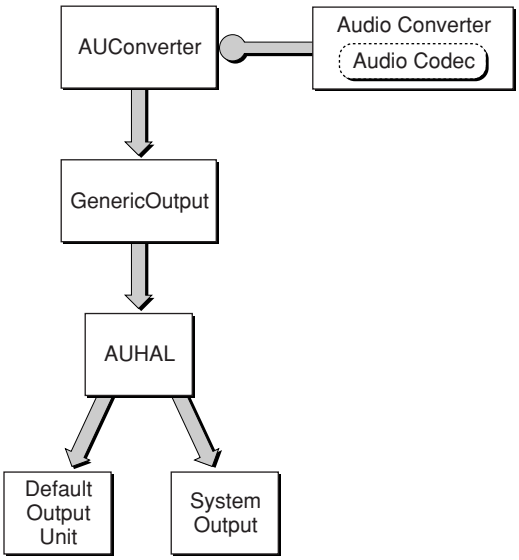
Figure 2-4 Playing back audio files



Playing back the contents of an audio file is one of the most common tasks that developers perform. In Core Audio, this is accomplished by reading in the data using an audio file instance. Once the instance is set up, an I/O unit instance can pull on the file, extracting the audio data and outputting it to the assigned audio device. If the data is encoded in the canonical format, no further decoding is needed to output the sound. If the data is encoded, it will need to be converted into the canonical format before it can be played back.

An I/O unit is a type of audio unit that acts as a proxy for an audio device. When data is sent to it, it will be relayed to the device that it represents. The most common use of this is to send data to the default output, as specified by the user. The unit to use in this case is the `Default Output` unit. A `System Output` unit is also provided, which is discussed is the next example.

Figure 2-5 I/O unit hierarchy

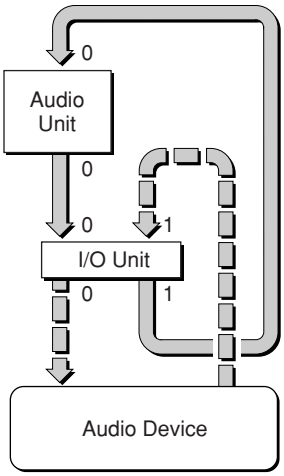


Each I/O unit inherits from `AUConverter`, an audio unit which owns an audio converter instance; this unit can be used in a graph to convert data between formats, sample rates, and the like. A `GenericOutput` unit implements adds the ability to start and stop the pulling of data to the output device.

When playing out to any piece of hardware, an `AUHAL` unit is needed. An instance of `AUHAL` can be attached to any audio device, making the instance a proxy for getting input and providing output to that device.

The `Default Output` unit is provided to play audio out to the user's prefered output, as designated in the System Preferences. Likewise, the `System Output` unit is provided to play back to the current system out device.

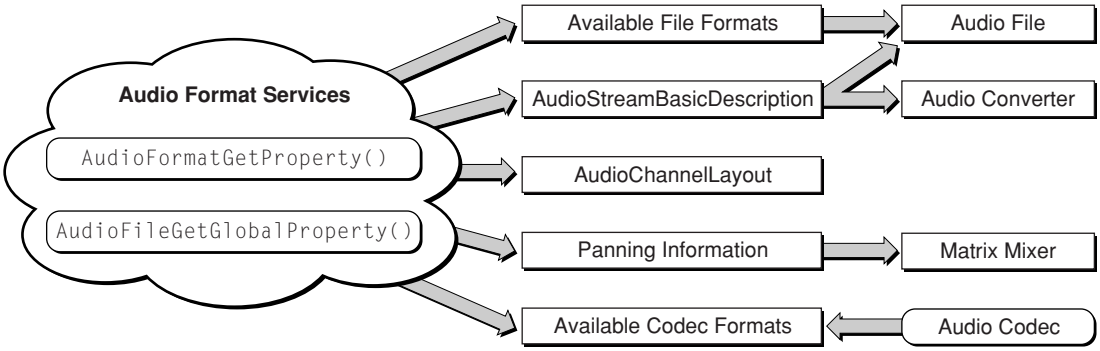
Figure 2-6 Using an I/O unit for input and output



Any of these I/O units may be used to pull input data from its associated audio device, through any number or combination of audio units or audio unit graphs, and output back through the I/O unit. The I/O unit itself has two busses: 0 and 1, where the 0 bus is designated as the output, and the 1 bus is the input bus. The connections between the output of the 0 bus and the audio device and the input of the 1 bus and the audio device are made when the unit is associated with the device.

To process data from a device and play it back through, simply associate the device with the unit, connect the output of the 1 bus with whatever audio units or graphs are being used to process the data, and connect the output of those units to the input of the 0 bus on the I/O unit. To start the render, tell the I/O unit to render. This, in turn, will cause the unit to ask the units attached to it to render, eventually leading back to the I/O unit's input bus, which will pull from the audio device. The data will pass through the input bus and will work its way through all of the attached units until it reaches the I/O units output bus, where it will automatically be output to the audio device.

Figure 2-7 Audio Format Services

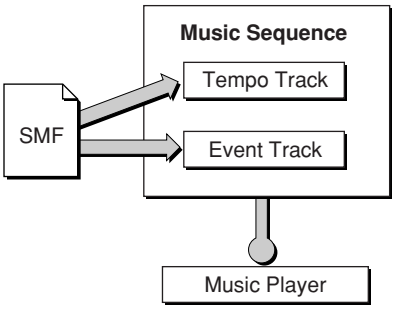


When working with streams of audio data, information about the data and the formats that the system has available become important. The Audio Format API provides a mechanism to get information about audio data, like the available codecs for encoding and decoding information, the encoding information for channel layouts, and panning information for use with the Matrix Mixer audio unit. Also, the Audio File API provides a function useful for determining the available file formats on the system.

MIDI Data Operations

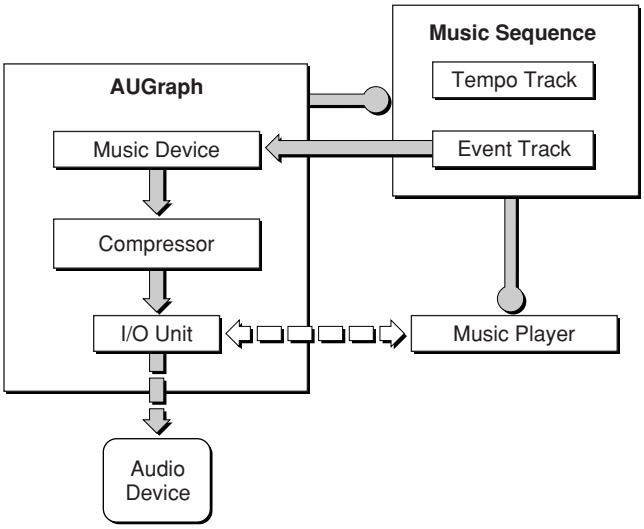
MIDI stands for Music Instrument Digital Interface. Established as the standard method of communication between music devices, Core Audio features full-fledged MIDI support, including provisions for communication with MIDI devices and reading-in and playback of standard MIDI files.

Figure 2-8 Reading in a standard MIDI file



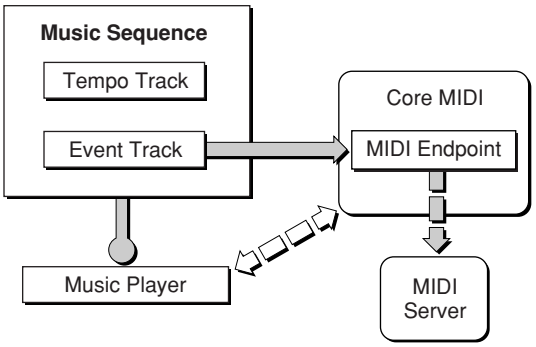
The Music Sequence API is provided to sequence events for MIDI endpoints and audio units. One of its functions, though, is the ability to read in MIDI files and parse their contents into its tracks. Normally, each channel of MIDI data in the file can be made into one track in the sequence, allowing each track, and therefore, each channel of data, to be targeted at a different MIDI endpoint.

Figure 2-9 Music Sequence play through



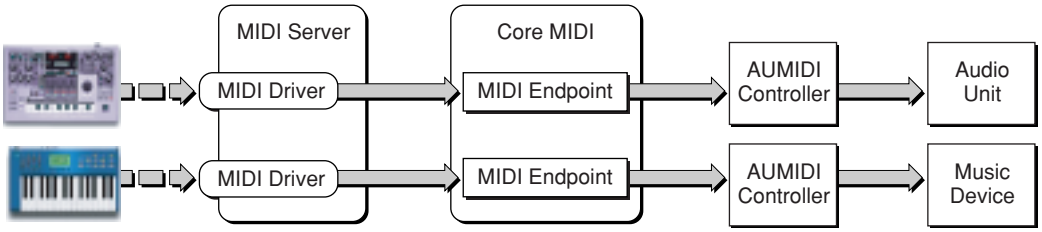
To playback the MIDI file as audio data, a music player is assigned to a sequence, and the sequence's tracks are assigned to a music device. A music device is a particular type of audio unit that generates audio data by having its parameters altered; in this case, the event track is assigned to a music device which is part of a graph, and the events in that track contain the parameter changes needed to affect the output of the music device. The graph itself is assigned to a sequence, so that the sequence knows which instances its tracks are assigned to. Beyond that, the music player assigned to the sequence communicates with the I/O unit at the head of the graph, to ensure that all timing issues for outputting sound to the unit's assigned device are taken care of. This is done inherently when the sequence is assigned to the graph, and so no extra steps need to be taken in order for this synchronization to happen. The compressor is included in order to make sure a constant stream of data is being supplied to the I/O unit.

Figure 2-10 Music Sequence play through



To play MIDI data back through an attached MIDI device, an event track needs to be assigned to a MIDI endpoint, a proxy for a MIDI device. As with the previous example, the music player will inherently communicate with Core MIDI to ensure all timing issues are solved and that a constant amount of data is being fed to the MIDI sever, and therefore, the MIDI device.

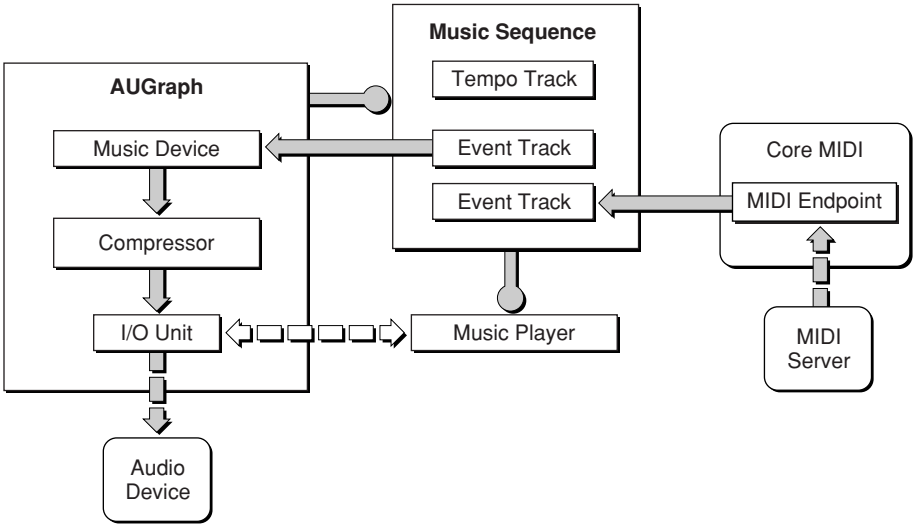
Figure 2-11 MIDI device input



When a MIDI control surface is being used to control the properties of a software component, like an audio unit, it will be assigned to an endpoint, which in turn, is assigned to an `AUMIDIController`, which will parse the incoming MIDI signals into parameter changes for use with an audio unit.

To playback the signals generated by a MIDI keyboard, a similar scheme is used. An endpoint is assigned to the keyboard, and the signals coming from the keyboard are assigned to an `AUMIDIController`, which, in turn, will issue parameter changes to a music device. The music device will synthesize the audio data, based on the parameters given to it via the `AUMIDIController`.

Figure 2-12 MIDI input parsing

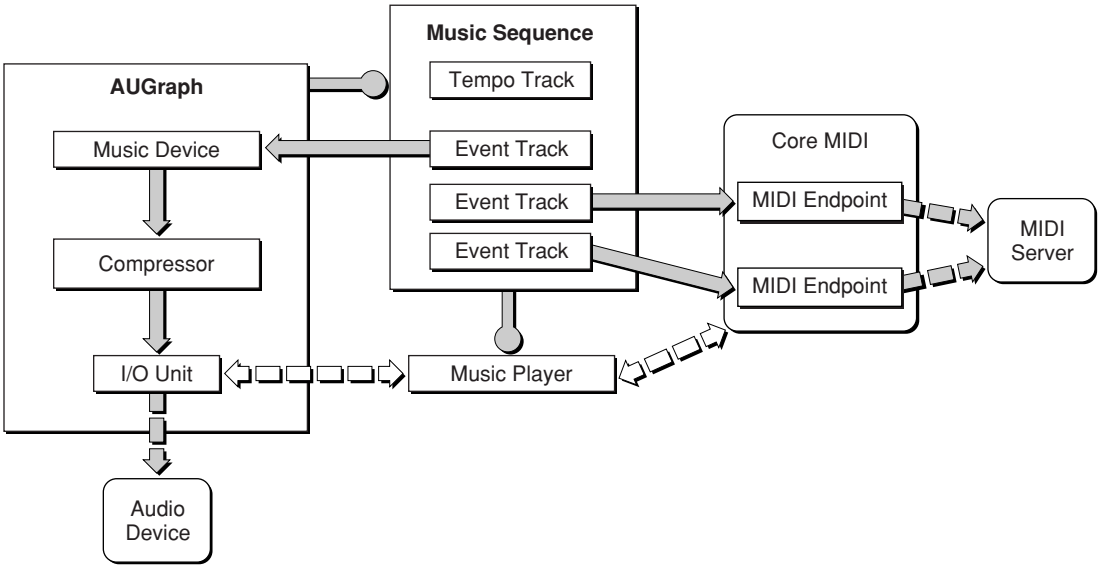


To take in MIDI data for saving, it is common to have already-existing data playing, while new data comes in and is recorded. The playback of existing data is handled as before, with the track being assigned to a music device, which outputs its data, via a graph, to an I/O unit. Beyond that, however, the data coming in from any MIDI device needs to be parsed and placed in another event track within the sequence. The Music Player API provides functions for determining when to place the events, based on the time the event happens.

Higher Level Audio Operations

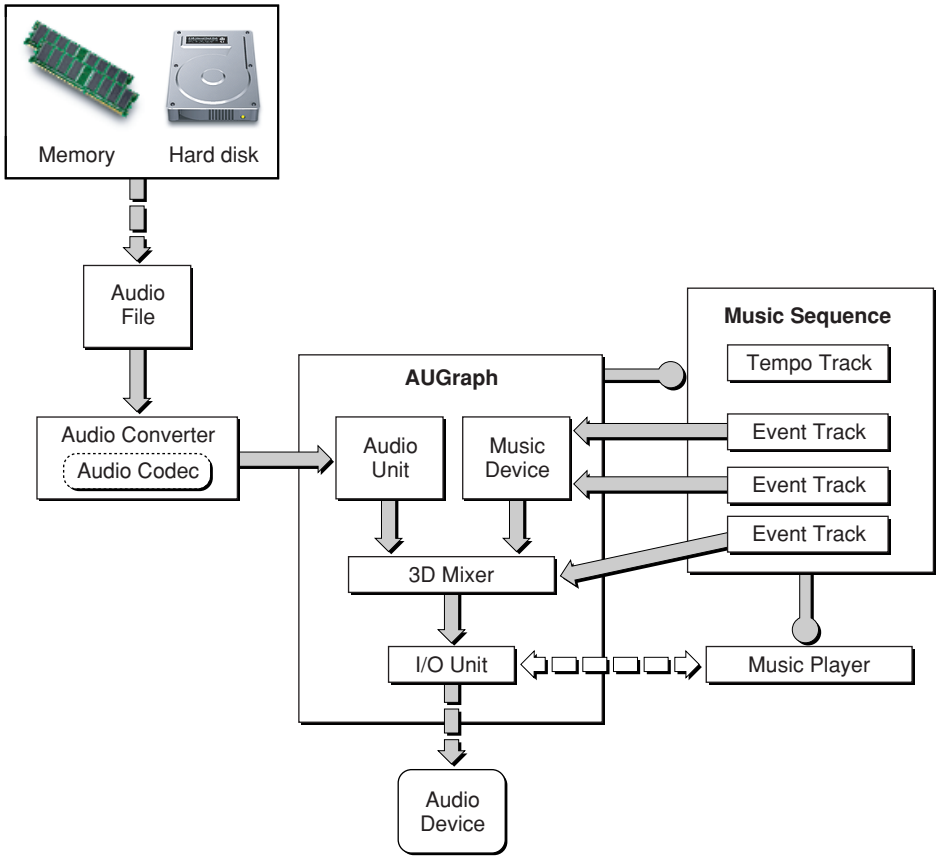
Often, elements from the audio data operations and the MIDI data operations come together to provide a complete audio experience. These examples look at some cases where MIDI data is synthesized and also output to a MIDI device concurrently, or when events control a music device's synthesis of audio data and the parameters of an audio unit, all while mixing in data from an encoded file being read off of disk.

Figure 2-13 MIDI synthesis and output



In this example, you can see a music sequence being used to control the synthesis of sound, via an audio unit graph containing a music device, while additional events are sent to a MIDI endpoint, which, in turn, are assigned to MIDI devices. This is common when using the Mac as another MIDI device, generating synthesized data to accompany an external MIDI device. Note that the music player automatically takes care of all timing issues between the different outputs, ensuring that the output remains in sync.

Figure 2-14 Mixing MIDI and audio data

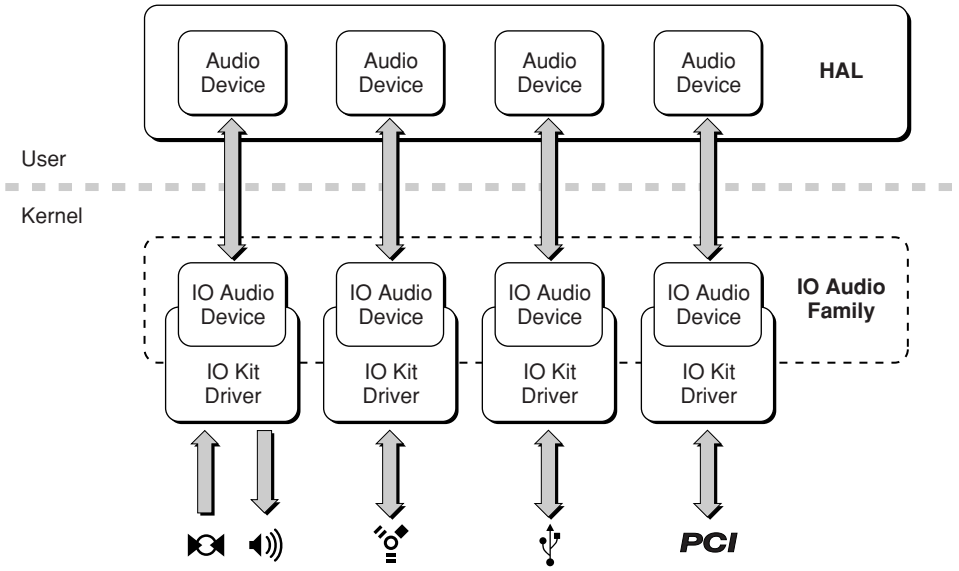


This example focuses on an audio unit graph, which is used to mix synthesized MIDI data, via a music device, and audio data coming in from a file. This scenario is common in gaming situations, where ambient noises are saved as MIDI data, and the sound track is an encoded file on disk. Note that the sequence controls a 3D Mixer audio unit, often used to mix various audio sources and to provide a spacial orientation for the sources and the output. As with the previous example, the music player will ensure that the output is in sync with the sequence.

Interfacing with Hardware

Most of the processing done with audio and MIDI data in Core Audio will eventually be played back via audio or MIDI hardware. As a developer, it will be helpful to you if you understand the architecture behind the hardware interfaces, even if an abstraction is used when developing an application.

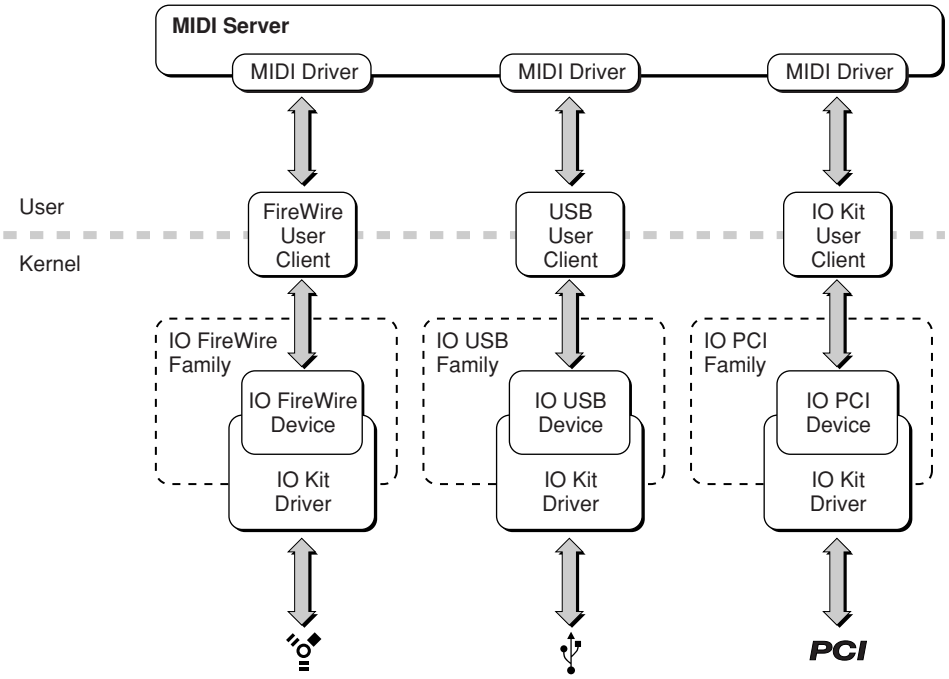
Figure 2-15 Audio hardware architecture



When accessing audio hardware, whether it be via on-board audio inputs and outputs, USB, or other means, a driver must exist to handle the exchange of data between the hardware and the Mac. In order for the driver to be used by Core Audio, it must conform to the `IO Audio Family` of IOKit drivers; this means that the driver music implement `IO Audio Device` functionality within the driver, in order for proper communication to exist between itself and the Hardware Abstraction Layer.

The Hardware Abstraction Layer, or HAL, is provided to make discovery and access to audio hardware simpler. Each driver in the `IO Audio Family` is represented as an audio device in the HAL. To make communication with audio devices easier, and I/O unit may created and bound to an audio device, allowing a device to be used as a source, destination, or both in connections with audio units and audio unit graphs. This is common and encouraged when working with audio hardware.

Figure 2-16 MIDI hardware architecture



The MIDI hardware architecture is different than that of audio hardware, in that MIDI drivers are in user space, usually working with default drivers provided by the operating system. This means that raw incoming and outgoing data is passed between the hardware and the MIDI driver, and the MIDI driver takes care of the formatting and preparation of the data. The MIDI Server than works with Core MIDI, routing MIDI data via endpoints, the abstraction provided to allow for easy access to MIDI devices.

Audio Codec

This chapter discusses audio codec development using the Audio Codec API. The section "Audio Codec Reference" (page 26) describes the constants, data types, and functions that are relevant to audio codec development.

Overview of Audio Codec

Audio codecs are encoders and decoders provided by Apple or third-party developers for the purpose of compressing and decompressing audio streams into and from encoded formats. While codec use is encouraged, normal use should be performed through the "Audio Converter" (page 45).

Before developing an audio codec for Mac OS X, install the Core Audio SDK, available from http://developer.apple.com/audio/.

Of particular interest are the contents of `/AudioCodecs/ACPublic/`, in the installed SDK.

The ACCodec Class

`ACCodec` is an abstract class that defines the basic methods that an audio codec must implement. At the very least, all codecs must subclass `ACCodec` to provide basic services for those who wish to use the codec, based on standard methods of communication expected of them. However, two subclasses of `ACCodec` are provided, which implement many of this class's abstract methods for your convenience.

The ACBaseCodec Class

`ACBaseCodec` is a subclass of `ACCodec` and provides many of the services needed by most codecs to interact with codec clients. Property management is fully implemented in `ACBaseCodec` but you may override it as needed. Also, this class provides format management for input and outputs, including getting and setting the number of input and output formats, and getting and setting the `AudioStreamBasicDescription` format information for inputs and outputs.

`ACBaseCodec` does not implement a buffer for the codec, however, and you must implement all methods pertaining to buffer usage, depending on your chosen buffer size and implementation.

When developing an audio codec, you must subclass `ACBaseCodec`.

The ACSimpleCodec Class

`ACSimpleCodec`, a subclass of `ACBaseCodec`, provides a ring buffer implementation with a variable buffer size providing for reallocation of the buffer. This class is provided as a convenience for you in cases where you don't need a custom buffering scheme. When developing a codec, you don't need to subclass `ACSimpleCodec`, so long as you provide a buffering scheme.

Miscellaneous Headers

Inside `/AudioCodecs/ACPublic/` are other headers that need not be modified but are necessary for the operation of a codec, and therefore are included in order for the codec to operate properly:

- `ACConditionalMacros.h`. This header helps determine which system headers need to be included at compile time.
- `ACCodecDispatch.h`. An implementation of an audio codec component dispatch method.
- `ACCodecDispatchTypes.h`. Glue that helps `ACCodecDispatch` work properly on various platforms.

Audio Codec Reference

This reference section describes the methods that need to be implemented in order to deploy an audio codec component for Mac OS X.

Note: The term "magic cookie" is used in this reference section. A magic cookie refers to header information (usually a vector of bits) that is placed at the beginning of most audio files and contains information vital for the codec.

Audio Codec Types

Defined Data Types

Typedefs are used for naming convenience, in an effort to make information stored in generic variables more easily recognizable. There are two typedefs in `AudioCodec.h`:

- `typedef ComponentInstance AudioCodec`
- `typedef UInt32 AudioCodecPropertyID`

Data Structures

AudioStreamLoudnessStatistics

Encapsulates various pieces of information with regards to the loudness of the stream currently in use.

```
typedef struct AudioStreamLoudnessStatistics {
Float64 mAveragePerceivedPowerCoefficient;
Float64 mMaximumPerceivedPowerCoefficient;
UInt64 mMaximumPerceivedPowerPacketOffset;
Float32 mPeakAmplitude;
UInt32 mReserved;
UInt64 mPeakAmplitudeSampleOffset;
} AudioStreamLoudnessStatistics;
```

Discussion
An instance of this structure is returned when the `kAudioCodecPropertyCurrentLoudnessStatistics` property is queried. The `mAveragePercievedPowerCoefficient` and `mMaximumPercievedPowerCoefficient` are a normalized value on a scale of 0 to 1, while `mMaximumPerceivedPowerPacketOffset` specifies the power's offset. The `mPeakAmplitude` value is the largest sample value in the entire audio stream. `mPeakAmplitudeSampleOffset` is the number of the sample where the peak is found.

Availability
Available in Mac OS X v10.3 through Mac OS X v10.4.

Declared In
`AudioCodec.h`

AudioCodecPrimeInfo

Holds information about leading and following frames for a stream of data.

```
typedef struct AudioCodecPrimeInfo {
UInt32 leadingFrames;
UInt32 trailingFrames;
} AudioCodecPrimeInfo;
```

Discussion
Many times, an audio stream has audio data that fixes and appends the actual content that needs to be encoded or decoded, based on the codec's implementation. This structure holds how many leading and trailing frames there are. A pointer to an instance of this structure is accessible via the `kAudioCodecPrimeInfo` property.

Availability
Available in Mac OS X v10.3 and later.

Declared In
`AudioCodec.h`

Audio Codec Constants

Constants are used throughout the Audio Codec API to provide information required by the Component Manager in order to function or to work in conjunction with properties to describe a state or setting.

Component Identifiers

Identifies the codec for the Component Manager.

`kAudioDecoderComponentType = 'adec'`
: The identifier for a codec that translates data in some format into linear PCM. The subtype of this codec specifies the format of the incoming data.

`kAudioEncoderComponentType = 'aenc'`
: The identifier for a codec that translates data from linear PCM into the codec's specified output format. The subtype of this codec specifies the format of the outgoing data.

`kAudioUnityCodecComponentType = 'acdc'`
: The identifier for a codec that translates data between different types of the same format. The subtype of this codec specifies the format ID of the format that is being converted between.

Quality Settings

Specify the relative quality of a codec. The values are arbitrary and are provided as a suggestion. You can add more quality setting constants, or ignore them.

`kAudioCodecQuality_Max = 0x7F`
: The maximum value allowed for the codec.

`kAudioCodecQuality_High = 0x60`
: A high quality setting.

`kAudioCodecQuality_Medium = 0x40`
: A medium quality setting.

`kAudioCodecQuality_Low = 0x20`
: A low quality setting.

`kAudioCodecQuality_Min = 0`
: The minimum quality setting allowed for the codec.

These constants are used by the `kAudioCodecPropertyQualitySetting` property. They are arbitrary and are provided as a suggestion. You can add more quality setting constants or ignore them altogether.

Priming Selectors

Specify the priming method use.

`kAudioCodecPrimeMethod_Pre = 0`
: The codec primes with the leading and trailing input frames.

`kAudioCodecPrimeMethod_Normal = 1`
: The codec primes with the trailing input frames only; all leading frames are assumed to be silence.

`kAudioCodecPrimeMethod_None = 2`
: The codec does not prime; both leading and trailing frames are assumed to be silent.

These constants are used with the `kAudioCodecPrimeMethod` property. The number of frames that are used in priming is stored in the `AudioCodecPrimeInfo` (page 27) structure, which is accessible through the `kAudioCodecPrimeInfo` property.

Output Packet Status Constants

When using the `ProduceOutputPackets()` method to pull data through the codec, one of the parameters, `outStatus`, has a value that reflects the state of the encode or decode after the pull has occurred. These are the possible values:

`kAudioCodecProduceOutputPacketFailure = 1`
: There was an error processing the data; check *ioNumberPackets* to see how many packets were processed.

`kAudioCodecProduceOutputPacketSuccess = 2`
: The requested packets were processed successfully and to completion.

`kAudioCodecProduceOutputPacketSuccessHasMore = 3`
: The requested packets were processed successfully, and there are more left to process.

`kAudioCodecProduceOutputPacketNeedsMoreInputData = 4`
: There was not enough data to produce the requested number of packets; check *ioNumberPackets* to see how many were produced.

`kAudioCodecProduceOutputPacketAtEOF = 5`
: An end-of-file was encountered during processing, and therefore the requested number of packets were not produced; check `ioNumberPackets` to see how many were produced.

Audio Codec Properties

The property management system keeps track of information about the codec's operation, settings, and capabilities. These are used in conjunction with `GetPropertyInfo` (page 35), `GetProperty` (page 36), and `SetProperty` (page 36).

`kAudioCodecPropertyNameCFString = 'lnam'`
: Returns a `CFStringRef` with the name of the codec's format.

`kAudioCodecPropertyManufacturerCFString = 'lmak'`
: Returns a `CFStringRef` with the name of the codec's manufacturer.

`kAudioCodecPropertyRequiresPacketDescription = 'pakd'`
: Returns a `UInt32` where a value of `1` means that the codec requires an `AudioStreamPacketDescription` (page 162) to be supplied with any data in the format. If `0` is returned, `AudioStreamPacketDescription` is not needed. This property is used primary in the `AppendInputData` (page 43) method.

`kAudioCodecPropertyPacketFrameSize = 'pakf'`
: Passes back a `UInt32` that reflects the number of frames of audio data in each packet of data in the codec's format. The value is for an encoder's input or a decoder's output. Note that this value may be queried only after a codec's initialization.

`kAudioCodecPropertyHasVariablePacketByteSizes = 'vpk?'`
: Returns a `UInt32` where a value of `0` indicates that all of the packets in the codec's format have the same byte size, whereas a value of `1` indicates that the packets vary in size.

`kAudioCodecPropertyMaximumPacketByteSize = 'pakb'`
: If the codec's format has a constant packet size, this value will be the number of bytes in a packet of the codec's format; if the format has a variable bit rate, this value will be the number of bytes in the largest of the packets in the codec's format.

`kAudioCodecPropertyCurrentInputFormat = 'ifmt'`
: Returns an `AudioStreamBasicDescription` (page 160) describing the current input format for the codec.

`kAudioCodecPropertySupportedInputFormats = 'ifm#'`
: Returns an `AudioStreamBasicDescription` (page 160) array describing the input formats supported by the codec.

`kAudioCodecPropertyCurrentOutputFormat = 'ofmt'`
: Returns an `AudioStreamBasicDescription` (page 160) describing the current output format for the codec.

`kAudioCodecPropertySupportedOutputFormats = 'ofm#'`
: Returns an array of `AudioStreamBasicDescription` (page 160) describing the output formats supported by the codec.

`kAudioCodecPropertyMagicCookie = 'kuki'`
: Returns an untyped buffer of configuration data the codec requires to process the stream of data. Note that not every codec requires a magic cookie.

`kAudioCodecPropertyInputBufferSize = 'tbuf'`
: Returns a `UInt32` that contains the maximum input buffer size for the codec, in bytes.

`kAudioCodecPropertyUsedInputBufferSize = 'ubuf'`
: Returns a `UInt32` that contains the number of bytes currently in use in the buffer.

`kAudioCodecPropertyIsInitialized = 'init'`
: Returns a `UInt32` where a value of `0` means the codec is initialized, while any other value means the codec is initialized.

`kAudioCodecPropertyCurrentTargetBitRate = 'brat'`
: A `UInt32` containing the number of bits per second to aim for when encoding data. This property is only relevant to encoders.

`kAudioCodecPropertyAvailableBitRates = 'brt#'`
: A `UInt32` array containing the target bit rates supported by the encoder. Note that use of this property is deprecated in favor of `kAudioCodecPropertyAvailableBitRateRange`.

`kAudioCodecPropertyCurrentInputSampleRate = 'cisr'`
: Passes back a `Float64` containing current input sample rate in Hz.

`kAudioCodecPropertyCurrentOutputSampleRate = 'cosr'`
: Passes back a `Float64` containing current output sample rate in Hz.

`kAudioCodecPropertyAvailableInputSampleRates = 'aisr'`
: Returns an array of `AudioValueRange` (page 159) structures indicating the valid ranges for the input sample rate of the codec for the current bit rate.

`kAudioCodecPropertyAvailableOutputSampleRates = 'aosr'`
: Returns an array of `AudioValueRange` (page 159) structures indicating the valid ranges for the output sample rates of the codec for the current bit rate.

`kAudioCodecPropertyQualitySetting = 'srcq'`
: Returns a value to indicate the relative value of the codec's encoding or decoding quality; see "Quality Settings" (page 28) for possible values.

`kAudioCodecPropertyCurrentLoudnessStatistics = 'loud'`
: Passes back a reference to an `AudioStreamLoudnessStatistics` (page 26) array that provides statistics about the loudness of each channel in the stream of data being processed by the codec. Note that this property can be queried only when the codec is initialized. Until data has actually moved through it, the values are all defaults.

`kAudioCodecPropertyAvailableBitRateRange = 'abrt'`
: Returns an array of `AudioValueRange` (page 159) structures that represents the target bit rates supported by the encoder.

`kAudioCodecPropertyApplicableBitRateRange = 'brta'`
: Returns an array of `AudioValueRange` (page 159) structures that represents the target bit rates supported by the encoder as it is currently configured.

`kAudioCodecPropertyApplicableInputSampleRates = 'isra'`
: Returns an array of `AudioValueRange` (page 159) structures that represents the valid ranges for the input sample rates of the codec for the current bit rate.

`kAudioCodecPropertyApplicableOutputSampleRates = 'osra'`
: Returns an array of `AudioValueRange` (page 159) structures that represents the valid ranges for the output sample rates of the codec for the current bit rate.

`kAudioCodecPropertyMinimumNumberInputPackets = 'mnip'`
: Returns a `UInt32` reflecting the minimum number of packets that need to be supplied to the codec.

`kAudioCodecPropertyMinimumNumberOutputPackets = 'mnop'`
: Returns a `UInt32` indicating the minimum number of output packets that need to be handled from the codec.

`kAudioCodecPropertyZeroFramesPadded = 'pad0'`
: Returns a `UInt32` indicating the number of zeroes (samples) that were appended to the last packet of input data to make it a complete packet.

`kAudioCodecPropertyChannelLayout = 'cmap'`
: Returns an array of `AudioChannelLayout` (page 164) structures that specifies the channel layout that the codec is using.

`kAudioCodecPropertyAvailableChannelLayouts = 'cmp#'`
: Returns an array of `AudioChannelLayout` (page 164) structures array that contains the channel layouts the codec is capable of using.

`kAudioCodecPrimeMethod = 'prmm'`
: Passes back a `kAudioCodecPrimeMethod` constant specifying the priming method currently used by the codec. See "Priming Selectors" (page 28) for possible values.

`kAudioCodecPrimeInfo = 'prim'`
: Uses a pointer to an instance of `AudioCodecPrimeInfo` (page 27) to specify the leading and trailing number of frames used in priming.

Base Classes

There are three classes provided in the Audio Codec SDK, located in `/AudioCodecs/ACPublic/`, one of which must be subclassed when developing a codec: `ACCodec`, `ACBaseCodec`, and `ACSimpleCodec`. It is strongly advised that the developer subclass `ACBaseCodec` over `ACCodec`when developing an audio codec, since it provides many of the property and format management feature required of an audio codec. Note that abstract methods that exist in `ACCodec` must be implemented in subclasses of `ACBaseCodec` and `ACSimpleCodec`.

ACCodec

`ACCodec` defines the basic methods which any audio codec must implement to be used in Mac OS X. Most of the methods belonging to `ACCodec` are abstract, since they need to customized by the codec developer for the codec being implemented. Please note that many of these methods are implemented by `ACBaseCodec` and `ACSimpleCodec`.

Construction and Destruction

Constructors and destructors should be implemented for the codec as needed, and should at least allocate the needed buffer space for the codec and deallocate the space upon the codec's destruction.

Property Management

These methods provide access to the property management system, which keeps track of various pieces of information about the codec. Their implementation is required since other components in Core Audio rely on their existence. Note that `ACBaseCodec` contains implementations of these methods that are adequate for most uses.

GetPropertyInfo

Passes back the size of the property data belonging to `inPropertyID`, and its writable state.

```
virtual void GetPropertyInfo(
AudioCodecPropertyID inPropertyID,
UInt32& outSize,
bool& outWritable
) = 0
```

Availability
Available in Mac OS X v10.0 through Mac OS X v10.1.

Declared In
`AudioUnit.k.h`

GetProperty

Passes the property data belonging to `inPropertyID` into `outPropertyData`.

```
virtual void GetProperty(
AudioCodecPropertyID inPropertyID,
UInt32& ioPropertyDataSize,
void* outPropertyData
) = 0
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Movies.k.h`

SetProperty

Takes `inPropertyData` and sets it to the property data belonging to `inPropertyID`.

```
virtual void SetProperty(
AudioCodecPropertyID inPropertyID,
UInt32 inPropertyDataSize,
const void* inPropertyData
) = 0
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Movies.k.h`

Data Handling

The Data Handling methods deal with getting data into, processed, and out of the audio codec. All of these methods are abstract and must be implemented by subclasses of `ACCodec`.

Initialize

Specifies the input and output formats for the data coming into and going out of the codec, as well as the magic cookie for the data currently being encoded or decoded.

```
virtual void Initialize(
const AudioStreamBasicDescription* inInputFormat,
const AudioStreamBasicDescription* inOutputFormat,
const void* inMagicCookie,
UInt32 inMagicCookieByteSize
) = 0
```

Discussion
When a codec is initialized, all of its properties should be locked, so that they may not be changed during the encoding or decoding. Note that a codec may be used only after it is initialized.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`ImageCodec.k.h`

Uninitialize

Unlocks the codec, so that its properties may be altered.

```
virtual void Uninitialize() = 0
```

Discussion
An audio codec that has been uninitialized may not be used to encode or decode data, since its properties may be altered at any time.

Availability
Available in Mac OS X v10.0 through Mac OS X v10.1.

Declared In
`AudioUnit.k.h`

AppendInputData

Passes in data to placed in the buffer for encoding or decoding.

```
virtual void AppendInputData(
const void* inInputData,
UInt32& ioInputDataByteSize,
UInt32& ioNumberPackets,
const AudioStreamPacketDescription* inPacketDescription
) = 0
```

Discussion
Here, `inInputData` is the data that should be put in the input buffer, and the rest of the parameters describe the size and nature of the data.

ProduceOutputPackets

Runs the codec and returns encoded or decoded data.

```
virtual UInt32 ProduceOutputPackets(
void* outOutputData,
UInt32& ioOutputDataByteSize,
UInt32& ioNumberPackets,
AudioStreamPacketDescription* outPacketDescription
UInt32& outStatus,
) = 0
```

Discussion
Calling `ProduceOutputPackets()` should pull `ioNumberPackets` from the buffer, encode or decoded them, and place them in `outOutputData`. The value of `outStatus` should be based on the `kAudioCodecProduceOutputPacket` set of enumerated values, informing the caller if the encode or decode failed, was a success, was a success and has more to encode or decode, was partially successful, yet needed more input, or was at the end of the file.

Reset

Clears out the codec's input buffer and returns any state info to its initial settings.

```
virtual void Reset() = 0
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`QTStreamingComponents.k.h`

Component Support

An audio codec is a Component, and therefore, needs to be accessible from the Component Manager in order for an application to use it. These methods perform duties that Components must perform in order to be used.

Register

Registers the audio codec with the Component Manager.

```
virtual bool Register() const
```

Discussion
This method is provided in `ACCodec` and usually does not need to be overridden; however, `Register()` is virtual, should this be necessary.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`Components.k.h`

GetVersion

Returns the version number of the codec.

```
virtual UInt32 GetVersion() const
```

Discussion
This method should be overridden to reflect the version number of the codec; however, this is not required of the codec, only recommended.

ACBaseCodec

`ACBaseCodec` is provided as a base for building new audio codec components. It provides all of the Property Management features required of an audio codec, as well as most Format Management methods. Use of `ACBaseCodec` is encouraged when the audio codec being developed has its own custom buffering scheme, and would not benefit from the ring buffer already provided for in `ACSimpleCodec`.

Construction and Destruction

Constructors and destructors should be developed as needed, and may allocate the needed buffer space for the codec. Also, `AddInputFormat` (page 41) and `AddOutputFormat` (page 41) should be called from a constructor to set up descriptions for the codec's supported formats.

Property Management

These methods are required of an audio codec and are implemented for developer convenience in `ACBaseCodec`. While overriding these methods is possible, it is usually not necessary.

GetPropertyInfo

Takes in an `AudioCodecPropertyID`, and, by reference, passes back the size of the property data, and a flag telling if the property is writable.

```
virtual void GetPropertyInfo(
AudioCodecPropertyID inPropertyID,
UInt32& outPropertyDataSize,
bool& outWritable
)
```

Availability
Available in Mac OS X v10.0 through Mac OS X v10.1.

Declared In
`AudioUnit.k.h`

GetProperty

Takes in a `AudioCodecPropertyID`, the property data size (attained using `GetPropertySize()`), and a `void` pointer for the resulting data.

```
virtual void GetProperty(
AudioCodecPropertyID inPropertyID,
UInt32& ioPropertyDataSize,
void* outPropertyData)
```

Discussion
`inPropertyID` corresponds to any of the properties listed in the `Audio Codec Properties` (page 29) section. For the `outPropertyData` parameter, pass in a `void` pointer, and the data promised for the property will be passed back into the pointer, typecast to the appropriate type.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Movies.k.h`

SetProperty

Takes in the `AudioCodecPropertyID` of the property to be set, the size of the data to be set, and a `void` pointer to the data.

```
virtual void SetProperty(
AudioCodecPropertyID inPropertyID,
UInt32 inPropertyDataSize,
const void* inPropertyData
)
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`Movies.k.h`

Data Handling

These methods are used for initialization, uninitialization, and like-minded methods for a codec. Many of these methods may be left as implemented in `ACBaseCodec`, but may be overridden as needed. Note that some methods are abstract, meaning that, in an `ACBaseCodec` subclass, they must be implemented.

IsInitialized

Returns the value of `mIsInitialized`, which should be set in `Initialize()` and `Uninitialize()`.

```
bool IsInitialized() const { return mIsInitialized; }
```

Initialize

Passes in an input and output format, the codec's magic cookie, and the cookie's size.

```
virtual void Initialize(
const AudioStreamBasicDescription* inInputFormat,
const AudioStreamBasicDescription* inOutputFormat,
const void* inMagicCookie,
UInt32 inMagicCookieByteSize)
```

Discussion
The purpose of `Initialize()` is to provide a mechanism for the codec user to specify the formats of the incoming and outgoing data, the proper data for the magic cookie, and the size of the magic cookie. Initializing a codec should lock it so that none of its attributes maybe changed. Note that encoding and decoding should only commence after the codec is initialized

Availability
Available in Mac OS X v10.0 and later.

Declared In
`ImageCodec.k.h`

Uninitialize

Unlocks the codec, so that its attributes may be modified.

```
virtual void Uninitialize()
```

Discussion
Note that no encoding or decoding should be allowed when an audio codec is uninitialized.

Availability
Available in Mac OS X v10.0 through Mac OS X v10.1.

Declared In
`AudioUnit.k.h`

Reset

Clears out the codec's buffer and returns it to the post-initialized state.

```
virtual void Reset()
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`QTStreamingComponents.k.h`

GetInputBufferSize

Returns the current input buffer size as a `UInt32`.

```
virtual UInt32 GetInputBufferByteSize() const = 0
```

Discussion
This abstract method must be implemented by any subclass of `ACBasicCodec`, since the return value is dependant on the buffer size used in the buffer scheme implemented by the developer.

GetUsedInputBufferSize

Returns a `UInt32` which reflects how much of the input buffer is currently filled up.

```
virtual UInt32 GetUsedInputBufferByteSize() const = 0
```

Discussion
This abstract method must be implemented by any subclass of `ACBasicCodec`.

ReallocateInputBuffer

Resizes the buffer to `inInputBufferSize`.

```
virtual void ReallocateInputBuffer(UInt32 inInputBufferByteSize) = 0
```

Discussion
For codecs which implement user-definable input buffer sizes, `ReallocateInputBuffer()` is a mechanism which allows for resizing of the input buffer.

Format Management

This portion of `ACBaseCodec` focuses on keeping track of the formats that a codec can decode and encode. If should be noted that while none of these methods are declared abstract, it would be advantageous to override some of them; conversely, some of these methods are not virtual, so what `ACBasicCodec` implements for them is adequate.

GetNumberSupportedInputFormats

Counts and returns the number of formats that the codec supports.

```
UInt32 GetNumberSupportedInputFormats() const
```

GetSupportedInputFormats

Passes back an array of the supported input formats for this codec.

```
void GetSupportedInputFormats(
AudioStreamBasicDescription* outInputFormats,
UInt32& ioNumberInputFormats
) const
```

Discussion
After the number of supported formats is acquired, it should be used in this method, which will pass back an `AudioStreamBasicDescription` array which contains the descriptions of all of the supported input formats. `ioNumberInputFormats` is also passed back as a check to ensure that the correct number of formats were returned.

GetCurrentInputFormat

Passes back an `AudioStreamBasicDescription` with the current input format information in it.

```
void GetCurrentInputFormat(AudioStreamBasicDescription& outInputFormat)
```

SetCurrentInputFormat

Takes an `AudioStreamBasicDescription` and sets it to be the current input format.

```
virtual void SetCurrentInputFormat(
const AudioStreamBasicDescription& inInputFormat
)
```

Discussion
Setting a codec's input format is performed by calling this method and passing it an instance of `AudioStreamBasicDescription`, configured to the proper format. Note that `ACBasicCodec` provides a method for setting the current input format for the codec, but that this may need to be overridden based on the implemented codec's capabilities.

GetNumberSupportedOutputFormats

Counts and returns the number of supported output formats.

```
UInt32 GetNumberSupportedOutputFormats() const
```

GetSupporedOutputFormats

Passes back and array of the supported output formats.

```
void GetSupportedOutputFormats(
AudioStreamBasicDescription* outOutputFormats,
UInt32& ioNumberOutputFormats
) const
```

Discussion
The value of `outOutputFormats` is an `AudioStreamBasicDescription` array. The input for `ioNumberOutputFormats` should be the result of `GetNumberSupportedOutFormats()`, and its output will be the actual number of `AudioStreamBasicDescription` instances in `outOutputFormats`.

GetCurrentOutputFormat

Passes back a description of the current output format.

```
void GetCurrentOutputFormat(AudioStreamBasicDescription& outOutputFormat)
```

SetCurrentOutputFormat

Sets the output format to `inOutputFormat`.

```
virtual void SetCurrentOutputFormat(
const AudioStreamBasicDescription& inOutputFormat
)
```

Discussion
This method is declared as virtual, so that it may be modified as needed to adjust to the needs of different codecs.

GetMagicCookieByteSize

Returns the size of the current magic cookie.

```
virtual UInt32 GetMagicCookieByteSize() const
```

Discussion
If the codec implements a magic cookie, it is strongly advised that the codec developer override this method, since each codec's magic cookie has its own size.

GetMagicCookie

Extracts the magic cookie from the codec.

```
virtual void GetMagicCookie(
void* outMagicCookieData,
UInt32& ioMagicCookieDataByteSize
) const
```

Discussion
This method is virtual and should be overridden based on the codec's requirements for magic cookies.

SetMagicCookie

Sets the magic cookie for the codec.

```
virtual void SetMagicCookie(
const void* outMagicCookieData,
UInt32 inMagicCookieDataByteSize
)
```

Discussion
This method is virtual and should be overridden based on the codec's requirements for magic cookies.

AddInputFormat

Adds `inInputFormat` to the `mInputFormatList`.

```
void AddInputFormat(const AudioStreamBasicDescription& inInputFormat)
```

Discussion
This method is provided for the subclass's constructor, so that the developer may supply input format information. This is then used by other methods in the Property Management system.

AddOutputFormat

Adds `inOutputFormat` to the `mOutputFormatList`.

```
void AddOutputFormat(const AudioStreamBasicDescription& inIOutputFormat)
```

Discussion
This method is provided for the subclass's constructor, so that the developer may supply input format information. This is then used by other methods in the Property Management system.

ACSimpleCodec

`ACSimpleCodec` builds upon the foundation laid in `ACBaseCodec` by providing a simple ring buffer implementation. Subclassing from `ACSimpleCodec` is not required, but when the implemented codec does not require a custom buffering scheme, using `ACSimpleCodec` is recommended.

Construction and Destruction

The `ACSimpleCodec` constructor takes in the size that the ring buffer should be, in bytes, and allocates the appropriate space in memory. The destructor deallocates the buffer, and should be called from any subclass's destructor.

Data Handling

`ACSimpleCodec` inherits all of the format and property management methods from `ACBaseCodec`, and so the only methods that need overridding for `ACSimpleCodec` are those pertaining to Data Handling; that is, those that deal with taking in and outputting converted data.

Initialize

Sets the codec's input format, output format, and its current magic cookie.

```
virtual void Initialize(
const AudioStreamBasicDescription* inInputFormat,
const AudioStreamBasicDescription* inOutputFormat,
const void* inmagicCookie,
UInt32 inmagicCookieByteSize
) = 0
```

Discussion
This method initializes the codec, meaning that is locks down the input and output formats, as well as the magic cookie for the buffer that is about to be encoded. It also locks all of the codecs properties, so that they may not be altered during a conversion. Encoding and decoding should be possible only when the codec is initialized. Note that this method is declared as abstract, meaning that it must be overridden by any subclass of `ACSimpleCodec`; this is due to varying magic cookie implementations.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`ImageCodec.k.h`

Uninitialize

Unlocks the codec so that its formats and properties may be altered.

```
virtual void Uninitialize()
```

Discussion
This method is virtual, meaning that it may be overridden as needed in subclasses of `ACSimpleCodec`.

Availability
Available in Mac OS X v10.0 through Mac OS X v10.1.

Declared In
`AudioUnit.k.h`

Reset

Clears the codec's buffer and returns the codec to the post-initialization state.

```
virtual void Reset()
```

Discussion
This method is virtual, meaning that it may be overridden as needed in subclasses of `ACSimpleCodec`.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`QTStreamingComponents.k.h`

AppendInputData

Takes a buffer of data in, along with its format description and size information.

```
virtual void AppendInputData(
const void* inInputData,
UInt32& ioInputDataByteSize,
UInt32& ioNumberPackets,
const AudioStreamPacketDescription* inPacketDescription
)
```

Discussion
Note that this method is virtual, meaning that while `ACSimpleCodec` provides this service, it may be overridden if the codec requires it.

GetInputBufferSize

Returns the size of the codec's buffer.

```
virtual UInt32 GetInputBufferByteSize() const
```

Discussion
Note that this method is virtual, meaning that while `ACSimpleCodec` provides this service, it may be overridden if the codec requires it.

GetUsedInputBufferSize

Returns the size of the codec's buffer currently in filled.

```
virtual UInt32 GetUsedInputBufferByteSize() const
```

Discussion
Note that this method is virtual, meaning that while `ACSimpleCodec` provides this service, it may be overridden if the codec requires it.

ConsumeInputData

Runs `inConsumedByteSize` amount of data through the codec.

```
void ConsumeInputData(UInt32 inConsumedByteSize)
```

Discussion
This method should be called from within the codec's `ProduceOutputPackets()`.

GetInputBufferStart

Returns the current start position of the buffer.

```
Byte* GetInputBufferStart() const {
return mInputBuffer + mInputBufferStart;
}
```

GetInputBufferContiguousByteSize

Returns the amount of the buffer left to process.

```
UInt32 GetInputBufferContiguousByteSize() const {
return (mInputBufferStart <= mInputBufferEnd)
? (mInputBufferEnd - mInputBufferStart)
: (mInputBufferByteSize - mInputBufferStart)
}
```

ReallocateInputBuffer

Resizes the buffer to a new size.

```
virtual void ReallocateInputBuffer(UInt32 inInputBufferByteSize)
```

Discussion
Calling this method wipes the buffer contents before it resizes it. Note that this method is virtual, so it may be overridden as needed.

Audio Codec Result Codes

These constants define errors that Audio Codec methods can return when processing data.

```
kAudioCodecNoError = 0,
kAudioCodecUnspecifiedError = 'what',
kAudioCodecUnknownPropertyError = 'who?',
kAudioCodecBadPropertySizeError ='!siz',
kAudioCodecIllegalOperationError = 'nope',
kAudioCodecUnsupportedFormatError = '!dat',
kAudioCodecStateError ='!stt',
kAudioCodecNotEnoughBufferSpaceError = '!buf'
```

Audio Toolbox

This chapter discusses the Audio Converter, Audio File, Audio Format and AUGraph APIs, which are part of the Audio Toolbox for Mac OS X, and the services provided by the Audio Toolbox framework that applications may use for audio processing. The section "Audio Toolbox Reference" (page 58) describes the constants, data types, and functions of the Audio Toolbox framework.

Overview of the Audio Toolbox

The Audio Toolbox framework provides a set of services that applications can use for audio processing:

- `AudioConverter.h`
- `AudioFormat.h`
- `AudioFile.h`
- `AUGraph.h`

In Java, these services are provided in the `com.apple.audio.toolbox` package.

Audio Converter

Audio Converter provides format conversion services. When encoding or decoding audio data, Audio Converter should be utilized, as it allows for sample rate conversions, interleaving and deinterleaving of audio streams, floating-point-to-integer and integer-to-floating-point conversions, and bit rate conversions. Also, the API handles channel reordering, as well as converting between PCM and compressed formats. When encoding or decoding an audio stream, use of Audio Converter is strongly recommended over the direct use of an audio codec, since optimizations are in place to provided for optimal conversions.

Audio Format

The Audio Format API is provided to help handle information about different audio formats. It is able to inspect `AudioStreamBasicDescription` instances and provide more information about a particular format's parameters. This API also can derive information from `AudioChannelLayout` instances, including a description of the channels present in the instance, and the ordering of the channels. Finally, Audio Format can provide information about the encoders and decoders available on the system.

Audio File

Audio File is a system with which audio files may be created, opened, modified, and saved. Besides these operations, it also allows for discovery of global properties, including:

- File types that can be read.
- File types that can be written.
- A name for a file type.
- Stream formats that can be read.
- All file extensions that can be read.
- File extensions for a file type.

AUGraph

The AUGraph is a high-level representation of a set of Audio Units, along with the connections between them. These APIs may be used to construct arbitrary signal paths through which audio may be processed, that is, a modular routing system. The APIs deal with large numbers of Audio Units and their relationships to one another.

AUGraphs provide the following services:

- Real-time routing changes that allow for connections to be created and broken while audio is being processed.
- Maintaining representation even when Audio Units are not instantiated.

The head of a graph is always an output unit, which may save the processed audio stream to disk, into memory, or as sound out. Starting a graph entails “pulling” on the head unit (provided for by API), which will, in turn, pull on the next unit in the graph. The contents of a graph may be saved output and saved for later use.

Music Player and Music Sequence

The Music Player and Music Sequence APIs are used in tandem to sequence various events. Events can range from the changing of an audio unit's parameters to sending a MIDI endpoint a message. Standard MIDI files are played back using the Music Player API, particularly using the provided functions (see Reading in an SMF section). Similarly, you can save incoming MIDI data to a music sequence and then save the sequence as a standard MIDI file.There are three pieces in the Music Player API: players, sequences, and tracks.Players are assigned to a sequence, and trigger the sequence to start and stop. The relation between a player and a sequence is one-to-one, meaning that each player may only have one sequence assigned to it, and vice versa. Players also keep track of the current playback time in the sequence, and allow the playback time to be set. Finally, a scalar can be applied to a player, which will alter the tempo of the assigned sequence by that scalar.A sequence is a collection of tracks. A track is collection of events targeted at either a MIDI endpoint, an audio unit, or a callback. A sequence may contain an arbitrary number of tracks, created as needed. Each sequence also contains one special track, the tempo track. This track measures out the playback rate, in beats-per-minute (bpm). Adding tempo events to a tempo track will change the rate at which events occur from that point on, or until the next tempo event occurs.

When recording MIDI events for saving to disk, the incoming MIDI data needs to be parsed and placed into the sequence, which then can be saved as a standard MIDI file, for later use.

Using the Audio Toolbox

This usage section describes how to utilize the APIs that comprise the Audio Toolbox framework available for Mac OS X.

Using Audio Converter

The Audio Converter API allows for the conversion between various audio formats. These examples are provided to give the developer a feel for using the Audio Converter tool.

Creating a New Audio Converter

```
AudioStreamBasicDescription in, out;
 /* ... Fill out stream descriptions ... */
AudioConverterRef converter;
OSStatus err = AudioConverterNew(&in, &out, &converter);
```

These steps should be followed when creating a new converter:

1. Declare two `AudioStreamBasicDescription` instances, one for the input, and one for the output.
2. Populate the two descriptions with the appropriate stream information.
3. Declare a new converter instance.
4. Invoke the `AudioConverterNew()` function, providing the input, output, and converter as parameters. Note that the parameters are passed by reference.

Converting Audio Data

```
AudioConverterRef converter;
 /* ... Set up the converter .. */
const UInt32 kRequestPackets = 8192;
AudioBufferList bufferList;
 /* ... Allocate the output buffer ... */

while( /* ... While there is data left to be converted ... */ )
{

    UInt32 ioOutputDataPacketSize = kRequestPackets;

    OSStatus err = AudioConverterFillComplexBuffer(converter, inputProcPtr,
                    userData, &ioOutputDataPacketSize, &bufferList, NULL);

}
```

These steps should be followed when pulling data from a converter:

1. Allocate and set up a converter instance.
2. *Optional:* Set up a constant for the amount of data to be pulled.
3. Set up an `AudioBufferList` (page 160) instance to hold the converted data. If the data is interleaved, then only one index is needed for the instance's `mBuffers` array; if the data consists of multiple mono channels, then allocate one index in the `mBuffers` array for each channel.
4. Enter into a loop which pulls data until the `*AudioConverterComplexInputProc` signals that no more data is left to be pulled, or until the desired amount of data is pulled.
5. Inside of the loop, use `AudioConverterFillComplexBuffer()` to pull the data. The parameters passed in this example are:
    - *converter* - The converter to be used.
    - *inputProcPtr* - A callback which provides the input data for conversion.
    - *userData* - Any parameters or constants needed by the `inputProcPtr` callback.
    - *ioOutputDataPacketSize* - Upon input, the requested amount of converted data; on output, the actual amount of data converted.
    - *bufferList* - The buffer for the converted audio data.
    - *NULL* - An `AudioStreamPacketDescription` instance used to describe the size of the resulting packet; only needed when receiving variable bit rate (VBR) data.

Supplying Data for AudioConverterFillComplexBuffer()

```
OSStatus FromFloatInputProc (
    AudioConverterRef inAudioConverter,
    UInt32 *ioNumberDataPackets,
    AudioBufferList *ioData,
    AudioStreamPacketDescription **outDataPacketDescription,
    void *inUserData )
{
MyUserData *data = static_cast<MyUserData*>(inUserData);
AudioBufferList *bufferList = data->bufferList;
for (UInt32 i=0; i < bufferList->mNumberBuffers; ++i)
{
    ioData->mBuffers[i].mNumberChannels =
        bufferList->mBuffers[i].mNumberChannels;
    ioData->mBuffers[i].mData = bufferList->mBuffers[i].mData;
    ioData->mBuffers[i].mDataByteSize =
        bufferList->mBuffers[i].mDataByteSize;
}
*ioNumberDataPackets = ioData->mBuffers[0].mDataByteSize /
                                data->mInputASBD.mBytesPerPacket;
return noErr;
}
```

This example looks at creating an `*AudioConverterComplexInputDataProc` for use by `AudioConverterFillComplexBuffer()`:

1. An *AudioConverterComplexInputDataProc takes in the following arguments:
    - *inAudioConverter* - The converter in use.
    - *ioNumberPackets* - The number of packets requested.
    - *ioData* - The data to be returned to AudioConverterFillComplexBuffer() for conversion.
    - *outPacketDescription* - Provided to give the details of the packet format passed back when decoding. This should be an AudioStreamPacketDescription (page 162) array, with each packet corresponding to a descrption.
    - *inUserData* - Data needed by the callback, for any purpose; in this case, it holds the location of the input data.
2. In a loop, fill each buffer in ioData with the number of channels, requested amount of data, and data byte size.
3. Calculate the number of provided packets, and place the value in *ioNumberDataPackets*.

Using Audio Format

The Audio Format API is provided to acquire information about formats and channel layouts. This example is provided to give the developer a feel for using the Audio Format API.

Getting Format ID Information

```
UInt32 size;
OSStatus err;
OSType *formatIDs;

err = AudioFormatGetPropertyInfo(
                    kAudioFormatProperty_EncodeFormatIDs, 0, NULL, &size);
if (err) return err;

formatIDs = (OSType*)malloc(size);
UInt32 numFormats = size / sizeof(OSType);

err = AudioFormatGetProperty(
         kAudioFormatProperty_EncodeFormatIDs, 0, NULL, &size, formatIDs);
if (err) return err;

for (UInt32 i=0; i<numFormats; ++i)
{
    AudioStreamBasicDescription absd;
    memset(&absd, 0, sizeof(absd));
    absd.mFormatID = formatIDs[i];

    CFStringRef name;
    size = sizeof(CFStringRef);
    err = AudioFormatGetProperty(
        kAudioFormatProperty_FormatName, sizeof(absd), &absd, &size, &name);
    if (err) return err;
```

```
    CFShow(name);
}
```

This example shows how to use the property management system available in Audio Format to acquire information about the encoder formats, and output their names:

1. Use the `AudioFormatGetPropertyInfo()` function to get the size of the data that will be returned by calling `AudioFormatGetProperty()` for that property. The arguments for `AudioFormatGetPropertyInfo()` are:
    - *`kAudioFormatProperty_EncodeFormatIDs`* - The property we are querying for.
    - *`0`* - The size of the specifier; in this case, `0`, since this property does not require a specifier.
    - *`NULL`* - The specifier; in this case, `NULL`, since the property does not require a specifier.
    - *`&size`* - The size of the data that will be returned when `AudioFormatGetProperty()` is called for this property.
2. Once the size is obtained, `AudioFormatGetProperty()` may be called. The same parameters are passed in as with `AudioFormatGetPropertyInfo()`, with the addition of a `void` pointer, in this case *`formatIDs`*, to hold the returned data in.
3. Now that an array of the format IDs have been attained, it may be iterated over by following these steps:
    a. Create an `AudioStreamBasicDescription` instance and clear its contents.
    b. Set the format ID of the `AudioStreamBasicDescription` instance to one of the format IDs returned when `AudioFormatGetProperty()` was called with the `kAudioFormatProperty_EncodeFormatIDs` property.
    c. Create a `CFStringRef` to hold the name of the format, and obtain its size.
    d. To obtain the format's name, call `AudioFormatGetProperty()` with `kAudioFormatProperty_FormatName` as the property, the `AudioStreamBasicDescription` instance as the specifier, and the `CFString` as the *`outPropertyData`* value.
    e. Print out the name of the format, as stored in the `CFString`.

Using Audio File

The Audio File API is used to discover global file format information and to provide an interface for creating, opening, modifying, and saving audio files. This example is provided to give the developer a feel for using the Audio File API.

Acquiring Global File Information

```
OSStatus err;
UInt32 propertySize;

err = AudioFileGetGlobalInfoSize(
                kAudioFileGlobalInfo_WritableTypes, 0, NULL, &propertySize);
```

```
if (err) return err;

OSType *types = (OSType*)malloc(propertySize);
err = AudioFileGetGlobalInfo(
         kAudioFileGlobalInfo_WritableTypes, 0, NULL, &propertySize, types);
if (err) return err;

UInt32 numTypes = propertySize / sizeof(OSType);
for (UInt32 i=0; i<numTypes; ++i)
{
    CFStringRef name;
    UInt32 outSize = sizeof(name);
    err = AudioFileGetGlobalInfo(
kAudioFileGlobalInfo_FileTypeName, sizeof(OSType), types+i, &outSize, &name);
    if (err) return err;

    CFShow(name);
}
```

This example shows how to obtain an array of the writable file types for the system and output their names:

1. Use the `AudioFileGetGlobalInfoSize()` function to get the size of the data that will be returned by calling `AudioFileGetGlobalInfo()` for that property. The arguments for `AudioFileGetGlobalInfoSize()` are:
    - *`kAudioFileGlobalInfo_WritableTypes`* - The property we are querying for.
    - `0` - The specifier's size; in this case, `0`, since there is now need for a specifier for this property.
    - `NULL` - The specifier; in this case, `NULL`, since this property does not require a specifier.
    - `&propertySize` - The size of the data that will be returned when `AudioFileGetGlobalInfo()` is called for this property.
2. Once the size is obtained, it will be used by the `AudioFileGetGlobalInfo()` function. In addition to the four parameters used for `AudioFileGetGlobalInfoSize()`, a fifth is used to point to the actual property data returned by this method. Note that the space for holding this returned data is allocated beforehand, using the *`propertySize`* obtained previously.
3. Once the types have been returned, a loop is used to cycle through the types and query Audio File for the name of the type, which is then printed:
    a. Create a `CFStringRef` to hold the name of the type, and obtain its size.
    b. To obtain the type's name, call `AudioFileGetGlobalInfo()` with `kAudioFileGlobalInfo_FileTypeName` as the property, the type as the specifier, and the `CFString` as the *`outPropertyData`* value.
    c. Print out the name of the format, as stored in the `CFString`.

Using AUGraph

Audio Unit Graph State

An audio unit graph maintains its representation using the `AUNode` type, even when the Audio Unit components themselves are not instantiated.

The AUGraph states are defined as open, initialized, running, and closed. These correspond directly with the Audio Unit states.

The AUGraph APIs are responsible for representing the description of a set of Audio Unit components, as well as the audio connections between their inputs and outputs. This representation may be saved and restored persistently and instantiated by opening all of the Audio Units (`AUGraphOpen()`), and making the physical connections between them stored in the representation (`AUGraphInitialize()`). Thus, the graph is a description of the various Audio Units and their connections, but also manage the actual instantiated Audio Units.

The AUGraph is a complete description of an audio signal processing network.

The AUGraph may be introspected in order to get complete information about all of the Audio Units in the graph. The various nodes (`AUNode`) in the graph representing Audio Units may be added or removed, and the connections between them modified.

An AUNode representing an Audio Unit component is created by specifying a `ComponentDescription` record (from the Component Manager), as well as optional "class" data, which is passed to the Audio Unit when it is opened.

This class data is in an arbitrary format, and may differ depending on the particular Audio Unit. In general, the data is used by the Audio Unit to configure itself when it is opened (in object-oriented terms, it corresponds to constructor arguments). In addition, certain AudioUnits may provide their own class data when they are closed, allowing their current state to be saved for the next time they are instantiated. This provides a general mechanism for persistence.

An AUGraph's state can be manipulated in both the rendering thread and in other threads. Consequently, any activities that effect the state of the graph are guarded with locks.To avoid blocking the render thread, many of the calls to AUGraph may return `kAUGraphErr_CannotDoInCurrentContext`. This result is only generated when an graph modification is called from within a render callback. It means that the lock that it required was held at that time by another thread. If this result code is returned, the action may be retried, typically on the next render cycle (so in the mean time the lock can be cleared), or the action may be delegated to another thread. As a general rule, the render thread should not be allowed to spin.

Setting up an Audio Unit Graph

When using AUGraph, certain steps need to be followed in order for the graph to function properly. These steps must be followed in this order:

- *Create the graph.* Call `NewAUGraph` (page 80) on an `AUGraph` instance.
- *Populate the graph*. Use `AUGraphNewNode` (page 81) and `AUGraphNewNodeSubGraph` (page 81) to populate the graph.

- *Make connections between nodes*. `AUGraphConnectNodeInput` (page 84) sets up connections between the nodes. Nodes may have multiple inputs and outputs, though sharing an input or output is not allowed. This is otherwise known as "fan in" and "fan out," and is not allowed in AUGraph.
- *Open the graph.* Up until this point, each node was being used in the abstract `AUNode` representation. Upon calling `AUGraphOpen` (page 87), each node is instantiated. This allows for properties to be set for each Audio Unit inside of the graph.
- *Set output sample rates and channel layouts.* A common error is for sample rates and channel layouts to be mismatched between nodes. If any format changes occur, it is imperative that sample rates and channel number be set for all Audio Unit outputs prior to initialization
- *Initialize the graph.* Once setup has occurred, `AUGraphInitialize` (page 87) may be called. This makes all the connections between the nodes, and initializes all of the Audio Units that are part of a connection. At the least, the output unit is initialized, even if no connections lead to it.
- *Start the graph*. Calling `AUGraphStart` (page 88) begins rendering, starting with the output unit and traversing through the graph.

Modifying an Audio Unit Graph

Once a graph has been created, its contents may be modified by adding and removing connections and nodes. These functions are provided for performing these actions:

- `AUGraphNewNode` (page 81).
- `AUGraphNewNodeSubGraph` (page 81)
- `AUGraphRemoveNode` (page 82)
- `AUGraphConnectNodeInput` (page 84)
- `AUGraphDisconnectNodeInput` (page 84)
- `AUGraphClearConnections` (page 85)

After the graph is initialized, any of these functions may be called, and the changes will occur immediately, meaning that the node will be initialized right way, and connections will be made immediately as well.

When a graph is running, however, changes to not immediately take effect. Calling any of these functions is allowed, but the actions they perform are queued. To apply the actions to a running graph, `AUGraphUpdate` (page 86) must be called.

Calling an update signals to the render thread that an update is ready to occur. When the render thread gets to a point in its cycle where updates are allowed (usually before and after a render), the update is actually performed. Before calling an update, check the format, sample rates, and channel layouts of the connections to avoid errors. If an error does occur, all updates are halted.

Closing an Audio Unit Graph

When audio data rendering is no longer needed, the graph may be stopped by calling `AUGraphStop` (page 88). This does not alter the graph in any way; it simply halts the pull on the output node. However, AUGraph uses a reference counting scheme to ensure that one process does not stop the graph while another may still be accessing it. Each `AUGraphStart()` invocation adds one to the reference, while each `AUGraphStop()` subtracts one from the reference. When the reference becomes zero, it then stops rendering. To determine if a graph is still running, use `AUGraphIsRunning` (page 89).

Uninitializing the graph is done by calling `AUGraphUninitialize` (page 88). First, doing so will stop audio render, no matter what the reference count for the graph is. Beyond that, it also calls the `Uninitialize()` function for each Audio Unit and subgraph.

If the graph is no longer needed, calling `AUGraphClose` (page 87) will close all the Audio Unit components in the graph, leaving only a nodal representation of the graph. As with uninitialization, if the graph is rendering audio data, calling this function halts the render.

It is worth noting that the graph's structure may be serialized using the `AUGraphGetNodeInfo` (page 83) and `AUGraphGetNodeConnections` (page 85) functions at any time the graph exists.

When the AUGraph is no longer needed, use `DisposeAUGraph` (page 81) to deallocate it.

Using Music Player and Music Sequence

Setting Up a Music Sequence

A music sequence is designed to hold various music tracks, which are intended to be logical groupings of events. To use a music sequence, these functions need to be called:

- `NewMusicSequence` (page 103) - This creates a new music sequence. The sequence, as is, contains only a tempo track.
- `MusicSequenceNewTrack` (page 104) - Call this function for each new track that you want in the sequence.

Also of note is the fact that you can reverse all the events in all of the tracks of a sequence by calling `MusicSequenceReverse` (page 109).

Adding Events to Tracks

Events trigger changes to the destination of a track. There are eight different types of events:

- `MIDIChannelMessage` (page 93) - For use with `MusicTrackNewMIDIChannelEvent` (page 113). These events will pass the assigned data on to the channel when triggered.
- `MIDINoteMessage` (page 92) - For use with `MusicTrackNewMIDINoteEvent` (page 112). These events will pass the information for a note to the targeted endpoint when triggered.
- `MIDIRawData` (page 93) - For use with `MusicTrackNewMIDIRawDataEvent` (page 113). These events will pass raw MIDI data on to an endpoint when triggered.
- `MIDIMetaEvent` (page 94) - For use with `MusicTrackNewMetaEvent` (page 114). These events should be used when MIDI meta data needs to be passed on to an endpoint.
- `MusicEventUserData` (page 95) - For use with `MusicTrackNewUserEvent` (page 115). This event will call a `MusicSequenceUserCallback` (page 121) callback, passing in the structure's user data to the callback. The callback is registered via `MusicSequenceSetUserCallback` (page 110), with each sequence allowing for one callback to be registered to it.
- `ExtendedNoteOnEvent` (page 95) - For use with `MusicTrackNewExtendedNoteEvent` (page 114). These events will send a note message to a music device audio unit when triggered.

- `ExtendedControlEvent` (page 96) - For use with `MusicTrackNewExtendedControlEvent` (page 114). These events will send a control message, changing the parameters of a music device audio unit when triggered.
- `ParameterEvent` (page 96) - For use with `MusicTrackNewParameterEvent` (page 115). These events will issue a parameter change in an audio unit when triggered.

Setting Destinations for Sequences and Tracks

A sequence or track must address either a MIDI endpoint or an audio unit (when used inside of an audio unit graph). All of the events belonging to a sequence or track will then be sent to its assigned destination.

An entire sequence can be assigned to an endpoint or a graph via:

- `MusicSequenceSetMIDIEndpoint` (page 106)
- `MusicSequenceSetAUGraph` (page 105)

When targeting a sequence to a specific graph, its tracks need to be assigned to units within the track; this is done via `MusicTrackSetDestNode` (page 110). Within this context, endpoints can still be addressed by a track using `MusicTrackSetDestMIDIEndpoint` (page 111).

Using the Tempo Track

Each sequence has a tempo track assigned to it, which can not be removed. This track is designed to control the rate of playback across the sequence's tracks. The units of measurement used here are beats-per-minute (bpm), which can be any floating point value. An event in the tempo track will change the playback rate to the new event's specified rate.

All of the events for this track must be of type `ExtendedTempoEvent` (page 97); no others are allowed in the tempo track, and this event type is not allowed in event tracks. Each tempo track starts out with one of these, specifying the initial playback rate. By default, this rate is 120 bpm, but it can be modified to be any floating point value.

To access the tempo track, call `MusicSequenceGetTempoTrack` (page 105). Once you acquire the tempo track, and tempo event can be added to it by calling `MusicTrackExtendedTempoEvent` (page 115) and passing in a new event. The event should have the intended rate in it. This means that, once the event is reached, the new rate will be used from that point on, until playback stops, or the next tempo event is reached. During that time, all events in all tracks within the sequence will occur at the new rate. For example, if the initial tempo was 120 bpm, and the next tempo event was set to occur at the 90th beat, it will occur after 45 seconds. If that event changes the tempo to 60 bpm, all of the events from that point on will happen at half the rate of the previous tempo. So if the next tempo event is set to occur at the 120th beat, will happen 75 seconds after playback has begun.

Disposing of Sequences and Tracks

When a sewuence is no longer needed, it may be disposed of. This is done by calling `DisposeMusicSequence` (page 103).

To delete a track and its accompanying events, call `MusicSequenceDisposeTrack` (page 104).

Getting Information about a Sequences and Tracks

A number of functions are provided with the Music Sequence API to get information about a sequence and its tracks:

- `MusicSequenceGetTrackCount` (page 104) - Returns the number of tracks in the current sequence.
- `MusicSequenceGetIndTrack` (page 105) - Returns a pointer towards a track for a particular index.
- `MusicSequenceGetTrackIndex` (page 105) - Returns the index of a track.
- `MusicSequenceGetAUGraph` (page 106) - Returns a pointer for the audio unit graph currently assigned to the sequence. Returns NULL if there is no graph assigned.
- `MusicTrackGetSequence` (page 110) - Returns a pointer to the sequence which contains the current track.
- `MusicTrackGetDestNode` (page 111) - Returns a pointer towards the node used by the selected track.
- `MusicTrackGetDestMIDIEndpoint` (page 111) - Returns a pointer towards the MIDI endpoint used by the selected track.

Using Music Track Properties

Properties are used to change the status of various tracks, as described in "Music Track Properties" (page 98). Use `MusicTrackGetProperty` (page 112) to retrieve the current value of any of the properties, and `MusicTrackSetProperty` (page 112) to change the value of the property.

Accessing Events within a Track

To gain access to the events within a track, the track needs to be iterated over. Iterating involves creating a new iterator, and then moving forward or backward between the events within the track. These functions are used when setting up an iterator and iterating on a track:

- `NewMusicEventIterator` (page 118) - Creates a new iterator.
- `DisposeMusicEventIterator` (page 118) - Disposes of the iterator.
- `MusicEventIteratorNextEvent` (page 118) - Moves the iterator to the next event.
- `MusicEventIteratorPreviousEvent` (page 119) - Moves the iterator to the previous event
- `MusicEventIteratorGetEventInfo` (page 119) - Retrieves information about the current event.
- `MusicEventIteratorSetEventInfo` (page 119) - Sets information about the current event.
- `MusicEventIteratorDeleteEvent` (page 120) - Removes the event from the track.
- `MusicEventIteratorSetEventTime` (page 120) - Moves the event to the new time.
- `MusicEventIteratorHasPreviousEvent` (page 120) - Returns a boolean signifying if there is an event before the iterator.
- `MusicEventIteratorHasNextEvent` (page 120) - Returns a boolean signifying if there is an event after the iterator.
- `MusicEventIteratorHasCurrentEvent` (page 121) - Returns a boolean signifying if the iterator currently points towards an event, or is at the end of the track.

Editing a Track

Events within a track can be modified based on their placement within a track. The idea is to be able to grab the events within a certain amount period of time and then to move them elsewhere, or delete them.

To move events within a track, use `MusicTrackMoveEvents` (page 116). All you need to do is to specify the range of events to move, and where to move them to.

The `NewMusicTrackFrom` (page 116) function will take the specified range of events, and will create a new track with the range in it.

`MusicTrackClear` (page 116) will remove the events in the given range, while `MusicTrackCut` (page 116) will remove the given range, and move the events after the range up to fill the space left by the cut.

Use `MusicTrackCopyInsert` (page 117) to copy a series of event from one track to another. Doing so with this function move the events behind the insertion point back to the end of the range in the destination track.

Finally, `MusicTrackMerge` (page 117) will take the source range and merge it with the events following the insertion point in the destination.

Setting Up a Music Player

A music player is associated with a music sequence, in a one-to-one relationship. The player keeps track of the playhead for the sequence, and allows for movement within the sequence. Activating and stopping the sequence is done via a player.

- `NewMusicPlayer` (page 99) - Creates a new music player.
- `DisposeMusicPlayer` (page 99) - Disposes of the music player. Note that this does not dispose of the sequence attached to the player.
- `MusicPlayerSetSequence` (page 99) - Sets the sequence controlled by the player.
- `MusicPlayerStart` (page 102) - Begins playback of the sequence.
- `MusicPlayerStop` (page 102) - Halts sequence playback.
- `MusicPlayerIsPlaying` (page 102) - Returns a boolean signifying if the player is in use.

Of note is when the playhead is moved within the player. This is done using `MusicPlayerSetTime` (page 100), which also prerolls the sequence for you. Prerolling is when the sequence is prepared to begin in mid-sequence, with all parameters and endpoints being adjusted to the points they should be at the playhead. To determine what time the player is currently at, use `MusicPlayerGetTime` (page 100).

If a new event is added to a sequence, it will need to be manually prerolled using `MusicPlayerPreroll` (page 101).

Reading in Standard MIDI Files or MIDI Data

The Music Player API is used to read in MIDI files. To do so, simply call `MusicSequenceLoadSMF` (page 106), specifying the file from which the data is to be read in, or `MusicSequenceLoadSMFData` (page 107) when the MIDI is to be read in from memory. When these functions are used, the MIDI data is parsed and placed, as events, in a track inside of a sequence.

Calling `MusicSequenceLoadSMFWithFlags` (page 107) and `MusicSequenceLoadSMFDataWithFlags` (page 107), with kMusicSequenceLoadSMF_ChannelsToTracks passed in as the flag will result in each channel in the MIDI data being parsed into its own track in the sequence. Beyond that, any meta data that is found in the MIDI sequence in placed in the last track of the sequence.

Saving MIDI Data

When you want to save incoming MIDI data to disk, first you need to capture the incoming MIDI data. The data then needs to be parsed and placed into a sequence, specifically into a track. You will need to use the `MusicPlayerGetBeatsForHostTime` (page 101) function to determine how far into the sequence the new MIDI event will need to be. This can only be done as the sequence in running, so that calling this will return the current beat in the sequence when it is invoked.

Once all of the incoming MIDI data has been captured and placed into a sequence, it needs to be saved to disk. To do this, call the `MusicSequenceSaveSMF` (page 108) function, if saving the data to disk, or, if saving it to memory, call `MusicSequenceSaveSMFData` (page 108).

Audio Toolbox Reference

This reference section describes the constants, data types and functions that comprise the Audio Toolbox framework available for Mac OS X.

Audio Converter Reference

Audio converters are designed to meet a developer's encoding and decoding needs. It allows for conversions between most conceivable combinations of input and output formats, assuming proper codecs are available on the system.

Audio Converter Types

Defined Data Types

Typedefs are used to simplify the declaration of converters and the use of properties in the context of an audio converter.

- `typedef struct OpaqueAudioConverter* AudioConverterRef`
- `typedef UInt32 AudioConverterPropertyID`

Data Structures

AudioConverterPrimeInfo

Stores information regarding the number of frames used in priming input for the current codec.

```
typedef struct AudioConverterPrimeInfo {
UInt32 leadingFrames;
UInt32 trailingFrames;
} AudioConverterPrimeInfo;
```

Discussion
An instance of this structure is input via the `kAudioConverterPrimeInfo` property. The instance works in conjunction with the `kAudioConverterPrimeMethod` property, which specifies the priming method used by the codec. When a priming method is in use, the members of this structure are used to specify the number of leading and trailing frames (when using `kConverterPrimeMethod_Pre`), or just the number of trailing frames (when using `kConverterPrimeMethod_Normal`), for all input packets.

Availability
Available in Mac OS X v10.3 and later.

Declared In
`AudioConverter.h`

Audio Converter Constants

Constants are provided for the developer's convenience. They provide a consistent set of values for various aspects of a converter's operations, and may be appended by the developer at any time.

Converter Quality Settings

Used by `kAudioConverterSampleRateConverterQuality` to set the relative quality of the conversion.

```
kAudioConverterQuality_Max = 0x7F
kAudioConverterQuality_High = 0x60
kAudioConverterQuality_Medium = 0x40
kAudioConverterQuality_Low = 0x20
kAudioConverterQuality_Min = 0
```

Note: The relative quality of a conversion is an arbitrary aspect of the codec used, and may or may not alter the quality of the resulting conversion.

Priming Method Selectors

Specifies the priming method currently in use, as referenced in the `kAudioConverterPrimeMethod` property.

`kConverterPrimeMethod_Pre = 0`
: When `kAudioConverterPrimeMethod` is set to this value, the converter will expect that the packet be primed with both leading and trailing frames.

`kConverterPrimeMethod_Normal = 1`
: Set `kAudioConverterPrimeMethod` to this when the trailing frames are needed for the conversion; leading frames are assumed to silent.

`kConverterPrimeMethod_None = 2`
: Used when the leading and trailing frames are assumed to be silent and priming is not needed.

Audio Converter Properties

The properties are used to query a converter for its settings, and sometimes, to modify those properties.

`kAudioConverterPropertyMinimumInputBufferSize = 'mibs'`
: Returns a `UInt32` containing the size of the smallest input buffer size, in bytes, that can be supplied into the `AudioConverterConvertBuffer()` function or the `*AudioConverterInputProc` callback.

`kAudioConverterPropertyMinimumOutputBufferSize = 'mobs'`
: Returns a `UInt32` containing the size of the smallest buffer that will bereturned as a result of `AudioConverterConvertBuffer()` or `AudioConverterFillBuffer()`.

`kAudioConverterPropertyMaximumInputBufferSize = 'xibs'`
: Returns a `UInt32` containing the largest buffer size that will requested by `*AudioConverterInputProc`; returns `0xFFFFFFFF` if the value depends on the size of the input.

`kAudioConverterPropertyMaximumInputPacketSize = 'xips'`
: Returns a `UInt32` containing the size, in bytes, of the largest packet of data that may be input.

`kAudioConverterPropertyMaximumOutputPacketSize = 'xops'`
: Returns a `UInt32` containing the size, in bytes, of the largest packet of data that will be output.

`kAudioConverterPropertyCalculateInputBufferSize = 'cibs'`
: On input, takes a `UInt32` with the desired output size, in bytes; returns the number of bytes needed as input to generate the requested output.

`kAudioConverterPropertyCalculateOutputBufferSize = 'cobs'`
: On input, takes a `UInt32` with the desired input size, in bytes; returns the number of bytes returned as output for the requested input.

`kAudioConverterPropertyInputCodecParameters = 'icdp'`
: Takes in a buffer of untyped data for private use relative and specific to the format.

`kAudioConverterPropertyOutputCodecParameters = 'ocdp'`
: Takes in a buffer of untyped data for private use relative and specific to the format.

`kAudioConverterSampleRateConverterAlgorithm = 'srci'`
: *Deprecated.* Use `kAudioConverterSampleRateConverterQuality` instead.

`kAudioConverterSampleRateConverterQuality = 'srcq'`
: Specifies the quality of the sample rate conversion, using the "Converter Quality Settings" (page 59).

`kAudioConverterPrimeMethod = 'prmm'`
: Specifies the priming method, using the "Priming Method Selectors" (page 59).

`kAudioConverterPrimeInfo = 'prim'`
: Returns in a pointer to an `AudioConverterPrimeInfo` (page 58) instance.

`kAudioConverterChannelMap = 'chmp'`
: Takes an array of `SInt32` values where the index represents an output channel and the value stored at the index in the array is the connecting input channel; the size of the array is the number of output channels.

`kAudioConverterDecompressionMagicCookie = 'dmgc'`
: Takes a `void` pointer towards the magic cookie that may be required to decompress the data.

`kAudioConverterCompressionMagicCookie = 'cmgc'`
: Returns a `void` pointer towards the magic cookie used to compress the output data; may be passed back via `kAudioConverterDecompressionMagicCookie` for decompressing the data.

Audio Converter Functions

AudioConverterNew

Creates a new audio converter.

```
extern OSStatus AudioConverterNew(
const AudioStreamBasicDescription* inSourceFormat,
const AudioStreamBasicDescription* inDestinationFormat, AudioConverterRef*
outAudioConverter
);
```

Discussion
This function takes in two `AudioStreamBasicDescription` instances, one for the source, and one for the destination, sets up all of the internal links needed for the conversion, and returns a pointer for the new converter. Note that if the setup fails, an error is returned which specifies the error that was encountered.

Availability
Available in Mac OS X v10.1 and later.

Declared In
`AudioConverter.h`

AudioConverterDispose

Destroys an audio converter.

```
extern OSStatus AudioConverterDispose(AudioConverterRef inAudioConverter);
```

Discussion
This function deallocates the memory used by *`inAudioConverter`*.

Availability
Available in Mac OS X v10.1 and later.

Declared In
`AudioConverter.h`

AudioConverterReset

Resets the audio converter to its post-initialization state.

```
extern OSStatus AudioConverterReset(AudioConverterRef inAudioConverter);
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`AudioConverter.h`

AudioConverterGetPropertyInfo

Retrieves the size and writable state of the data belonging to the queried property.

```
extern OSStatus AudioConverterGetPropertyInfo(
AudioConverterRef inAudioConverter,
AudioConverterPropertyID inPropertyID,
UInt32* outSize,
Boolean* outWritable
);
```

Discussion
The `outSize` value returned reflects the size, in bytes, of the data returned by calling `AudioConverterGetProperty()` with the respective property.

Availability
Available in Mac OS X v10.1 and later.

Declared In
`AudioConverter.h`

AudioConverterGetProperty

Returns the requested property data.

```
extern OSStatus AudioConverterGetProperty(
AudioConverterRef inAudioConverter,
AudioConverterPropertyID inPropertyID,
UInt32* ioPropertyDataSize,
void* outPropertyData
);
```

Discussion
The *`ioPropertyDataSize`* parameter should be the value obtained from calling `AudioConverterGetPropertyInfo()`; the output value of `ioPropertyDataSize` will be the actual data size of the returned data, for reference.

Availability
Available in Mac OS X v10.1 and later.

Declared In
`AudioConverter.h`

AudioConverterSetProperty

Sets the property data to `inPropertyData`.

```
extern OSStatus AudioConverterSetProperty(
AudioConverterRef inAudioConverter,
AudioConverterPropertyID inPropertyID,
UInt32 inPropertyDataSize,
const void* inPropertyData
);
```

Discussion
The *inPropertyDataSize* should be the size of data being input, and *inPropertyData* should point to the data to be set for *inPropertyID*.

Availability
Available in Mac OS X v10.1 and later.

Declared In
AudioConverter.h

AudioConverterInputDataProc

Should provide data for `AudioConverterFillBuffer()`.

```
typedef OSStatus (*AudioConverterInputDataProc) (
AudioConverterRef inAudioConverter,
UInt32* ioDataSize,
void** outData,
void* inUserData
);
```

Discussion
Deprecated. On input, `ioDataSize` will be the amount of data the converter needs to fill its buffer; on output, this value should reflect the amount of the data provided (if there is no more input data available, `0` should be returned).

AudioConverterFillBuffer

Fills the provided buffer with converted data.

```
extern OSStatus AudioConverterFillBuffer(
AudioConverterRef inAudioConverter,
AudioConverterInputDataProc inInputDataProc,
void* inInputDataProcUserData,
UInt32* ioOutputDataSize,
void* outOutputData
);
```

Discussion
Deprecated. Uses the provided `inInputDataProc` callback to acquire data, converts it, and places the converted data in. `outOutputData`. Deprecated since it can only work with a single buffer. Use `AudioConverterFillComplexBuffer()` instead.

Availability
Available in Mac OS X v10.1 and later.
Deprecated in Mac OS X v10.5.

Declared In
`AudioConverter.h`

AudioConverterComplexInputDataProc

Should provide `AudioConverterFillComplexBuffer()` with data for conversion.

```
typedef OSStatus (*AudioConverterComplexInputDataProc) (
AudioConverterRef inAudioConverter,
UInt32* ioNumberDataPackets,
AudioBufferList* ioData,
AudioStreamPacketDescription** outDataPacketDescription,
void* inUserData
);
```

Discussion
`AudioConverterFillComplexBuffer()` will use this callback to acquire data to convert. The returned data will be an `AudioBufferList`, meaning that the data should be in separate indices, one for each channel. Use `inUserData` for any data the callback may need passed to it. The caller will pass the number of packets requested in `ioNumberDataPackets`, and upon completion, the callback should return the number of packets actually provided, or `0` if there is no data left to provide. The resulting packet format is specified in `outDataPacketDescription`.

AudioConverterFillComplexBuffer

Fills the `AudioBufferList` with converted data.

```
extern OSStatus AudioConverterFillComplexBuffer(
AudioConverterRef inAudioConverter,
AudioConverterComplexInputDataProc inInputDataProc,
void* inInputDataProcUserData,
UInt32* ioOutputDataPacketSize,
AudioBufferList* outOutputData,
AudioStreamPacketDescription* outPacketDescription
);
```

Discussion
Using the callback provided in `inInputDataProc`, this function will convert input data using `inAudioConverter` and will place the resulting converted data in `outOutputData`. Any relevant data for the callback should be passed in via `inInputDataProcUserData`, while `outPacketDescription` will contain the format of the returned data. On input, *`ioOutDataPacketSize`* should contain the number of packets requested, and as output, will contain the number of packets returned.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioConverter.h`

Audio Converter Result Codes

These values are returned when errors occur.

```
kAudioConverterErr_FormatNotSupported = 'fmt?'
kAudioConverterErr_OperationNotSupported = 0x6F703F3F
kAudioConverterErr_PropertyNotSupported = 'prop'
kAudioConverterErr_InvalidInputSize = 'insz'
kAudioConverterErr_InvalidOutputSize = 'otsz'
kAudioConverterErr_UnspecifiedError = 'what'
kAudioConverterErr_BadPropertySizeError = '!siz'
kAudioConverterErr_RequiresPacketDescriptionsError = '!pkd'
```

Audio Format Reference

The audio format system is provided to allow the developer to get more information about certain aspects of `AudioStreamBasicDescription` and `AudioChannelLayout` instances, and other important pieces of information.

Audio Format Types

Defined Data Types

The `AudioFormatPropertyID` typedef is used to hold the property ID being queried using the audio format functions.

- `typedef UInt32 AudioFormatPropertyID`

Data Structures

AudioPanningInfo

Stores information about the position of sound sources.

```
typedef struct AudioPanningInfo {
UInt32 mPanningMode;
UInt32 mCoordinateFlags;
Float32 mCoordinates[3];
AudioChannelLayout* mOutputChannelMap;
} AudioPanningInfo;
```

Discussion
The *`mPanningMode`* value is based on the panning mode constants. The value of *`mCoordinateFlags`* will be based on the Coordinate Flag constants. The precise coordinates of the source is located in *`mCoordinates`*, and the *`mOutputChannelMap`* points to an instance of an `AudioChannelLayout` (page 164) (specified in `CoreAudioTypes.h`), which tracks channel layouts in hardware and in files.

Availability
Available in Mac OS X v10.3 and later.

Declared In
`AudioFormat.h`

Audio Format Constants

Constants are provided for the developer's convenience. They provide a consistent set of values for various aspects of a converter's operations.

Panning Modes

These constants define various panning algorithms that can be specified in an `AudioPanningInfo` (page 65) instance.

`kPanningMode_SoundField = 3`
: An Ambisonic format.

`kPanningMode_VectorBasedPanning = 4`
: A format for panning between two speakers.

Coordinate Flags

Used by the `mCoordinateFlags` value in the `AudioPanningInfo` structure; found in `CoreAudioTypes.h`.

`kAudioChannelFlags_RectangularCoordinates = (1L<<0)`
: Use if cartesian coordinates are used for speaker positioning; either this or spherical coordinates must be chosen.

`kAudioChannelFlags_SphericalCoordinates = (1L<<1)`
: Use if spherical coordinates are used for speaker positioning; either this or cartesian coordinates must be chosen.

`kAudioChannelFlags_Meters = (1L<<2)`
: Use when units are in meters; if not set, then the units are relative to the coordinate system chosen.

Audio Format Properties

The audio format tool uses the property system to get various pieces of information about structures used in Core Audio.

AudioStreamBasicDescription Properties

When the specifier parameter for `AudioFormatGetPropertyInfo()` and `AudioFormatGetProperty()` is an `AudioStreamBasicDescription` instance, these properties may be queried.

`kAudioFormatProperty_FormatInfo = 'fmti'`
: Returns an `AudioStreamBasicDescription` whose values contain information about the specifier's format.

`kAudioFormatProperty_FormatIsVBR = 'fvbr'`
: Returns a `UInt32` where a non-zero value means that the format has a variable bit rate (VBR).

`kAudioFormatProperty_FormatIsExternallyFramed = 'fexf'`
: Returns a `UInt32`, where a non-zero value indicates that the format is externally framed.

`kAudioFormatProperty_FormatName = 'fnam'`
: Returns a `CFStringRef` containing the name of the specified format.

`kAudioFormatProperty_AvailableEncodeChannelLayouts = 'aecl'`
: Takes in an `AudioStreamBasicDescription` and returns an `AudioChannelLayoutTag` array containing Audio Channel Layout constants.

AudioChannelLayout Properties

`kAudioFormatProperty_ChannelLayoutForTag = 'cmpl`
: Takes an "Channel Layout Tags" (page 167) value (as specified in `CoreAudioTypes.h`) as the specifier and returns an `AudioChannelLayout` (page 164) with all of its members filled with their respective versions of the input data.

`kAudioFormatProperty_TagForChannelLayout = 'cmpt'`
: Takes an `AudioChannelLayout` as the specifier and returns an `AudioChannelLayoutTag` with all of its members filled with their respective versions of the input data.

`kAudioFormatProperty_ChannelLayoutForBitmap = 'cmpb'`
: Takes in a `UInt32` that contains a layout bitmap and returns an `AudioChannelLayout` with all of its members filled with their respective versions of the input data.

`kAudioFormatProperty_BitmapForLayoutTag = 'bmtg'`
: Takes in a "Channel Layout Tags" (page 167) value and returns a `UInt32` with the bitmap of the channel layout.

`kAudioFormatProperty_ChannelLayoutName = 'lonm'`
: Takes in an `AudioChannelLayout` and returns a `CFStringRef` with the name of the channel.

`kAudioFormatProperty_ChannelName = 'cnam'`
: Takes in an `AudioChannelDescription` with a populated `mChannelLabel` value, and returns a `CFStringRef` with the name of the channel.

`kAudioFormatProperty_MatrixMixMap = 'mmap'`
: Takes in an array of two `AudioChannelLayout` pointers, the first to the input and the second to the output, and returns a two dimensional `Float32` array, with the input being the rows and the output being the columns, where the value at a coordinate is the gain that needs to be applied to the input to achieve the output at that channel.

`kAudioFormatProperty_NumberOfChannelsForLayout = 'nchm'`
: Takes in an `AudioChannelLayout` as the specifier and returns a `UInt32` with the number of channels represented in the layout.

`kAudioFormatProperty_PanningMatrix = 'panm'`
: Takes in an `AudioPanningInfo` instance and returns a `Float32` array where each channel receives a volume level for each channel in the `AudioPanningInfo`'s `AudioChannelLayout` array.

Other Properties

These are other properties that involve discovering encoding and decoding formats and available sample and bit rates.

`kAudioFormatProperty_EncodeFormatIDs = 'acif'`
: Does not take a specifier (set to `NULL`), and returns a `UInt32` array containing "Format IDs" (page 161) (specified in `CoreAudioTypes.h`) for valid input formats into a converter.

`kAudioFormatProperty_DecodeFormatIDs = 'acof'`
: Does not take a specifier (set to `NULL`), and returns a `UInt32` array containing Format ID constants for valid output formats into a converter.

`kAudioFormatProperty_AvailableEncodeSampleRates = 'aesr'`
: Takes in a Format ID constant and returns an `AudioValueRange` (page 159) with all of the available sample rates.

`kAudioFormatProperty_AvailableEncodeBitRates = 'aebr'`
: Takes in a Format ID constant and returns an `AudioValueRange` with all of the available bit rates.

Audio Format Functions

These functions comprise the Audio Format property management system. These functions work by providing property ID, which notifies them as to which action should be performed, and a specifier, which is the data on which the operation is to be performed.

AudioFormatGetPropertyInfo

Retrieves the size of the data to be returned by the property.

```
extern OSStatus AudioFormatGetPropertyInfo(
AudioFormatPropertyID inPropertyID,
UInt32 inSpecifierSize,
void* inSpecifier,
UInt32* outPropertyDataSize
);
```

Availability
Available in Mac OS X v10.3 and later.

Declared In
`AudioFormat.h`

AudioFormatGetProperty

Retrieves the property information for the given property ID and selected specifier.

```
extern OSStatus AudioFormatGetProperty(
AudioFormatPropertyID inPropertyID,
UInt32 inSpecifierSize,
void* inSpecifier,
UInt32* ioPropertyDataSize,
void* outPropertyData
);
```

Availability
Available in Mac OS X v10.3 and later.

Declared In
`AudioFormat.h`

Audio Format Result Codes

These values are returned when errors occur.

`kAudioFormatUnspecifiedError = 'what'`

```
kAudioFormatUnsupportedPropertyError = 'prop'
kAudioFormatBadPropertySizeError = '!siz'
kAudioFormatBadSpecifierSizeError = '!spc'
kAudioFormatUnsupportedDataFormatError = 'fmt?'
kAudioFormatUnknownFormatError = '!fmt'
```

Audio File Reference

The Audio File API allows for opening and saving audio files in various formats, for later use.

Audio File Types

Defined Data Types

Typedefs are used to simplify the declaration of converters and the use of properties in the context of an audio file.

- `typedef struct OpaqueAudioFileID *AudioFileID`
- `typedef UInt32 AudioFilePropertyID`

Data Structures

AudioFileTypeAndFormatID

Used by the `kAudioGlobalInfo_AvailableStreamDescriptionForFormat` property to query for `AudioStreamBasicDescriptions` based on format and file type.

```
typedef struct AudioFileTypeAndFormatID{
UInt32 mFileType;
UInt32 mFormatID;
} AudioFileTypeAndFormat;
```

Discussion
The value of `mFileType` is a "File Types" (page 69) value, while `mFormatID` is from the "Format IDs" (page 161) in `CoreAudioTypes.h`.

Constants

Constants are provided for the developer's convenience. They provide a consistent set of values for various aspects of an audio file.

File Types

These constants are used to specify file types when using functions and structures related to audio files.

```
kAudioFileAIFFType = 'AIFF'
kAudioFileAIFCType = 'AIFC'
kAudioFileWAVEType = 'WAVE'
kAudioFileSoundDesigner2Type = 'Sd2f'
```

```
kAudioFileNextType = 'NeXT'
kAudioFileMP3Type = 'MPG3'
kAudioFileAC3Type = 'ac-3'
kAudioFileAAC_ADTSType = 'adts'
```

Audio File Properties

The Audio File API uses the property system to get and set information about files and global settings.

Audio File Properties

These properties are to be used when getting and setting information about an particular audio file.

`kAudioFilePropertyFileFormat = 'ffmt'`
: Passes a `UInt32` that identifies the file's format, based on the "Format IDs" (page 161) found in `CoreAudioTypes.h`.

`kAudioFilePropertyDataFormat = 'dfmt'`
: Passes an `AudioStreamBasicDescription` that describes the file's format.

`kAudioFilePropertyIsOptimized = 'optm'`
: Returns a `UInt32` with either a value of `0`, meaning that the file is not optimized, and therefore, not ready to be written to, or a value of `1`, meaning that the file is currently optimized.

`kAudioFilePropertyMagicCookieData = 'mgic'`
: Passes a `void` pointer towards memory set up for use as a magic cookie.

`kAudioFilePropertyAudioDataByteCount = 'bcnt'`
: Passes a `UInt64` that contains the size of the audio data in the file, in bytes.

`kAudioFilePropertyAudioDataPacketCount = 'pcnt'`
: Passes a `UInt64` that contains the size of the audio data in the file, in packets.

`kAudioFilePropertyMaximumPacketSize = 'psze'`
: Passes a `UInt32` that contains the maximum packet size in the file.

`kAudioFilePropertyDataOffset = 'doff'`
: Passes an `SInt64` that contains offset of where the audio data begins inside the file.

`kAudioFilePropertyChannelLayout = 'cmap'`
: Passes an `AudioChannelLayout`, specified in `CoreAudioTypes.h`, used in the file.

`kAudioFilePropertyDeferSizeUpdates = 'dszu'`
: Passes a `UInt32` where a value of `1` means that the file size information in the file header is updated only when the file is read, optimized, or closed; a value of `0` denotes that the header is updated with every write.

`kAudioFilePropertyDataFormatName = 'fnme'`
: Deprecated in favor of the `kAudioFormatProperty_formatName` property, available from Audio Format "Audio Format Properties" (page 66).

Audio File Global Info Properties

The Global Info Properties are used to retrieve general information about the environment that is being used. Many of these properties require a specifier for use, meaning that, in addition to passing a property ID, a piece of information being queried upon is passed in as a specifier.

`kAudioFileGlobalInfo_ReadableTypes = 'afrf'`
: Takes `NULL` as its specifier, and returns a `UInt32` array containing the File Type constants which are readable.

`kAudioFileGlobalInfo_WritableTypes = 'afwf'`
: Takes `NULL` as its specifier, and returns a `UInt32` array containing the File Type constants which are writable.

`kAudioFileGlobalInfo_FileTypeName = 'ftnm'`
: Takes a `UInt32` containing a File Type constant as its specifier and returns a `CFString` containing the name of the file type.

`kAudioFileGlobalInfo_ExtensionsForType = 'fext'`
: Takes a `UInt32` containing a File Type constant as its specifier and returns a `CFArray` of `CFString` values containing the file extensions recognized for this file type.

`kAudioFileGlobalInfo_AllExtensions = 'alxt'`
: Takes `NULL` as its specifier and returns a `CFArray` of `CFString` values containing all of the recognizable file extensions.

`kAudioFileGlobalInfo_AvailableFormatIDs = 'fmid'`
: Takes a `UInt32` containing a File Type constant as its specifier and returns a `UInt32` array containing format ID constants for formats readable by audio file.

`kAudioFileGlobalInfo_AvailableStreamDescriptionsForFormat = 'sdid'`
: Takes an `AudioFileTypeAndFormatID` instance as its specifier and returns an `AudioStreamBasicDescription` array whose elements correspond with the elements in the specifier.

Audio File Functions

These functions are provided to access the functionality of the Audio File API.

Data Handling

AudioFileCreate

Creates a new file using the descriptions provided.

```
extern OSStatus AudioFileCreate(
const FSRef *inParentRef,
CFStringRef inFileName,
UInt32 inFileType,
const AudioStreamBasicDescription *inFormat,
UInt32 inFlags,
FSRef *outNewFileRef,
AudioFileID *outAudioFile
);
```

Discussion
The directory that the file to be place into is provided with `inParentRef`, the name of the file is contained within `inFileName`, a File Type constant must be provided with `inFileType`, the format must be specified using `inFormat`, `inFlag` contains flags for opening and creating the file (currently undefined; should be set to `0`), and `outNewFileRef` is provided for file system use, while *`outAudioFile`* is for use with other audio file functions.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFileInitialize

Wipes clean a existing file to prepare it for writing.

```
extern OSStatus AudioFileInitialize(
const FSRef *inFileRef,
UInt32 inFileType,
const AudioStreamBasicDescription *inFormat,
UInt32 inFlags,
AudioFileID *outAudioFile
);
```

Discussion
The `inFileRef` is the file to be initialized, with the `inFileType` being a File Type constant value, `inFormat` being an `AudioStreamBasicDescription` specifying the format for the file, `inFlags` being relevant creation and opening flags (currently undefined; should be set to `0`), and `outAudioFile` being an `AudioFileID` for use with other audio file functions.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFileOpen

Opens a file while preserving its contents.

```
extern OSStatus AudioFileOpen (
const FSRef *inFileRef,
SInt8 inPermissions,
UInt32 inFlags,
AudioFileID *outAudioFile
);
```

Discussion
The `inFileRef` should be a reference to an existing file, `inPermissions` being the permissions for the file, as used by `FSOpenFork()`, and `inFlags`, currently undefined, should be set to `0`; `outAudioFile` is a file instance that will be returned for use in other audio file functions.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFile_ReadProc

Should read the contents of a file.

```
typedef OSStatus (*AudioFile_ReadProc)(
void * inRefCon,
SInt64 inPosition,
ByteCount requestCount,
void *buffer,
ByteCount* actualCount
);
```

Discussion
This callback needs to be provided by the developer for the purpose of reading the audio data for use with `AudioFormatInitializeWithCallbacks()` and `AudioFormatOpenWithCallbacks()`. Constants for use by the callback are passed in via `inRefCon`, the position to be read from will be passed in via `inPosition`, the number of bytes requested is passed in via `requestCount`, the processed data is passed in via `buffer`, and `actualCount` returns the number of bytes returned.

AudioFile_WriteProc

Should write the given buffer to the file.

```
typedef OSStatus (*AudioFile_WriteProc)(
void * inRefCon,
SInt64 inPosition,
ByteCount requestCount,
const void *buffer,
ByteCount* actualCount
);
```

Discussion
This callback needs to be provided by the developer for the purpose of writing to a file. Constants for use by the callback are passed in via `inRefCon`, the position to be read from will be passed in via `inPosition`, the number of bytes requested is passed in via `requestCount`, the data processed to is passed in via `buffer`, and `actualCount` returns the number of bytes written.

AudioFile_GetSizeProc

Should provide the size of the file to the caller.

```
typedef SInt64 (*AudioFile_GetSizeProc)(void * inRefCon);
```

Discussion
This callback should return an `SInt32` with the audio stream data size to the caller. If any constants need to be passed to the callback, their values should be pointed to by `inRefCon`.

AudioFile_SetSizeProc

Should set the file size to the passed value.

```
typedef OSStatus (*AudioFile_SetSizeProc)(
void * inRefCon,
SInt64 inSize
);
```

Discussion
This callback should set the size of the file to `inSize`, while `inRefCon` is provided to pass any needed arguments to the callback.

AudioFileInitializeWithCallbacks

Initializes an audio file using the provided callbacks.

```
extern OSStatus AudioFileInitializeWithCallbacks(
void * inRefCon,
AudioFile_ReadProc inReadFunc,
AudioFile_WriteProc inWriteFunc,
AudioFile_GetSizeProc inGetSizeFunc,
AudioFile_SetSizeProc inSetSizeFunc,
UInt32 inFileType,
const AudioStreamBasicDescription *inFormat,
UInt32 inFlags,
AudioFileID *outAudioFile
);
```

Discussion
This function will wipe the data target clean and set the various attributes using *`inFileType`*, *`inFormat`*, and *`inFlags`*. The callbacks need to be provided by the developer, according to the callback specifications elsewhere in this reference. Upon completion, *`outAudioFile`* will contain a reference to a file instance, for use with other audio file functions.

Availability
Available in Mac OS X v10.3 and later.

Declared In
`AudioFile.h`

AudioFileOpenFileWithCallbacks

Opens the file an prepares it for use.

```
extern OSStatus AudioFileOpenWithCallbacks(
void * inRefCon,
AudioFile_ReadProc inReadFunc,
AudioFile_WriteProc inWriteFunc,
AudioFile_GetSizeProc inGetSizeFunc,
AudioFile_SetSizeProc inSetSizeFunc,
UInt32 inFlags,
AudioFileID *outAudioFile
);
```

Discussion
Using this function will prepare the target data, while the callbacks specified here will be used when reading, writing, and modifying the data. This function is provided to allow for the use of Audio File's APIs with sources other than files.

AudioFileClose

Closes the file.

```
extern OSStatus AudioFileClose(AudioFileID inAudioFile);
```

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFileOptimize

Optimizes the file.

```
extern OSStatus AudioFileOptimize(AudioFileID inAudioFile);
```

Discussion
Optimizing a file will prepare it for any data which may be appended to the end of it. This is a costly operation and should not be performed during a process-intensive routine. The `kAudioFilepropertyIsOptimized` flag is available to determine whether or not the file is optimized.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFileReadBytes

Reads in a certain number of bytes from the file.

```
extern OSStatus AudioFileReadBytes(
AudioFileID inAudioFile,
Boolean inUseCache,
SInt64 inStartingByte,
UInt32 *ioNumBytes,
void *outBuffer
);
```

Discussion
Here, `inAudioFile` is the file being read from, `inStartingByte` is the point from which to read from, `ioNumBytes` being the amount to read, and `outBuffer` is where the read data is stored. To cache the read, set `inUseCache` to true.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFileWriteBytes

Write the contents of the buffer to the file.

```
extern OSStatus AudioFileWriteBytes(
AudioFileID inAudioFile,
Boolean inUseCache,
SInt64 inStartingByte,
UInt32 *ioNumBytes,
void *inBuffer
);
```

Discussion
Specify the file to be written to with *inAudioFile*, where to write within the file by specifying `inStartingByte`, how much is to be written using `ioNumBytes` (and verifying how much was written at output, as well), and the data to be written should be pointed to by `inBuffer`. To cache the written data, set `inUseCache` to true.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFileReadPackets

Reads in a certain number of packets from the input file.

```
extern OSStatus AudioFileReadPackets(
AudioFileID inAudioFile,
Boolean inUseCache,
UInt32 *outNumBytes,
AudioStreamPacketDescription *outPacketDescriptions,
SInt64 inStartingPacket,
UInt32 *ioNumPackets,
void *outBuffer
);
```

Discussion
This function reads in the contents of the file by packet, starting at `inStartingPoint`. The packets that have been read are described in `outPacketDescriptions`, while the number of packets is specified in `ioNumPackets` (with the actual number of packets read being the return value), and the size, in bytes, of the read in packets returned in `outNumBytes`. If the read should be cached, set `inUseCache` to true.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFileWritePackets

Writes the buffer to the file, by packets.

```
extern OSStatus AudioFileWritePackets(
AudioFileID inAudioFile,
Boolean inUseCache,
UInt32 inNumBytes,
AudioStreamPacketDescription *inPacketDescriptions,
SInt64 inStartingPacket,
UInt32 *ioNumPackets,
void *inBuffer
);
```

Discussion
When writing to `inAudioFile`, specify the starting index as `inStartingPacket`, the format of the packet as defined in `inPacketDescrptions`, the size of the write as `inNumBytes`, and the number of packets to be written in `ioNumPackets`. If the write should be cached, set `inUseCache` to true.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

Property Access

AudioFileGetPropertyInfo

Returns the size of the data that will be returned for the property.

```
extern OSStatus AudioFileGetPropertyInfo(
AudioFileID inAudioFile,
AudioFilePropertyID inPropertyID,
UInt32 *outDataSize,
UInt32 *isWritable
);
```

Discussion
The file being queried should be passed in as `inAudioFile`, while the property being queried is passed in as `inPropertyID`. The size of the resulting data is returned in `outDataSize`, and `isWritable` will reflect if the data is modifiable.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFileGetProperty

Returns the data for the specified property.

```
extern OSStatus AudioFileGetProperty(
AudioFileID inAudioFile,
AudioFilePropertyID inPropertyID,
UInt32 *ioDataSize,
void *outPropertyData
);
```

Discussion
The file and property being queried should be specified in *inAudioFile* and *inPropertyID*, respectively, with the size retrieved with `AudioFileGetPropertyInfo()` passed into *ioDataSize*, and the resulting data being placed in *outPropertyData*.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

AudioFileSetProperty

Sets the data for the respective property.

```
extern OSStatus AudioFileSetProperty(
AudioFileID inAudioFile,
AudioFilePropertyID inPropertyID,
UInt32 inDataSize,
const void *inPropertyData
);
```

Discussion
The file and property being set should be specified in `inAudioFile` and `inPropertyID`, respectively, with the size of the data being written passed into `ioDataSize`, and the data being written coming from `inPropertyData`.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AudioFile.h`

Global Info Access

AudioFileGetGlobalInfoSize

Calculates the size of the data that will be returned for the property.

```
extern OSStatus AudioFileGetGlobalInfoSize(
AudioFilePropertyID inPropertyID,
UInt32 inSpecifierSize,
void *inSpecifier,
UInt32 *outDataSize
);
```

Discussion
Gets the size of the `inPropertyID` for the `inSpecifier` and places it in `outDataSize`.

Availability
Available in Mac OS X v10.3 and later.

Declared In
`AudioFile.h`

AudioFileGetGlobalInfo

Retrieves the data for the queried property and specifier.

```
extern OSStatus AudioFileGetGlobalInfo(
AudioFilePropertyID inPropertyID,
UInt32 inSpecifierSize,
void *inSpecifier,
UInt32 *ioDataSize,
void *outPropertyData
);
```

Discussion
This function takes `inPropertyID` and returns `outPropertyData` based on `inSpecifier`.

Availability
Available in Mac OS X v10.3 and later.

Declared In
`AudioFile.h`

Audio File Result Codes

These values are returned when errors occur.

```
kAudioFileUnspecifiedError = 'wht?'
kAudioFileUnsupportedFileTypeError = 'typ?'
kAudioFileUnsupportedDataFormatError = 'fmt?'
kAudioFileUnsupportedPropertyError = 'pty?'
kAudioFileBadPropertySizeError = '!siz'
kAudioFilePermissionsError = 'prm?'
kAudioFileNotOptimizedError = 'optm'
kAudioFileFormatNameUnavailableError = 'nme?'
kAudioFileInvalidChunkError = 'chk?'
kAudioFileDoesNotAllow64BitDataSizeError = 'off?'
kAudioFileInvalidPacketOffsetError = 'pck?'
kAudioFileInvalidFileError = 'dta?'
kAudioFileOperationNotSupportedError = 0x6F703F3F
```

AUGraph Reference

The AUGraph API allows for creating graphs of Audio Units for processing audio data.

AUGraph Types

Defined Data Types

Typedefs are used to simplify the declaration of converters and the use of properties in the context of a graph.

- `typedef SInt32 AUNode`
- `typedef struct OpaqueAUGraph *AUGraph`

Data Structures

AudioUnitNodeConnection

Used to symbolize the connection between two nodes.

```
typedef struct AudioUnitNodeConnection{
AUNode sourceNode;
UInt32 sourceOutputNumber;
AUNode destNode;
UInt32 destInputNumber;
} AudioUnitNodeConnection;
```

Availability
Available in Mac OS X v10.3 and later.

Declared In
`AUGraph.h`

AUGraph Functions

These functions are provided to access the functionality of the AUGraph API.

NewAUGraph

Creates a new `AUGraph` instance.

```
extern OSStatus NewAUGraph(AUGraph *outGraph);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUGraph.h`

DisposeAUGraph

Destroys an `AUGraph` instance.

```
extern OSStatus DisposeAUGraph(AUGraph inGraph);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUGraph.h`

AUGraphNewNode

Creates a new node inside of the specified graph.

```
extern OSStatus AUGraphNewNode(
AUGraph inGraph,
const ComponentDescription *inDescription,
UInt32 inClassDataSize,
const void *inClassData,
AUNode *outNode
);
```

Discussion
The graph to which the new node is to be added is set in *`inGraph`*, while the node to be added may be specified using either a `ComponentDescription`, obtained from the Component Manager. The value of *`inClassData`* is a `CFPropertyList` containing the serialized data of a saved state. The function returns *`outNode`* for future reference towards the newly-created node.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.5.

Declared In
`AUGraph.h`

AUGraphNewNodeSubGraph

Adds a new subgraph within the graph.

```
extern OSStatus AUGraphNewNodeSubGraph(
AUGraph inGraph,
AUNode *outNode
);
```

Discussion
The subgraph node pointed to by *`outNode`* may be populated as if it were a graph in itself. The entire graph becomes active when the subgraph node is connected to the rest of the graph, and it is deactivated when it is disconnected.

Availability
Available in Mac OS X v10.2 and later.

Declared In
AUGraph.h

AUGraphRemoveNode

Removes the specified node from the graph.

```
extern OSStatus AUGraphRemoveNode(
AUGraph inGraph,
AUNode inNode
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUGraph.h

AUGraphGetNodeCount

Returns the number of nodes in the current graph.

```
extern OSStatus AUGraphGetNodeCount(
AUGraph inGraph,
UInt32 *outNumberOfNodes
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUGraph.h

AUGraphGetIndNode

Returns a pointer to the node at the specified index.

```
extern OSStatus AUGraphGetIndNode(
AUGraph inGraph,
UInt32 inIndex,
AUNode *outNode
);
```

Discussion
The index for the node is arbitrarily assigned when the node is added to the graph.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUGraph.h

AUGraphGetNodeInfo

Returns information about a node.

```
extern OSStatus AUGraphGetNodeInfo(
AUGraph inGraph,
AUNode inNode,
ComponentDescription *outDescription,
UInt32 *outClassDataSize,
void **outClassData,
AudioUnit *outAudioUnit
);
```

Discussion
This function retrieves various pieces of information about a graph's nodes, which may be saved and used to rebuild the graph later using `AUGraphNewNode()`. The node and graph containing the node in question are passed as *`inGraph`* and *`inNode`*, respectively. Upon output, *`outDescription`* points to a `ComponentDescription`, provided by the Component Manager. Also, *`outClassData`* points towards a `CFPropertyRef`, which may be saved and used to rebuild the graph later on. The node's Audio Unit type is pointed to by `outAudioUnit`. The *`outClassDataSize`* parameter is currently not used, and will return `0`.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.5.

Declared In
`AUGraph.h`

AUGraphGetNodeInfoSubGraph

Returns a pointer towards a subgraph.

```
extern OSStatus AUGraphGetNodeInfoSubGraph(
const AUGraph inGraph,
const AUNode inNode,
AUGraph *outSubGraph
);
```

Availability
Available in Mac OS X v10.2 and later.

Declared In
`AUGraph.h`

AUGraphIsNodeSubGraph

Indicates if a node is a subgraph.

```
extern OSStatus AUGraphIsNodeSubGraph(
const AUGraph inGraph,
const AUNode inNode,
Boolean* outFlag
);
```

Availability
Available in Mac OS X v10.2 and later.

Declared In
AUGraph.h

AUGraphConnectNodeInput

Connects two graph nodes together and specifies the way inputs are ordered.

```
extern OSStatus AUGraphConnectNodeInput(
AUGraph inGraph,
AUNode inSourceNode,
UInt32 inSourceOutputNumber,
AUNode inDestNode,
UInt32 inDestInputNumber
);
```

Discussion
When connecting nodes together, the developer must specify how the output of one node maps to the input of another. To prevent fan out, all output-input connections are one-to-one, where each node may have multiple inputs and outputs (indexed starting with 0).

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUGraph.h

AUGraphDisconnectNodeInput

Disconnects the input from the graph.

```
extern OSStatus AUGraphDisconnectNodeInput(
AUGraph inGraph,
AUNode inDestNode,
UInt32 inDestInputNumber
);
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
AUGraph.h

AUGraphClearConnections

Clears all of the connections between all inputs and outputs.

```
extern OSStatus AUGraphClearConnections(AUGraph inGraph);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUGraph.h

AUGraphGetNumberOfConnections

Returns the number of connections present in the graph.

```
extern OSStatus AUGraphGetNumberOfConnections(
AUGraph inGraph,
UInt32 *outNumberOfConnections
);
```

Availability
Available in Mac OS X v10.1 and later.
Deprecated in Mac OS X v10.5.

Declared In
AUGraph.h

AUGraphCountNodeConnections

Returns the number of connections that involve the specified node.

```
extern OSStatus AUGraphCountNodeConnections(
AUGraph inGraph,
AUNode inNode,
UInt32 *outNumConnections
);
```

Availability
Available in Mac OS X v10.3 and later.
Deprecated in Mac OS X v10.5.

Declared In
AUGraph.h

AUGraphGetNodeConnections

Returns an array containing the number of connections involving the specified node.

```
extern OSStatus AUGraphGetNodeConnections(
AUGraph inGraph,
AUNode inNode,
AudioUnitNodeConnection *outConnections,
UInt32 *ioNumConnections
);
```

Discussion
This function returns an "AudioUnitNodeConnection" (page 80) array containing information about all of the pairs of connections that involve *inNode*. The size of the array will be reflected in *ioNumConnections*, while the value returned by `AUGraphCountNodeConnections()` should be passed to this parameter upon input.

Availability
Available in Mac OS X v10.3 and later.
Deprecated in Mac OS X v10.5.

Declared In
`AUGraph.h`

AUGraphGetConnectionInfo

Returns information about a particular connection.

```
extern OSStatus AUGraphGetConnectionInfo(
AUGraph inGraph,
UInt32 inConnectionIndex,
AUNode *outSourceNode,
UInt32 *outSourceOutputNumber,
AUNode *outDestNode,
UInt32 *outDestInputNumber
);
```

Discussion
Passing an index will return the information about it. The indices are arbitrarily assigned when the connections are made, and should follow the indices contained in the *outConnections* array returned by `AUGraphGetNodeConnections()`.

Availability
Available in Mac OS X v10.1 and later.
Deprecated in Mac OS X v10.5.

Declared In
`AUGraph.h`

AUGraphUpdate

Updates all changes made to the graph while it is running.

```
extern OSStatus AUGraphUpdate(
AUGraph inGraph,
Boolean *outIsUpdated
);
```

Discussion
When a graph is running, no changes actually occur to the graph until `AUGraphUpdate()` is called. All node connect and disconnect requests are queued until this function called. When the graph is not running, all connect and disconnect requests are processed immediately, and therefore, `AUGraphUpdate()` is not necessary. If the value of *`outIsUpdated`* is `NULL`, the update will block all rendering until it is finished; a non-`NULL` value will allow `AUGraphUpdate()` to return immediately. A `true` value for *`outIsUpdated`* will indicate that all changes have occurred to the graph, whereas a `false` value means that there are still changes that have not occurred.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUGraph.h`

AUGraphOpen

Instantiates every Audio Unit in the graph.

```
extern OSStatus AUGraphOpen(AUGraph inGraph);
```

Discussion
This function should be called after the initial set of nodes is added to the graph and connections have been made between them. This will instantiate the nodes, meaning that their properties will be ready for modification. Each node's sample rate may also be set after the graph is opened.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUGraph.h`

AUGraphClose

Closes the graph and deallocates its Audio Unit nodes.

```
extern OSStatus AUGraphClose(AUGraph inGraph);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUGraph.h`

AUGraphInitialize

Initializes the graph and the connected Audio Units.

```
extern OSStatus AUGraphInitialize(AUGraph inGraph);
```

Discussion
Invoking this function will activate the connections between nodes and will initialize all nodes that are part of a connection. It is important to note that if format changes occur, sample rates for output nodes must be set before this function is called.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUGraph.h`

AUGraphUninitialize

Uninitializes the graph and all of the Audio Units.

```
extern OSStatus AUGraphUninitialize(AUGraph inGraph);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUGraph.h`

AUGraphStart

Begins audio rendering through the graph.

```
extern OSStatus AUGraphStart(AUGraph inGraph);
```

Discussion
This function starts with the head node, always an output unit, and works through the graph to get to the inputs, pulls the data, and renders it through all of the Audio Units in the path leading to the head.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUGraph.h`

AUGraphStop

Stops all rendering through the graph.

```
extern OSStatus AUGraphStop(AUGraph inGraph);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUGraph.h`

AUGraphIsOpen

Returns a boolean value indicting whether or not the graph is open.

```
extern OSStatus AUGraphIsOpen(
AUGraph inGraph,
Boolean *outIsOpen
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUGraph.h

AUGraphIsInitialized

Returns a boolean value indicting whether or not the graph is initialized.

```
extern OSStatus AUGraphIsInitialized(
AUGraph inGraph,
Boolean *outIsInitialized
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUGraph.h

AUGraphIsRunning

Returns a boolean value indicting whether or not the graph is running.

```
extern OSStatus AUGraphIsRunning(
AUGraph inGraph,
Boolean *outIsRunning
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUGraph.h

AUGraphGetCPULoad

Returns the amount of load on the CPU.

```
extern OSStatus AUGraphGetCPULoad(
AUGraph inGraph,
Float32 *outCPULoad
);
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`AUGraph.h`

AUGraphSetRenderNotification

Specifies a callback for the render process.

```
extern OSStatus AUGraphSetRenderNotification(
AUGraph inGraph,
AudioUnitRenderCallback inCallback,
void *inRefCon
);
```

Discussion
This function is intended for use when the graph has Audio Units of type '`aunt`'. The callback is specified in *`inCallback`*, and is called before and after an audio render occurs. Passing `NULL` to `inCallback` removes all callbacks from the notification. Multiple notifications are allowed.

Availability
Available in Mac OS X v10.1 and later.
Deprecated in Mac OS X v10.3.
Not available to 64-bit applications.

Declared In
`AUGraph.h`

AUGraphRemoveRenderNotification

Removes the specified callback from the notification.

```
extern OSStatus AUGraphRemoveRenderNotification(
AUGraph inGraph,
AudioUnitRenderCallback inCallback,
void *inRefCon
);
```

Discussion
This function is intended for use when the graph has Audio Units of type '`aunt`'.

Availability
Available in Mac OS X v10.2 and later.
Deprecated in Mac OS X v10.3.
Not available to 64-bit applications.

Declared In
`AUGraph.h`

AUGraphAddRenderNotify

Specifies a callback for the render process.

```
extern OSStatus AUGraphAddRenderNotify(
AUGraph inGraph,
AURenderCallback inCallback,
void *inRefCon
);
```

Discussion
This function is intended for use when the graph has Audio Units of type 'auXX', where XX is one of the various version 2 Audio Unit types, as specified in AudioUnit/AUComponent.h. The callback is specified in *inCallback*, and is called before and after an audio render occurs. Passing NULL to *inCallback* removes all callbacks from the notification. Multiple notifications are allowed.

Availability
Available in Mac OS X v10.2 and later.

Declared In
AUGraph.h

AUGraphRemoveRenderNotify

Removes the specified callback from the notification.

```
extern OSStatus AUGraphRemoveRenderNotify(
AUGraph inGraph,
AURenderCallback inCallback,
void *inRefCon
);
```

Discussion
This function is intended for use when the graph has Audio Units of type 'auXX', where XX is one of the various version 2 Audio Unit types, as specified in AudioUnit/AUComponent.h.

Availability
Available in Mac OS X v10.2 and later.

Declared In
AUGraph.h

AUGraph Result Codes

These values are returned when errors occur.

```
kAUGraphErr_NodeNotFound = -10860
kAUGraphErr_InvalidConnection = -10861
kAUGraphErr_OutputNodeErr = -10862
kAUGraphErr_CannotDoInCurrentContext = -10863
kAUGraphErr_InvalidAudioUnit = -10864
```

Music Player and Music Sequence Reference

The Music Player and Music Sequence components allow for the sequencing of MIDI endpoints and audio units.

Music Player and Music Sequence Types

Defined Data Types

These typedefs are provided to support the different structures and functions in Music Player.

- `typedef UInt32 MusicSequenceLoadFlags`
- `typedef UInt32 MusicEventType`
- `typedef Float64 MusicTimeStamp`
- `typedef struct OpaqueMusicPlayer *MusicPlayer`
- `typedef struct OpaqueMusicSequence *MusicSequence`
- `typedef struct OpaqueMusicTrack *MusicTrack`
- `typedef struct OpaqueMusicEventIterator *MusicEventIterator`

Data Structures

MIDINoteMessage

Stores information about a MIDI note event.

```
typedef struct MIDINoteMessage {
UInt8 channel;
UInt8 note;
UInt8 velocity;
UInt8 reserved;
Float32 duration;
} MIDINoteMessage;
```

Fields

`channel`
The channel number to which the note is assigned.

`note`
The value of the note to be played.

`velocity`
The volume at which the note is to be played.

`reserved`

`duration`
The length of time that the note should be played, in beats.

Discussion
This structure encapsulates the information needed to relay the properties of a note. An instance of this structure is used by the `MusicTrackNewMIDINoteEvent` (page 112) function. The values of the structure are:

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MIDIChannelMessage

Stores the data for a MIDI channel event.

```
typedef struct MIDIChannelMessage {
UInt8 status;
UInt8 data1;
UInt8 data2;
UInt8 reserved;
} MIDIChannelMessage;
```

Fields

`status`
The message and the channel it is to be relayed to.

`data1`
Data specific to the message.

`data2`
Data specific to the message.

`reserved`
???

Discussion
This structure encapsulates the information needed for a channel event to be used in a music track. An instance of this structure is used by the `MusicTrackNewMIDIChannelEvent` (page 113) function.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MIDIRawData

Stores the information for any MIDI event.

```
typedef struct MIDIRawData {
UInt32 length;
UInt8 data[1];
} MIDIRawData;
```

Fields

`length`

The size of the space allocated for `data`.

`data`

The raw MIDI data to be stored; allocate as much space as needed for the data.

Discussion

This structure encapsulates the data for an event where raw MIDI data is sent to an endpoint. An instance of this structure is used by the `MusicTrackNewMIDIRawDataEvent` (page 113) function. The values of the structure are:

Availability

Available in Mac OS X v10.0 and later.

Declared In

`MusicPlayer.h`

MIDIMetaEvent

Stores the data for a MIDI meta event.

```
typedef struct MIDIMetaEvent {
UInt8 metaEventType;
UInt8 unused1;
UInt8 unused2;
UInt8 unused3;
UInt32 dataLength;
UInt8 data[1];
} MIDIMetaEvent;
```

Fields

`metaEventType`

Specifies the type of meta event this structure encapsulates.

`unused1`

An unused value.

`unused2`

An unused value.

`unused3`

An unused value.

`dataLength`

The size of the space allocated for `data`.

`data`

The meta data for this event.

Discussion

This structure encapsulates the information needed to pass MIDI meta data, as found in standard MIDI files, to MIDI endpoints. An instance of this structure is used by the `MusicTrackNewMetaEvent` (page 114) function. The values of the structure are:

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicEventUserData

Stores data for a user event .

```
typedef struct MusicEventUserData {
UInt32 length;
UInt8 data[1];
} MusicEventUserData;
```

Fields

`length`
The size, in bytes, of the value stored in `data`.

`data`
The data stored for this event.

Discussion
This structure encapsulates the information used on a user event. An instance of this structure is used by the `MusicTrackNewUserEvent` (page 115) function.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`MusicPlayer.h`

ExtendedNoteOnEvent

Stores the data for a playback note.

```
typedef struct ExtendedNoteOnEvent {
MusicDeviceInstrumentID instrumentID;
MusicDeviceGroupID groupID;
Float32 duration;
MusicDeviceNoteParams extendedParams;
} ExtendedNoteOnEvent;
```

Fields

`instrumentID`
The instrument to be used by the Music Device.

`groupID`
The channel of the Music Device.

`duration`
The length of the note.

`extendedParams`
Any additional parameters that need to be sent to the Music Device.

Discussion
This structure encapsulates the information needed to playback a note using a Music Device. An instance of this structure is used by the `MusicTrackNewExtendedNoteEvent` (page 114) function.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

ExtendedControlEvent

Stores information regarding an event using a Music Device.

```
typedef struct ExtendedControlEvent {
MusicDeviceGroupID groupID;
AudioUnitParameterID controlID;
Float32 value;
} ExtendedControlEvent;
```

Fields

`groupID`
The channel of the Music Device to be controlled.

`controlID`
The Music Device parameter to be controlled.

`value`
The value to which the parameter should be set.

Discussion
This structure encapsulates the information needed to control a Music Device. An instance of this structure is used by the `MusicTrackNewExtendedControlEvent` (page 114) function.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

ParameterEvent

Stores information for an even using Audio Units.

```
typedef struct ParameterEvent {
AudioUnitParameterID parameterID;
AudioUnitScope scope;
AudioUnitElement element;
Float32 value;
} ParameterEvent;
```

Fields

`parameterID`
The parameter to be adjusted.

`scope`
The scope for this event.

`element`
The element to be controlled

`value`
The value to be passed into the parameter.

Discussion
This structure encapsulates the information needed to relay the properties of a note. An instance of this structure is used by the `MusicTrackNewParameterEvent` (page 115) function.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`MusicPlayer.h`

ExtendedTempoEvent

Specifies the tempo to be applied when this event occurs.

```
typedef struct ExtendedTempoEvent {
Float64 bpm;
} ExtendedTempoEvent;
```

Fields
`pbm`
The beats-per-minute to be used for the sequence from this event forward.

Discussion
This structure encapsulates the information needed to change the tempo of the sequence at a certain point. An instance of this structure is used by the `MusicTrackExtendedTempoEvent` (page 115) function.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

Music Player and Music Sequence Constants

Constants are provided for the developer's convenience. They provide a consistent set of values for various aspects of an audio file.

Music Events

These constants are used by the `MusicEventIteratorGetEventInfo` (page 119) and `MusicEventIteratorSetEventInfo` (page 119) functions to determine the type of event currently being pointed to by the iterator. Based on these values, the `outEventData` and `inEventData` pointers in these functions should point to instances of these types:

Table 4-1 Music Event Constants

Constant	Data Type
kMusicEventType_ExtendedNote	`ExtendedNoteOnEvent` (page 95)
kMusicEventType_ExtendedControl	`ExtendedControlEvent` (page 96)
kMusicEventType_ExtendedTempo	`ExtendedTempoEvent` (page 97)
kMusicEventType_User	`MusicEventUserData` (page 95)
kMusicEventType_Meta	`MIDIMetaEvent` (page 94)
kMusicEventType_MIDINoteMessage	`MIDINoteMessage` (page 92)
kMusicEventType_MIDIChannelMessage	`MIDIChannelMessage` (page 93)
kMusicEventType_MIDIRawData	`MIDIRawData` (page 93)
kMusicEventType_Parameter	`ParameterEvent` (page 96)

In addition to these types, two other values may be returned:

- `kMusicEventType_NULL` - Returned when a Music Track is empty.
- `kMusicEventType_Last` - Returned when all of the events in a track have been iterated through.

Other Constants

These additional constants have been provided as flags or definitions of values for consistency.

- `kMusicSequenceLoadSMF_ChannelsToTracks` - Used by the `MusicSequenceLoadSMFWithFlags` (page 107) and `MusicSequenceLoadSMFDataWithFlags` (page 107) to indicate that when a standard MIDI file is read in with this flag assigned, the resulting sequence will have a track created for each channel in the MIDI file, as well as another track for all of the meta information.
- `kMusicTimeStamp_EndOfTrack` - Used to provide an upper limit for the length of tracks; currently set to 1,000,000,000.0 beats.

Music Track Properties

These properties are used by `MusicTrackSetProperty` (page 112) and `MusicTrackGetProperty` (page 112) to define the status and certain values unique to the track.

`kSequenceTrackProperty_LoopInfo = 0`

Uses a structure containing a `MusicTimeStamp` indicating the length of the track and a `long` indicating the number of times the track is to be looped. If the track is to be loop infinitely, set this value equal to zero.

`kSequenceTrackProperty_OffsetTime = 1`
: Uses a `MusicTimeStamp` to determine after how many beats into the sequence the track should begin playing.

`kSequenceTrackProperty_MuteStatus = 2`
: Uses a `Boolean` to mute the track.

`kSequenceTrackProperty_SoloStatus = 3`
: Uses a `Boolean` to determine if the track is the only one to control an endpoint or node.

`kSequenceTrackProperty_AutomatedParameters = 4`
: Uses a `UInt32` to signify if the track modifys a node parameters.

`kSequenceTrackProperty_TrackLength = 5`
: Uses a `MusicTimeStamp` to reflect the length of the track, in beats.

Music Player, Music Sequence, and Music Track Functions

These functions make up the functionality of the Music Player API.

Music Player Functions

The functions work with a Music Player instances, which are used to play back a Music Sequence instance. Each Music Player is only allowed to be associated with one Music Sequence, and vice versa.

NewMusicPlayer

Creates a new Music Player instance.

```
extern OSStatus NewMusicPlayer(MusicPlayer *outPlayer);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

DisposeMusicPlayer

Disposes of a Music Player.

```
extern OSStatus DisposeMusicPlayer(MusicPlayer inPlayer);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicPlayerSetSequence

Associates a Music Player with a Music Sequence.

```
extern OSStatus MusicPlayerSetSequence(
MusicPlayer inPlayer,
MusicSequence inSequence
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicPlayerSetTime

Moves the Music Player's playhead to the desired time, in beats.

```
extern OSStatus MusicPlayerSetTime(
MusicPlayer inPlayer,
MusicTimeStamp inTime
);
```

Discussion
In addition to moving the playhead, `MusicPlayerSetTime()` also prerolls the track up to the playhead, setting Audio Unit parameters and MIDI endpoints to where they should be at the playhead, based on the sequence prior to the playhead.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicPlayerGetTime

Returns the current placement of the playhead, in beats.

```
extern OSStatus MusicPlayerGetTime(
MusicPlayer inPlayer,
MusicTimeStamp *outTime
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicPlayerGetHostTimeForBeats

Returns the number of seconds equivalent to the number beats provided.

```
extern OSStatus MusicPlayerGetHostTimeForBeats(
MusicPlayer inPlayer,
MusicTimeStamp inBeats,
UInt64* outHostTime
);
```

Discussion
This function determines what value to return by analyzing the tempo track in the sequence and determining the amount of time has passed when `inBeats` number of beats have occurred. Only valid when called while a Music Player is playing.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`MusicPlayer.h`

MusicPlayerGetBeatsForHostTime

Returns the beat for a given time.

```
extern OSStatus MusicPlayerGetBeatsForHostTime(
MusicPlayer inPlayer,
UInt64 inHostTime,
MusicTimeStamp *outBeats);
```

Discussion
This function determines what value to return by analyzing the tempo track in the sequence and determining the number beats that have passed at `inHostTime`. Only valid when called while a Music Player is playing.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`MusicPlayer.h`

MusicPlayerPreroll

Prepares a sequence to be played.

```
extern OSStatus MusicPlayerPreroll(MusicPlayer inPlayer);
```

Discussion
Prerolling a player prepares the player's sequence to be played. Calling this function will synchronize all of the tracks within the sequence, bringing MIDI endpoints to their correct state with respect to the playhead, while adjusting Audio Unit parameters as well. Adding an event prior to a playhead invalidates a preroll, and so this function should only be called after all events have been added, since this operation is rather costly.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicPlayerStart

Begins playback of a sequence.

```
extern OSStatus MusicPlayerStart(MusicPlayer inPlayer);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicPlayerStop

Halts the playback of a sequence.

```
extern OSStatus MusicPlayerStop(MusicPlayer inPlayer);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicPlayerIsPlaying

Returns a `Boolean` reflecting the current state of a player.

```
extern OSStatus MusicPlayerIsPlaying(
MusicPlayer inPlayer,
Boolean* outIsPlaying
);
```

Availability
Available in Mac OS X v10.2 and later.

Declared In
`MusicPlayer.h`

MusicPlayerSetPlayRateScalar

Sets a tempo multiplier for a sequence.

```
extern OSStatus MusicPlayerSetPlayRateScalar(
MusicPlayer inPlayer,
Float64 inScaleRate
);
```

Discussion
The value of `inScaleRate` will be applied to the tempo track of the sequence, adjusting the playback tempo uniformly. For instance, if a tempo track is set up to be entirely 60 bpm, and a value of two is set as the `inScaleRate`, playback will occur at 120 bpm. The scale rate is not allowed to be negative (reverse playback is not allowed), and must be greater than zero (use `MusicPlayerStop` (page 102) to stop playback instead).

Availability
Available in Mac OS X v10.3 and later.

Declared In
`MusicPlayer.h`

MusicPlayerGetPlayRateScalar

Returns the current tempo multiplier.

```
extern OSStatus MusicPlayerGetPlayRateScalar(
MusicPlayer inPlayer,
Float64 *outScaleRate
);
```

Discussion
The play rate scalar is a multiplier applied to every tempo event in the tempo track of a sequence. At playback, every tempo in the tempo track will multiplied by this value and the sequence will them play at the resulting tempo.

Availability
Available in Mac OS X v10.3 and later.

Declared In
`MusicPlayer.h`

Music Sequence Functions

The functions work with a Music Sequence instance, and are used to create, modify, and dispose of sequences.

NewMusicSequence

Creates a new Music Sequence.

```
extern OSStatus NewMusicSequence(MusicSequence *outSequence);
```

Discussion
After creation, a Music Sequence contains one track: the Tempo Track. This track determines the playback rate, in beats-per-minute (bpm), determined by events placed along the track. Only events of type `ExtendedTempoEvent` (page 97) are allowed in the tempo track. To get a pointer to the tempo track (to add and remove tempo events), use `MusicSequenceGetTempoTrack` (page 105).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

DisposeMusicSequence

Disposes of a Music Sequence.

```
extern OSStatus DisposeMusicSequence(MusicSequence
inSequence);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicSequenceNewTrack

Creates a new event track within a sequence.

```
extern OSStatus MusicSequenceNewTrack(
MusicSequence inSequence,
MusicTrack *outTrack
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicSequenceDisposeTrack

Removes and disposes of a track from within a sequence.

```
extern OSStatus MusicSequenceDisposeTrack(
MusicSequence inSequence,
MusicTrack inTrack
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicSequenceGetTrackCount

Returns the number of tracks within a sequence.

```
extern OSStatus MusicSequenceGetTrackCount(
MusicSequence inSequence,
UInt32 *outNumberOfTracks
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicSequenceGetIndTrack

Returns a pointer to the track at an index.

```
extern OSStatus MusicSequenceGetIndTrack(
MusicSequence inSequence,
UInt32 inTrackIndex,
MusicTrack *outTrack
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicSequenceGetTrackIndex

Returns the index within the sequence for the given track.

```
extern OSStatus MusicSequenceGetTrackIndex(
MusicSequence inSequence,
MusicTrack inTrack,
UInt32 *outTrackIndex
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicSequenceGetTempoTrack

Returns a pointer to a sequence's tempo track.

```
extern OSStatus MusicSequenceGetTempoTrack(
MusicSequence inSequence,
MusicTrack *outTrack
);
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`MusicPlayer.h`

MusicSequenceSetAUGraph

Sets the sequence to work with a particular AUGraph instance.

```
extern OSStatus MusicSequenceSetAUGraph(
MusicSequence inSequence,
AUGraph inGraph
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicSequenceGetAUGraph

Returns a pointer to the AUGraph which is being used by the given sequence.

```
extern OSStatus MusicSequenceGetAUGraph(
MusicSequence inSequence,
AUGraph *outGraph
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicSequenceSetMIDIEndpoint

Sets the MIDI device being used by the given sequence.

```
extern OSStatus MusicSequenceSetMIDIEndpoint(
MusicSequence inSequence,
MIDIEndpointRef inEndpoint
);
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`MusicPlayer.h`

MusicSequenceLoadSMF

Parses a standard MIDI file and places its contents into a track within a sequence.

```
extern OSStatus MusicSequenceLoadSMF(
MusicSequence inSequence,
const FSSpec *inFileSpec
);
```

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In
`MusicPlayer.h`

MusicSequenceLoadSMFData

Parses MIDI data out of memory and uses it to populate a sequence.

```
extern OSStatus MusicSequenceLoadSMFData(
MusicSequence inSequence,
CFDataRef inData
);
```

Availability
Available in Mac OS X v10.2 and later.
Deprecated in Mac OS X v10.5.
Not available to 64-bit applications.

Declared In
`MusicPlayer.h`

MusicSequenceLoadSMFWithFlags

Parses a standard MIDI file and places its contents into a sequence.

```
extern OSStatus MusicSequenceLoadSMFWithFlags(
MusicSequence inSequence,
FSRef *inFileRef,
MusicSequenceLoadFlags inFlags
);
```

Discussion
Use `MusicSequenceLoadSMFWithFlags` to load data from a standard MIDI file into a sequence. The use of flags allows for the resulting data to be customized. Passing in 0 to `inFlags` makes this function act like `MusicSequenceLoadSMF` (page 106). The only other flag currently available for use is `kMusicSequenceLoadSMF_ChannelsToTracks`, described in "Other Constants" (page 98).

Availability
Available in Mac OS X v10.3 and later.
Deprecated in Mac OS X v10.5.

Declared In
`MusicPlayer.h`

MusicSequenceLoadSMFDataWithFlags

Parses MIDI data out of memory and uses it to populate a sequence.

```
extern OSStatus MusicSequenceLoadSMFDataWithFlags(
MusicSequence inSequence,
CFDataRef inData,
MusicSequenceLoadFlags inFlags
);
```

Discussion
As with `MusicSequenceLoadSMFWithFlags` (page 107), passing a 0 to `inFlags` will cause this function to behave like `MusicSequenceLoadSMFData` (page 107). Passing it `kMusicSequenceLoadSMF_ChannelsToTracks` will cause the data to be formatted as described in "Other Constants" (page 98).

Availability
Available in Mac OS X v10.3 and later.
Deprecated in Mac OS X v10.5.

Declared In
`MusicPlayer.h`

MusicSequenceSaveSMF

Saves the contents of a sequence to file, in standard MIDI format.

```
extern OSStatus MusicSequenceSaveSMF(
MusicSequence inSequence,
const FSSpec *inFileSpec,
UInt16 inResolution
);
```

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`MusicPlayer.h`

MusicSequenceSaveSMFData

Saves the contents of a sequence to memory, in standard MIDI format

```
extern OSStatus MusicSequenceSaveSMFData(
MusicSequence inSequence,
CFDataRef *outData,
UInt16 inResolution
);
```

Availability
Available in Mac OS X v10.2 and later.
Deprecated in Mac OS X v10.5.

Declared In
`MusicPlayer.h`

MusicSequenceReverse

Revereses all of the events in a sequence.

```
extern OSStatus MusicSequenceReverse(MusicSequence
inSequence);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicSequenceGetSecondsForBeats

Returns the number of seconds elapsed for the given number of beats in the sequence.

```
extern OSStatus MusicSequenceGetSecondsForBeats(
MusicSequence inSequence,
MusicTimeStamp inBeats,
Float64* outSeconds
);
```

Discussion
This function uses the tempo track of the sequence to determine how much time passes until the provided number of beats has occurred.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`MusicPlayer.h`

MusicSequenceGetBeatsForSeconds

Returns the number of beats that have elapsed after the provided number of seconds.

```
extern OSStatus MusicSequenceGetBeatsForSeconds(
MusicSequence inSequence,
Float64 inSeconds,
MusicTimeStamp* outBeats
);
```

Discussion
This function uses the sequence's tempo track to determine the number of beats that occur after the provided number of seconds.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`MusicPlayer.h`

MusicSequenceSetUserCallback

Set the callback that is used whenever an event of type `MusicEventUserData` (page 95) occurs.

```
extern OSStatus MusicSequenceSetUserCallback(
MusicSequence inSequence,
MusicSequenceUserCallback inCallback,
void* inClientData
);
```

Discussion
The callback registered suing this function is of type `MusicSequenceUserCallback` (page 121). The pointer passed in via `inClientData` is passed on to the callback when it is called. Passing `NULL` to `inCallback` removes any callback previously registered.

Note that if `MusicPlayerSetTime` (page 100) is called, this callback will be called for any events between the previous playhead and the new playhead. See `MusicSequenceUserCallback` (page 121) for more information.

Availability
Available in Mac OS X v10.3 and later.

Declared In
`MusicPlayer.h`

Music Track Setup Functions

These functions are provided to create and set up a music track.

MusicTrackGetSequence

Returns the sequence to which the track belongs.

```
extern OSStatus MusicTrackGetSequence(
MusicTrack inTrack,
MusicSequence *outSequence
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackSetDestNode

Sets the AUGraph node which the track controls.

```
extern OSStatus MusicTrackSetDestNode(
MusicTrack inTrack,
AUNode inNode
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackSetDestMIDIEndpoint

Sets the track's destination endpoint.

```
extern OSStatus MusicTrackSetDestMIDIEndpoint(
MusicTrack inTrack,
MIDIEndpointRef inEndpoint
);
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`MusicPlayer.h`

MusicTrackGetDestNode

Returns a pointer towards the node that the track points to.

```
extern OSStatus MusicTrackGetDestNode(
MusicTrack inTrack,
AUNode *outNode
);
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`MusicPlayer.h`

MusicTrackGetDestMIDIEndpoint

Returns a pointer towards the endpoint that the track points to.

```
extern OSStatus MusicTrackGetDestMIDIEndpoint(
MusicTrack inTrack,
MIDIEndpointRef *outEndpoint
);
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
MusicPlayer.h

Music Track Property Functions

The functions with a Music Track's properties.

MusicTrackSetProperty

Sets the track's value for the given property.

```
extern OSStatus MusicTrackSetProperty(
MusicTrack inTrack,
UInt32 inPropertyID,
void *inData,
UInt32 inLength
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
MusicPlayer.h

MusicTrackGetProperty

Returns the track's value for a given property.

```
extern OSStatus MusicTrackGetProperty(
MusicTrack inTrack,
UInt32 inPropertyID,
void *outData,
UInt32 *ioLength
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
MusicPlayer.h

Music Track Event Setup Functions

These function place events on the various tracks within a sequence.

MusicTrackNewMIDINoteEvent

Creates a new MIDI note event.

```
extern OSStatus MusicTrackNewMIDINoteEvent(
MusicTrack inTrack,
MusicTimeStamp inTimeStamp,
const MIDINoteMessage *inMessage
);
```

Discussion
This places a `MIDINoteMessage` (page 92) instance on `inTrack`, which is to be played at `inTimeStamp`.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackNewMIDIChannelEvent

Creates a new MIDI channel event.

```
extern OSStatus MusicTrackNewMIDIChannelEvent(
MusicTrack inTrack,
MusicTimeStamp inTimeStamp,
const MIDIChannelMessage *inMessage
);
```

Discussion
This places a `MIDIChannelMessage` (page 93) instance on `inTrack`, which is to be played at `inTimeStamp`.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackNewMIDIRawDataEvent

Creates a new MIDI raw data event.

```
extern OSStatus MusicTrackNewMIDIRawDataEvent(
MusicTrack inTrack,
MusicTimeStamp inTimeStamp,
const MIDIRawData *inRawData
);
```

Discussion
This places a `MIDIRawData` (page 93) instance on `inTrack`, which is to be played at `inTimeStamp`.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackNewMetaEvent

Creates a new MIDI meta event.

```
extern OSStatus MusicTrackNewMetaEvent(
MusicTrack inTrack, M
usicTimeStamp inTimeStamp,
const MIDIMetaEvent *inMetaEvent
);
```

Discussion
This places a `MIDIMetaEvent` (page 94) instance on `inTrack`, which is to be played at `inTimeStamp`.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackNewExtendedNoteEvent

Creates a new extended note event.

```
extern OSStatus MusicTrackNewExtendedNoteEvent(
MusicTrack inTrack,
MusicTimeStamp inTimeStamp,
const ExtendedNoteOnEvent *inInfo);
```

Discussion
This places a `ExtendedNoteOnEvent` (page 95) instance on `inTrack`, which is to be played at `inTimeStamp`.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackNewExtendedControlEvent

Creates a new extended control event.

```
extern OSStatus MusicTrackNewExtendedControlEvent(
MusicTrack inTrack,
MusicTimeStamp inTimeStamp,
const ExtendedControlEvent *inInfo);
```

Discussion
This places a `ExtendedControlEvent` (page 96) instance on `inTrack`, which is to be played at `inTimeStamp`.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackNewParameterEvent

Creates a new parameter event.

```
extern OSStatus MusicTrackNewParameterEvent(
MusicTrack inTrack,
MusicTimeStamp inTimeStamp,
const ParameterEvent *inInfo
);
```

Discussion
This places a `ParameterEvent` (page 96) instance on `inTrack`, which is to be played at `inTimeStamp`.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`MusicPlayer.h`

MusicTrackExtendedTempoEvent

Creates a new extended tempo event.

```
extern OSStatus MusicTrackNewExtendedTempoEvent(
MusicTrack inTrack,
MusicTimeStamp inTimeStamp,
Float64 inBPM
);
```

Discussion
This places a `ExtendedTempoEvent` (page 97) instance on `inTrack`, which is to be played at `inTimeStamp`.

MusicTrackNewUserEvent

Creates a new user event.

```
extern OSStatus MusicTrackNewUserEvent(
MusicTrack inTrack,
MusicTimeStamp inTimeStamp,
const MusicEventUserData* inUserData
);
```

Discussion
This places a `MusicEventUserData` (page 95) on `inTrack`, which is to be played at `inTimeStamp`.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

Music Track Event Editing

The functions allow you to arrange events within a track, create new tracks from events, and edit groups of events.

MusicTrackMoveEvents

Moves the events from the given range to a new place.

```
extern OSStatus MusicTrackMoveEvents(
MusicTrack inTrack,
MusicTimeStamp inStartTime,
MusicTimeStamp inEndTime,
MusicTimeStamp inMoveTime
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

NewMusicTrackFrom

Creates a new music track from the event in the given range.

```
extern OSStatus NewMusicTrackFrom(
MusicTrack inSourceTrack,
MusicTimeStamp inSourceStartTime,
MusicTimeStamp inSourceEndTime,
MusicTrack *outNewTrack
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackClear

Removes the events in the given range.

```
extern OSStatus MusicTrackClear(
MusicTrack inTrack,
MusicTimeStamp inStartTime,
MusicTimeStamp inEndTime
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackCut

Removes the events in the given range, and moves those behind the range up.

```
extern OSStatus MusicTrackCut(
MusicTrack inTrack,
MusicTimeStamp inStartTime,
MusicTimeStamp inEndTime
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackCopyInsert

Inserts the selected range at the destination insertion time, move all of events behind the insertion time back after the range.

```
extern OSStatus MusicTrackCopyInsert(
MusicTrack inSourceTrack,
MusicTimeStamp inSourceStartTime,
MusicTimeStamp inSourceEndTime,
MusicTrack inDestTrack,
MusicTimeStamp inDestInsertTime
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicTrackMerge

Merges the selected range of events with the existing events in the track.

```
extern OSStatus MusicTrackMerge(
MusicTrack inSourceTrack,
MusicTimeStamp inSourceStartTime,
MusicTimeStamp inSourceEndTime,
MusicTrack inDestTrack,
MusicTimeStamp inDestInsertTime
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

Music Track Event Iteration

The functions work with a Music Track instances.

NewMusicEventIterator

Creates a new iterator for a track.

```
extern OSStatus NewMusicEventIterator(
MusicTrack inTrack,
MusicEventIterator *outIterator
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

DisposeMusicEventIterator

Destroys an iterator.

```
extern OSStatus DisposeMusicEventIterator(MusicEventIterator inIterator);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicEventIteratorSeek

Moves the iterator to the event closest to the supplied time stamp.

```
extern OSStatus MusicEventIteratorSeek(
MusicEventIterator inIterator,
MusicTimeStamp inTimeStamp
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicEventIteratorNextEvent

Moves the iterator o the next event in the track.

```
extern OSStatus MusicEventIteratorNextEvent(
MusicEventIterator inIterator
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
MusicPlayer.h

MusicEventIteratorPreviousEvent

Move the iterator to the previous event in the track.

```
extern OSStatus MusicEventIteratorPreviousEvent(
MusicEventIterator inIterator
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
MusicPlayer.h

MusicEventIteratorGetEventInfo

Returns information about the event currently iterated upon.

```
extern OSStatus MusicEventIteratorGetEventInfo(
MusicEventIterator inIterator,
MusicTimeStamp *outTimeStamp,
MusicEventType *outEventType,
const void* *outEventData,
UInt32 *outEventDataSize
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
MusicPlayer.h

MusicEventIteratorSetEventInfo

Sets the type and data for an event.

```
extern OSStatus MusicEventIteratorSetEventInfo(
MusicEventIterator inIterator,
MusicEventType inEventType,
const void *inEventData
);
```

Availability
Available in Mac OS X v10.2 and later.

Declared In
MusicPlayer.h

MusicEventIteratorDeleteEvent

Removes the event from the track.

```
extern OSStatus MusicEventIteratorDeleteEvent(
MusicEventIterator inIterator
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicEventIteratorSetEventTime

Sets the time that an event should occur at.

```
extern OSStatus MusicEventIteratorSetEventTime(
MusicEventIterator inIterator,
MusicTimeStamp inTimeStamp
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicEventIteratorHasPreviousEvent

Returns a boolean signifying if there is an event before the currently iterated event.

```
extern OSStatus MusicEventIteratorHasPreviousEvent(
MusicEventIterator inIterator,
Boolean *outHasPreviousEvent
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicEventIteratorHasNextEvent

Returns a boolean signifying if there is an event after the currently iterated event.

```
extern OSStatus MusicEventIteratorHasNextEvent(
MusicEventIterator inIterator,
Boolean *outHasNextEvent
);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`MusicPlayer.h`

MusicEventIteratorHasCurrentEvent

Returns a boolean which tells if the iterator points towards an event.

```
extern OSStatus MusicEventIteratorHasCurrentEvent(
MusicEventIterator inIterator,
Boolean *outHasCurrentEvent
);
```

Availability
Available in Mac OS X v10.2 and later.

Declared In
`MusicPlayer.h`

Music Sequence Callbacks

MusicSequenceUserCallback

The callback used whenever a user event occurs in an event track.

```
typedef CALLBACK_API_C(void,MusicSequenceUserCallback)(
void *inClientData,
MusicSequence inSequence,
MusicTrack inTrack,
MusicTimeStamp inEventTime,
const MusicEventUserData *inEventData,
MusicTimeStamp inStartSliceBeat,
MusicTimeStamp inEndSliceBeat
);
```

Music Player and Music Sequence Result Codes

These values are returned when errors occur.

```
kAudioToolboxErr_TrackIndexError = -10859
kAudioToolboxErr_TrackNotFound = -10858
kAudioToolboxErr_EndOfTrack = -10857
kAudioToolboxErr_StartOfTrack = -10856
kAudioToolboxErr_IllegalTrackDestination = -10855
kAudioToolboxErr_NoSequence = -10854
kAudioToolboxErr_InvalidEventType = -10853
```

```
kAudioToolboxErr_InvalidPlayerState = -10852
```

Audio Units

This chapter describes the functions, structures, and constants used throughout the Audio Unit framework. The section "Reference" (page 126) describes the constants, data types, and functions that are relevant to audio unit development.

Overview

In Core Audio, audio units serve a number of purposes. Audio units are used to generate, process, receive, or otherwise manipulate streams of audio. They are building blocks that may be used singly or connected together to form an audio signal graph, or `AUGraph` (page 46).

The Audio Unit Framework

The audio unit framework provides a set of services that developers can take advantage of in their own applications by using audio units. The framework also provides services for those who want to develop their own audio units.

Audio units are defined as processing units. Their input can come from a variety of sources (for example, encoded data, other audio units, or none); their output is generally a buffer of audio data.

Apple ships a set of AudioUnit components, as well as defining the interface for the AudioUnit component.

In Java, these services are available in the `com.apple.audio.units` package.

In the Macintosh system architecture, audio units are simply components, and like all components are identified based on their four-character code type, subType and ID field. The Component Manager provides a set of APIs for querying the available components on the system. You can use the `FindNextComponent()` call to find out what audio units are installed on the system. Instances are created by means of the `OpenAComponent()` call and released by the `CloseComponent()` call.

For more information on the Component Manager, consult More Macintosh Toolbox.

The Audio Unit API

The audio unit API presents the basic audio unit interface, as well as the constants that define the audio unit type, and the generic sub-types of an audio unit.

The specific ID of an audio unit represents the specific functionality of the audio unit itself. For example, the DLSMusicDevice is an audio unit that is able to use both Downloadable Sounds (DLS) and SoundFont 2 (SF2) files as sample data for sample-based synthesis. Its type is '`aumu`'–– an audio unit music device. Its sub-type is '`dls `' –– a DLS music device (note the space at the end).

For more specific information about components, refer to Inside Macintosh: More Macintosh Toolbox, Chapter 6, Component Manager.

Audio Unit State

The basic audio unit states are closed, open, and initialized, which correspond to these calls:

- `OpenAComponent()`
- `CloseAComponent()`
- `AudioUnitInitialize()`
- `AudioUnitUninitalize()`

No significant resource allocations are expected to occur when the audio unit component is first opened with `OpenAComponent()`. `AudioUnitInitialize()` is called after optional configuration has occured. This is where the audio unit allocates and is prepared to render.

`AudioUnitReset()` may be called on any initialized audio unit. The `AudioUnitReset()` call clears any buffers, resets filter memory, and stops any playing notes (for example, in a MusicDevice software synthesizer). It places the audio unit back to its initialized state.

Audio Unit Sources and Destinations

Audio units have sources and destinations. An audio unit can be just a source unit, such as software synthesizers, which are presented as a type of audio unit defined as a MusicDevice.

An audio unit can also be just a destination that is attached to a hardware output device.

Some audio units contain both input and output audio data. DSP processors, such as reverbs, filters, and mixers are examples, as are format converters, such as 16-bit integer to floating-point converters, interleavers-deinterleavers, and sample rate converters.

Audio Unit Properties

Properties represent a general and extensible mechanism for passing information to and from audio units. Information is communicated via a `void*` data parameter and a data byte-size parameter. The type of information is identified by an `AudioUnitPropertyID`.

Information is addressed to a particular section of an audio unit with `AudioUnitScope` and `AudioUnitElement`. `AudioUnitScope` includes the following constants:

- `kAudioUnitScope_Global`
- `kAudioUnitScope_Input`
- `kAudioUnitScope_Output`
- `kAudioUnitScope_Group`

`AudioUnitElement` is a zero-based index of a particular input, output, or group and is typically ignored for global scope. `AudioUnitPropertyID`s are defined in `Constants` (page 126).

Audio Unit Parameters

Parameters are values that can change over time, and are generally time-sensitive and can be scheduled. Parameters could include such things as volume or panning of a particular output on the mixer audio unit, for instance.

I/O Management

Audio unit I/O Management relies on a "pull" I/O model, which specifies through its properties the number and format of its inputs and outputs. Each input/output is a set of audio buffers corresponding to audio channels.

Data can be supplied to an audio unit through one of two mechanisms:

1. Connecting an audio unit output to another audio unit that will provide input using `kAudioUnitProperty_MakeConnection`. Audio data is automatically routed to the input with no required user intervention.

2. Registering a client callback using `kAudioUnitProperty_SetInputCallback` where the client can provide audio source data to an audio unit through the supplied callback.

The AUGraph API provides a higher-level connection service, freeing the client from calling the audio unit directly.

The "Pull" I/O Model

As mentioned, audio units use a "pull" I/O model, with each unit specifying through its properties the number and format of its inputs and ouputs. Each output is in itself a set of audio buffers corresponding to audio channels. Connections between units are also managed via properties. Data is requested from an audio unit through its `AudioUnitRender` (page 145) function being called by one of its destinations, or the `AURenderCallback` (page 152) being called.

Key points about `AudioUnitRender()` arguments:

1. The `AudioTimeStamp` specifies the start time of the buffer to be rendered, synchronizing the hosttime of the machine with the sample time of the audio to lock it with other realtime events such as MIDI.

2. The `AudioBufferList` argument passes in and receives back a set of buffers of audio data. The client may pass in a list or let the AudioUnit provide it.

A client can request notifications of the rendering activity of an audio unit by installing a callback using `kAudioUnitProperty_RenderNotification`. The client's callback will then be called by the audio unit, both before and after any call to the unit's render slice function.

The `ioActionFlags` parameter provides the unit with instructions on how to handle the buffer supplied:

`kAudioUnitRenderAction_Accumulate` –– The unit should sum its output into the given buffer, rather than replace it. This action is only valid for formats that support easy stream mixing like linear PCM. In addition, a buffer will always be supplied.

`kAudioUnitRenderAction_UseProvidedBuffer` –– This flag indicates that the rendered audio must be placed in the buffer pointed the `ioData` argument. In this case, `ioData` must point to a valid piece of allocated memory. If this flag is not set, the mData member of `ioData` may possibly be changed upon return, pointing to a different buffer (owned by the audio unit).

If the `ioData` member is `NULL`, then rendering may set `ioData` to a buffer list owned by the audio unit. In any case, on return, `ioData` points to the rendered audio data.

The `inTimeStamp` parameter gives the audio unit information about what the time is for the start of the rendered audio output.

The `inOutputBusNumber` parameter requests that audio be rendered for a particular audio output of the audio unit. Rendering is performed separately for each of its outputs. The audio unit is expected to cache its rendered audio for each output in the case that it is called more than once for the same output (`inOutputBusNumber` is the same) at the same time (`inTimeStamp` is the same). This solves the "fanout" problem.

Additional Information

Additional information and documentation is available with the Core Audio SDK, available from the Audio Developer webpage:

http://developer.apple.com/audio

Reference

This reference section describes the structures, constants, parameters, properties, and typedefs provided in the Audio Unit framework. Many of these are universal among all audio units, while some are specific to Apple-provided units.

Constants

Component Types and Subtypes

The following audio units are provided by Apple. The four-character codes identify each of these units with the Component Manager, allowing for you to find and use them.

`kAudioUnitType_Output = FOUR_CHAR_CODE('auou')`
: The component type for all output units.

`kAudioUnitSubType_HALOutput = FOUR_CHAR_CODE('ahal')`
: The component subtype for an output unit that uses a HAL output.

`kAudioUnitSubType_DefaultOutput = FOUR_CHAR_CODE('def ')`
The component subtype for an output unit that uses the default output, as selected by the user.

`kAudioUnitSubType_SystemOutput = FOUR_CHAR_CODE('sys ')`
The component subtype for an output unit that uses the system output.

`kAudioUnitSubType_GenericOutput = FOUR_CHAR_CODE('genr')`
The component subtype for a generic output unit.

`kAudioUnitType_MusicDevice = FOUR_CHAR_CODE('aumu')`
The component type for a music device unit.

`kAudioUnitSubType_DLSSynth = FOUR_CHAR_CODE('dls ')`
The component subtype for the DLS synth music device unit

`kAudioUnitType_MusicEffect = FOUR_CHAR_CODE('aumf')`
The component type for a music effect unit.

`kAudioUnitType_FormatConverter = FOUR_CHAR_CODE('aufc')`
The component type for a format converter unit.

`kAudioUnitSubType_AUConverter = FOUR_CHAR_CODE('conv')`
The component subtype for an AUConverter format converter unit.

`kAudioUnitSubType_Varispeed = FOUR_CHAR_CODE('vari')`
The component subtype for a Varispeed effect unit.

`kAudioUnitSubType_Delay = FOUR_CHAR_CODE('dely')`
The component subtype for a delay effect unit.

`kAudioUnitSubType_LowPassFilter = FOUR_CHAR_CODE('lpas')`
The component subtype for a low-pass filter effect unit.

`kAudioUnitSubType_HighPassFilter = FOUR_CHAR_CODE('hpas')`
The component subtype for a high-pass filter effect unit.

`kAudioUnitSubType_BandPassFilter = FOUR_CHAR_CODE('bpas')`
The component subtype for a band-pass filter effect unit.

`kAudioUnitSubType_HighShelfFilter = FOUR_CHAR_CODE('hshf')`
The component subtype for a high-shelf filter effect unit.

`kAudioUnitSubType_LowShelfFilter = FOUR_CHAR_CODE('lshf')`
The component subtype for a low-shelf filter effect unit.

`kAudioUnitSubType_ParametricEQ = FOUR_CHAR_CODE('pmeq')`
The component subtype for a parametric equalizer effect unit.

`kAudioUnitSubType_GraphicEQ = FOUR_CHAR_CODE('greq')`
The component subtype for a graphic equalizer effect unit.

`kAudioUnitSubType_PeakLimiter = FOUR_CHAR_CODE('lmtr')`
The component subtype for a peak limiter effect unit.

`kAudioUnitSubType_DynamicsProcessor = FOUR_CHAR_CODE('dcmp')`
The component subtype for a dynamics processor effect unit.

`kAudioUnitSubType_MultiBandCompressor = FOUR_CHAR_CODE('mcmp')`
The component subtype for multi-band compressor effect unit.

`kAudioUnitSubType_MatrixReverb = FOUR_CHAR_CODE('mrev')`
The component subtype for a matrix reverb effect unit.

`kAudioUnitType_Mixer = FOUR_CHAR_CODE('aumx')`
The component type for a mixer unit.

`kAudioUnitSubType_StereoMixer = FOUR_CHAR_CODE('smxr')`
: The component subtype for a stereo mixer unit.

`kAudioUnitSubType_3DMixer = FOUR_CHAR_CODE('3dmx')`
: The component subtype for a three-dimensional mixer unit.

`kAudioUnitSubType_MatrixMixer = FOUR_CHAR_CODE('mxmx')`
: The component subtype for a matrix mixer unit.

`kAudioUnitType_Panner = FOUR_CHAR_CODE('aupn')`
: The component type for a panner unit.

`kAudioUnitType_OfflineEffect = FOUR_CHAR_CODE('auol')`
: The component type fpr an offline effect unit.

`kAudioUnitManufacturer_Apple = FOUR_CHAR_CODE('appl')`
: The component manufacturer type for all units provided by Apple.

Render Action Flags

These flags provide you with information on the status of a render within an audio unit.

`kAudioUnitRenderAction_PreRender = (1 << 2)`
: The audio unit is prepared to render.

`kAudioUnitRenderAction_PostRender = (1 << 3)`
: The audio unit is finished with the current render.

`kAudioUnitRenderAction_OutputIsSilence = (1 << 4)`
: The current output of the render is silence.

`kAudioOfflineUnitRenderAction_Preflight = (1 << 5)`
: The audio unit has not yet rendered.

`AudioOfflineUnitRenderAction_Render = (1 << 6)`
: The audio unit is prepared to render.

`kAudioOfflineUnitRenderAction_Complete = (1 << 7)`
: The audio unit has been used to render.

Errors

These errors may arrise when rendering audio with an audio unit.

```
kAudioUnitErr_InvalidProperty           = -10879
kAudioUnitErr_InvalidParameter          = -10878
kAudioUnitErr_InvalidElement            = -10877
kAudioUnitErr_NoConnection              = -10876
kAudioUnitErr_FailedInitialization      = -10875
kAudioUnitErr_TooManyFramesToProcess    = -10874
kAudioUnitErr_IllegalInstrument         = -10873
kAudioUnitErr_InstrumentTypeNotFound    = -10872
kAudioUnitErr_InvalidFile               = -10871
kAudioUnitErr_UnknownFileType           = -10870
```

```
kAudioUnitErr_FileNotSpecified          = -10869
kAudioUnitErr_FormatNotSupported        = -10868
kAudioUnitErr_Uninitialized             = -10867
kAudioUnitErr_InvalidScope              = -10866
kAudioUnitErr_PropertyNotWritable       = -10865
kAudioUnitErr_InvalidPropertyValue      = -10851
kAudioUnitErr_PropertyNotInUse          = -10850
kAudioUnitErr_Initialized               = -10849
kAudioUnitErr_InvalidOfflineRender      = -10848
kAudioUnitErr_Unauthorized              = -10847
kAudioUnitErr_CannotDoInCurrentContext  = -10863
```

Parameter Event Types

These values specify what type of parameter event is occuring.

```
kParameterEvent_Immediate   = 1
kParameterEvent_Ramped      = 2
```

Component Call Selectors

These selectors determine the state of an audio unit.

```
kAudioUnitInitializeSelect              = 0x0001,
kAudioUnitUninitializeSelect            = 0x0002,
kAudioUnitGetPropertyInfoSelect         = 0x0003,
kAudioUnitGetPropertySelect             = 0x0004,
kAudioUnitSetPropertySelect             = 0x0005,
kAudioUnitAddPropertyListenerSelect     = 0x000A,
kAudioUnitRemovePropertyListenerSelect  = 0x000B,
kAudioUnitAddRenderNotifySelect         = 0x000F,
kAudioUnitRemoveRenderNotifySelect      = 0x0010,
kAudioUnitGetParameterSelect            = 0x0006,
kAudioUnitSetParameterSelect            = 0x0007,
kAudioUnitScheduleParametersSelect      = 0x0011,
kAudioUnitRenderSelect                  = 0x000E,
kAudioUnitResetSelect                   = 0x0009
```

Audio Unit Properties

These properties can be queried of any audio unit instance.

```
kAudioUnitProperty_ClassInfo                    = 0
kAudioUnitProperty_MakeConnection               = 1
kAudioUnitProperty_SampleRate                   = 2
kAudioUnitProperty_ParameterList                = 3
kAudioUnitProperty_ParameterInfo                = 4
kAudioUnitProperty_FastDispatch                 = 5
kAudioUnitProperty_CPULoad                      = 6
kAudioUnitProperty_StreamFormat                 = 8
kAudioUnitProperty_SRCAlgorithm                 = 9
kAudioUnitProperty_ReverbRoomType               = 10
kAudioUnitProperty_BusCount                     = 11
kAudioUnitProperty_ElementCount                 = kAudioUnitProperty_BusCount,
kAudioUnitProperty_Latency                      = 12
```

```
kAudioUnitProperty_SupportedNumChannels         = 13
kAudioUnitProperty_MaximumFramesPerSlice        = 14
kAudioUnitProperty_SetExternalBuffer            = 15
kAudioUnitProperty_ParameterValueStrings        = 16
kAudioUnitProperty_MIDIControlMapping           = 17
kAudioUnitProperty_GetUIComponentList           = 18
kAudioUnitProperty_AudioChannelLayout           = 19
kAudioUnitProperty_TailTime                     = 20
kAudioUnitProperty_BypassEffect                 = 21
kAudioUnitProperty_LastRenderError              = 22
kAudioUnitProperty_SetRenderCallback            = 23
kAudioUnitProperty_FactoryPresets               = 24
kAudioUnitProperty_ContextName                  = 25
kAudioUnitProperty_RenderQuality                = 26
kAudioUnitProperty_HostCallbacks                = 27
kAudioUnitProperty_CurrentPreset                = 28
kAudioUnitProperty_InPlaceProcessing            = 29
kAudioUnitProperty_ElementName                  = 30
kAudioUnitProperty_CocoaUI                      = 31
kAudioUnitProperty_SupportedChannelLayoutTags   = 32
kAudioUnitProperty_ParameterValueName           = 33
kAudioUnitProperty_ParameterIDName              = 34
kAudioUnitProperty_ParameterClumpName           = 35
kAudioUnitProperty_PresentPreset                = 36
ProperkAudioUnitProperty_UsesInternalReverb     = 1005
```

Music Device Properties

These properties can be queried of any music device audio unit instance.

```
kMusicDeviceProperty_InstrumentCount            = 1000
kMusicDeviceProperty_InstrumentName             = 1001
kMusicDeviceProperty_GroupOutputBus             = 1002
kMusicDeviceProperty_SoundBankFSSpec            = 1003
kMusicDeviceProperty_InstrumentNumber           = 1004
kMusicDeviceProperty_UsesInternalReverb         =
kAudioUnitProperty_UsesInternalReverb
kMusicDeviceProperty_MIDIXMLNames               = 1006
kMusicDeviceProperty_BankName                   = 1007
kMusicDeviceProperty_SoundBankData              = 1008
kMusicDeviceProperty_PartGroup                  = 1010
kMusicDeviceProperty_StreamFromDisk             = 1010
```

Output Unit Properties

These properties can be queried of any output audio unit instance.

```
kAudioOutputUnitProperty_CurrentDevice          = 2000
kAudioOutputUnitProperty_IsRunning              = 2001
kAudioOutputUnitProperty_ChannelMap             = 2002
kAudioOutputUnitProperty_EnableIO               = 2003
kAudioOutputUnitProperty_StartTime              = 2004
kAudioOutputUnitProperty_SetInputCallback       = 2005
kAudioOutputUnitProperty_HasIO                  = 2006
```

Various Audio Unit Properties

These properties can be queried of specific audio unit instances.

```
kAudioUnitProperty_SpatializationAlgorithm      = 3000
kAudioUnitProperty_SpeakerConfiguration         = 3001
kAudioUnitProperty_DopplerShift                 = 3002
kAudioUnitProperty_3DMixerRenderingFlags        = 3003
kAudioUnitProperty_3DMixerDistanceAtten         = 3004
kAudioUnitProperty_MatrixLevels                 = 3006
kAudioUnitProperty_MeteringMode                 = 3007
kAudioUnitProperty_PannerMode                   = 3008
kAudioUnitProperty_MatrixDimensions             = 3009
```

Offline Unit Properties

These properties can be queried of offline audio unit instances.

```
kAudioOfflineUnitProperty_InputSize             = 3020
kAudioOfflineUnitProperty_OutputSize            = 3021
kAudioUnitOfflineProperty_StartOffset           = 3022
kAudioUnitOfflineProperty_PreflightRequirements = 3023
kAudioUnitOfflineProperty_PreflightName         = 3024
```

Reverb Room-Type Properties

These properties can be queried of reverb audio unit instances.

```
kReverbRoomType_SmallRoom   = 0
kReverbRoomType_MediumRoom  = 1
kReverbRoomType_LargeRoom   = 2
kReverbRoomType_MediumHall  = 3
kReverbRoomType_LargeHall   = 4
kReverbRoomType_Plate       = 5
```

Spatialization Properties

These properties can be queried of panning audio unit instances.

```
kSpatializationAlgorithm_EqualPowerPanning      = 0
kSpatializationAlgorithm_SphericalHead          = 1
kSpatializationAlgorithm_HRTF                   = 2
kSpatializationAlgorithm_SoundField             = 3
kSpatializationAlgorithm_VectorBasedPanning     = 4
kSpatializationAlgorithm_StereoPassThrough      = 5
```

3D Mixer Properties

These properties can be queried of 3D Mixer audio unit instances.

```
k3DMixerRenderingFlags_InterAuralDelay          = (1L << 0)
k3DMixerRenderingFlags_DopplerShift             = (1L << 1)
k3DMixerRenderingFlags_DistanceAttenuation      = (1L << 2)
k3DMixerRenderingFlags_DistanceFilter           = (1L << 3)
```

```
k3DMixerRenderingFlags_DistanceDiffusion        = (1L << 4)
```

Render Quality Properties

These properties can be queried of any audio unit instance.

```
kRenderQuality_Max      = 0x7F
kRenderQuality_High     = 0x60
kRenderQuality_Medium   = 0x40
kRenderQuality_Low      = 0x20
kRenderQuality_Min      = 0
```

Panner Mode Properties

These properties can be queried of panning audio unit instances.

```
kPannerMode_Normal      = 0
kPannerMode_FaderMode   = 1
```

Offline Unit Preflight Properties

These properties can be queried of offline audio unit instances.

```
kOfflinePreflight_NotRequired   = 0
kOfflinePreflight_Optional      = 1
kOfflinePreflight_Required      = 2
```

Scope Properties

These properties can be queried of any audio unit instance.

```
kAudioUnitScope_Global  = 0
kAudioUnitScope_Input   = 1
kAudioUnitScope_Output  = 2
kAudioUnitScope_Group   = 3
kAudioUnitScope_Part    = 4
```

Preset Constants

These presets are used with `AUPreset` (page 141).

```
#define kAUPresetVersionKey         "version"
#define kAUPresetTypeKey            "type"
#define kAUPresetSubtypeKey         "subtype"
#define kAUPresetManufacturerKey    "manufacturer"
#define kAUPresetDataKey            "data"
#define kAUPresetNameKey            "name"
#define kAUPresetRenderQualityKey   "render-quality"
#define kAUPresetCPULoadKey         "cpu-load"
#define kAUPresetVSTDataKey         "vstdata"
#define kAUPresetElementNameKey     "element-name"
#define kAUPresetPartKey            "part"
```

Parameter Unit Constants

These presets are used with a parameter unit audio unit.

```
kAudioUnitParameterUnit_Generic             = 0
kAudioUnitParameterUnit_Indexed             = 1
kAudioUnitParameterUnit_Boolean             = 2
kAudioUnitParameterUnit_Percent             = 3
kAudioUnitParameterUnit_Seconds             = 4
kAudioUnitParameterUnit_SampleFrames        = 5
kAudioUnitParameterUnit_Phase               = 6
kAudioUnitParameterUnit_Rate                = 7
kAudioUnitParameterUnit_Hertz               = 8
kAudioUnitParameterUnit_Cents               = 9
kAudioUnitParameterUnit_RelativeSemiTones   = 10
kAudioUnitParameterUnit_MIDINoteNumber      = 11
kAudioUnitParameterUnit_MIDIController      = 12
kAudioUnitParameterUnit_Decibels            = 13
kAudioUnitParameterUnit_LinearGain          = 14
kAudioUnitParameterUnit_Degrees             = 15
kAudioUnitParameterUnit_EqualPowerCrossfade = 16
kAudioUnitParameterUnit_MixerFaderCurve1    = 17
kAudioUnitParameterUnit_Pan                 = 18
kAudioUnitParameterUnit_Meters              = 19
kAudioUnitParameterUnit_AbsoluteCents       = 20
kAudioUnitParameterUnit_Octaves             = 21
kAudioUnitParameterUnit_BPM                 = 22
kAudioUnitParameterUnit_Beats               = 23
kAudioUnitParameterUnit_Milliseconds        = 24
kAudioUnitParameterUnit_Ratio               = 25
```

Parameter Flags

These flags are used with an audio unit's parameters.

```
kAudioUnitParameterFlag_CFNameRelease       = (1L << 4)
kAudioUnitParameterFlag_HasClump            = (1L << 20)
kAudioUnitParameterFlag_HasName             = (1L << 21)
kAudioUnitParameterFlag_DisplayLogarithmic  = (1L << 22)
kAudioUnitParameterFlag_IsHighResolution    = (1L << 23)
kAudioUnitParameterFlag_NonRealTime         = (1L << 24)
kAudioUnitParameterFlag_CanRamp             = (1L << 25)
kAudioUnitParameterFlag_ExpertMode          = (1L << 26)
kAudioUnitParameterFlag_HasCFNameString     = (1L << 27)
kAudioUnitParameterFlag_IsGlobalMeta        = (1L << 28)
kAudioUnitParameterFlag_IsElementMeta       = (1L << 29)
kAudioUnitParameterFlag_IsReadable          = (1L << 30)
kAudioUnitParameterFlag_IsWritable          = (1L << 31)
```

MIDI Controller Parameters

These parameters are used with MIDI Controller audio units.

```
kAUGroupParameterID_Volume                  = 7
kAUGroupParameterID_Sustain                 = 64
kAUGroupParameterID_AllNotesOff             = 123
```

```
kAUGroupParameterID_ModWheel                = 1
kAUGroupParameterID_PitchBend               = 0xE0
kAUGroupParameterID_AllSoundOff             = 120
kAUGroupParameterID_ResetAllControllers     = 121
kAUGroupParameterID_Pan                     = 10
kAUGroupParameterID_Foot                    = 4
kAUGroupParameterID_ChannelPressure         = 0xD0
kAUGroupParameterID_KeyPressure             = 0xA0
kAUGroupParameterID_Expression              = 11
kAUGroupParameterID_DataEntry               = 6
kAUGroupParameterID_Volume_LSB              = kAUGroupParameterID_Volume + 32
kAUGroupParameterID_ModWheel_LSB            = kAUGroupParameterID_ModWheel + 32
kAUGroupParameterID_Pan_LSB                 = kAUGroupParameterID_Pan + 32
kAUGroupParameterID_Foot_LSB                = kAUGroupParameterID_Foot + 32
kAUGroupParameterID_Expression_LSB          = kAUGroupParameterID_Expression +
 32
kAUGroupParameterID_DataEntry_LSB           = kAUGroupParameterID_DataEntry +
32
kAUGroupParameterID_KeyPressure_FirstKey    = 256
kAUGroupParameterID_KeyPressure_LastKey     = 383
```

Bandpass Filter Unit Parameters

These parameters are used with the Bandpass Filter Unit.

```
kBandpassParam_CenterFrequency  = 0
kBandpassParam_Bandwidth        = 1
```

AUHipass Unit Parameters

These parameters are used with the AUHipass Unit.

```
kHighShelfParam_CutOffFrequency = 0
kHighShelfParam_Resonance       = 1
```

AULowpass Unit Parameters

These parameters are used with the AULowpass Unit.

```
kHighShelfParam_CutOffFrequency = 0
kHighShelfParam_Resonance       = 1
```

AUHighShelfFilter Unit Parameters

These parameters are used with the AUHighSelfFilter Unit.

```
kHipassParam_CutoffFrequency    = 0
kHipassParam_Gain               = 1
```

AULowShelfFilter Unit Parameters

These parameters are used with the AULowSelfFilter Unit.

```
kHipassParam_CutoffFrequency    = 0
kHipassParam_Gain               = 1
```

AUParametricEQ Unit Parameters

These parameters are used with the AUParametricEQ Unit.

```
kParametricEQParam_CenterFreq   = 0
kParametricEQParam_Q            = 1
kParametricEQParam_Gain         = 2
```

AUMatrixReverb Unit Parameters

These parameters are used with the AUMatrixReverb Unit.

```
kReverbParam_DryWetMix      = 0
kReverbParam_SmallLargeMix  = 1
kReverbParam_SmallSize      = 2
kReverbParam_LargeSize      = 3
kReverbParam_PreDelay       = 4
kReverbParam_LargeDelay     = 5
kReverbParam_SmallDensity   = 6
kReverbParam_LargeDensity   = 7
kReverbParam_LargeDelayRange= 8
kReverbParam_SmallBrightness= 9
kReverbParam_LargeBrightness= 10
kReverbParam_SmallDelayRange= 11
kReverbParam_ModulationRate = 12
kReverbParam_ModulationDepth= 13
```

Delay Unit Parameters

These parameters are used with the Delay Unit.

```
kDelayParam_WetDryMix   = 0
kDelayParam_DelayTime   = 1
kDelayParam_Feedback    = 2
kDelayParam_LopassCutoff= 3
```

AUPeakLimiter Unit Parameters

These parameters are used with the AUPeakLimiter Unit.

```
kLimiterParam_AttackTime = 0
kLimiterParam_DecayTime = 1
kLimiterParam_PreGain   = 2
```

AUDynamicsProcessor Unit Parameters

These parameters are used with the AUDynamicsProcessor Unit.

```
kDynamicsProcessorParam_Threshold           = 0
kDynamicsProcessorParam_HeadRoom            = 1
```

```
kDynamicsProcessorParam_ExpansionRatio      = 2
kDynamicsProcessorParam_ExpansionThreshold  = 3
kDynamicsProcessorParam_AttackTime          = 4
kDynamicsProcessorParam_ReleaseTime         = 5
kDynamicsProcessorParam_MasterGain          = 6
kDynamicsProcessorParam_CompressionAmount   = 1000
```

AUMultibandCompressor Unit Parameters

These parameters are used with the AUMultibandCompressor Unit.

```
kMultibandCompressorParam_Pregain           = 0
kMultibandCompressorParam_Postgain          = 1
kMultibandCompressorParam_Crossover1        = 2
kMultibandCompressorParam_Crossover2        = 3
kMultibandCompressorParam_Crossover3        = 4
kMultibandCompressorParam_Threshold1        = 5
kMultibandCompressorParam_Threshold2        = 6
kMultibandCompressorParam_Threshold3        = 7
kMultibandCompressorParam_Threshold4        = 8
kMultibandCompressorParam_Headroom1         = 9
kMultibandCompressorParam_Headroom2         = 10
kMultibandCompressorParam_Headroom3         = 11
kMultibandCompressorParam_Headroom4         = 12
kMultibandCompressorParam_AttackTime        = 13
kMultibandCompressorParam_ReleaseTime       = 14
kMultibandCompressorParam_EQ1               = 15
kMultibandCompressorParam_EQ2               = 16
kMultibandCompressorParam_EQ3               = 17
kMultibandCompressorParam_EQ4               = 18
kMultibandCompressorParam_CompressionAmount1 = 1000
kMultibandCompressorParam_CompressionAmount2 =
2000kMultibandCompressorParam_CompressionAmount3 = 3000
kMultibandCompressorParam_CompressionAmount4 = 4000
```

AUVarispeed Unit Parameters

These parameters are used with the AUVarispeed Unit.

```
kVarispeedParam_PlaybackRate    = 0
kVarispeedParam_PlaybackCents   = 1
```

3DMixer Unit Parameters

These parameters are used with the 3DMixer Unit.

```
k3DMixerParam_Azimuth           = 0
k3DMixerParam_Elevation         = 1
k3DMixerParam_Distance          = 2
k3DMixerParam_Gain              = 3
k3DMixerParam_PlaybackRate      = 4
k3DMixerParam_PreAveragePower   = 1000
k3DMixerParam_PrePeakHoldLevel  = 2000
k3DMixerParam_PostAveragePower  = 3000
k3DMixerParam_PostPeakHoldLevel = 4000
```

StereoMixer Unit Parameters

These parameters are used with the StereoMixer Unit.

```
kStereoMixerParam_Volume            = 0
kStereoMixerParam_Pan               = 1
kStereoMixerParam_PreAveragePower   = 1000
kStereoMixerParam_PrePeakHoldLevel  = 2000
kStereoMixerParam_PostAveragePower  = 3000
kStereoMixerParam_PostPeakHoldLevel = 4000
```

MatrixMixer Parameters

These parameters are used with the MatrixMixer Unit.

```
kMatrixMixerParam_Volume                    = 0
kMatrixMixerParam_Enable                    = 1
kMatrixMixerParam_PreAveragePower           = 1000
kMatrixMixerParam_PrePeakHoldLevel          = 2000
kMatrixMixerParam_PostAveragePower          = 3000
kMatrixMixerParam_PostPeakHoldLevel         = 4000
kMatrixMixerParam_PreAveragePowerLinear     = 5000
kMatrixMixerParam_PrePeakHoldLevelLinear    = 6000
kMatrixMixerParam_PostAveragePowerLinear    = 7000
kMatrixMixerParam_PostPeakHoldLevelLinear   = 8000
```

Output Unit Parameters

These parameters are used with Output Units.

```
kHALOutputParam_Volume = 14
```

DLSMusicDevice Parameters

These parameters are used with the DLSMusicDevice Unit.

```
kMusicDeviceParam_Tuning        = 0
kMusicDeviceParam_Volume        = 1
kMusicDeviceParam_ReverbVolume  = 2
```

Types

These basic types are common within the context of audio units.

```
typedef UInt32 AudioUnit
typedef UInt32 AudioUnitPropertyID
typedef UInt32 AudioUnitParameterID
typedef UInt32 AudioUnitScope
typedef UInt32 AudioUnitElement
typedef UInt32 AUParameterEventType
typedef UInt32 AudioUnitParameterUnit
```

Structures

These structures are used throughout the Audio Unit framework when working with properties and parameters.

AudioUnitParameter

Used by the Audio Unit Utilities to specify a parameter to be modified.

```
struct AudioUnitParameter {
  AudioUnit             mAudioUnit;
  AudioUnitParameterID  mParameterID;
  AudioUnitScope        mScope;
  AudioUnitElement      mElement;
};
```

Fields

`mAudioUnit`
The audio unit to be modified.

`mParameterID`
The parameter to be modified.

`mScope`
The scope in which the unit is being used.

`mElement`
The argument to be used with the parameter.

AudioUnitProperty

Used by the Audio Unit Utilities to specify a property to be modified.

```
struct AudioUnitProperty {
  AudioUnit             mAudioUnit;
  AudioUnitPropertyID   mPropertyID;
  AudioUnitScope        mScope;
  AudioUnitElement      mElement;
};
```

Fields

`mAudioUnit`
The audio unit to be modified.

`mPropertyID`
The parameter to be modified.

`mScope`
The scope in which the unit is being used.

`mElement`
The argument to be used with the property.

AudioUnitParameterEvent

Used to schedule a change in parameters during a render.

```
struct AudioUnitParameterEvent {
AudioUnitScope          scope;
AudioUnitElement        element;
AudioUnitParameterID    parameter;
AUParameterEventType    eventType;

union {
    struct {
        SInt32          startBufferOffset;
        UInt32          durationInFrames;
        Float32         startValue;
        Float32         endValue;
    } ramp;
    struct {
        UInt32          bufferOffset;
        Float32         value;
    } immediate;
} eventValues;

};
```

Fields

`scope`
The scope of the event.

`element`
Additional information about the scope of this event.

`parameter`
The parameter which is to be modified by the event.

`eventType`
A constant value; see `Parameter Event Types` (page 129).

`eventValues`
A union of the ramp and immediate event values.

`ramp`
The values if this event is a ramp event.

`immediate`
The values if this event is an immediate event.

`durationInFrames`
The length of the event, in frames.

`startBufferOffset`
The starting point of the event, after the beginning of the render.

`startValue`
The beginning value for the parameter.

`endValue`
The audio unit to be modifiedThe ending value of the parameter.

`bufferOffset`
Where in the current buffer the event should occur.

`value`
The value the parameter should be changed to.

AudioUnitConnection

Connects audio units together.

```
struct AudioUnitConnection {
    AudioUnit   sourceAudioUnit;
    UInt32      sourceOutputNumber;
    UInt32      destInputNumber;
}
```

Fields

`sourceAudioUnit`
The audio unit where data is coming from.

`sourceOutputNumber`
The output bus on the source audio unit.

`destInputNumber`
The destination bus on the recieving audio unit.

AURenderCallbackStruct

Encapsulates render callback information.

```
struct AURenderCallbackStruct {
    AURenderCallback    inputProc;
    void *              inputProcRefCon; }
```

Fields

`inputProc`
The callback function.

`inoutProcRefCon`
Any arguments that should be passed to the callback.

AudioUnitExternalBuffer

Encapsulates information about an external buffer.

```
struct AudioUnitExternalBuffer {
    Byte*   buffer;
    UInt32  size;
}
```

Fields

`buffer`
A pointer to a buffer of audio data.

`size`
The size of the external buffer.

AUChannelInfo

Encapsulates channel information used in a connection.

```
struct AUChannelInfo {
    SInt16  inChannels;
    SInt16  outChannels;
}
```

Fields

`inChannels`
: The number of channels on input.

`outChannels`
: The number of channels on output.

AUPreset

Encapsulates channel information used in a connection.

```
struct AUPreset {
    SInt32      presetNumber;
    CFStringRef presetName;
}
```

Fields

`presetNumber`
: An arbitrary value for a preset.

`presetName`
: The name for a preset.

HostCallbackInfo

Encapsulates callbacks to a host for information.

```
struct HostCallbackInfo {
    void*                               hostUserData;
    HostCallback_GetBeatAndTempo        beatAndTempoProc;
    HostCallback_GetMusicalTimeLocation musicalTimeLocationProc;
}
```

Fields

`hostUserData`
: Additional informaiton needed by the callbacks.

`beatAndTempoProc`
: A callback that determines beat and tempo.

`musicTimeLocationProc`
: A callback that determines the musical time, as numerator and denominator.

AudioUnitCocoaViewInfo

Encapsulates the information needed for a Cocoa view.

```
struct AudioUnitCocoaViewInfo {
    CFURLRef    mCocoaAUViewBundleLocation;
    CFStringRef mCocoaAUViewClass[1];
}
```

Fields

`mCocoaAUViewBundleLocation`
The location of the user interface bundle.

`mCocoaAUViewClass`
The names of the classes that implement the required protocol for an `AUView`.

AudioUnitParameterValueName

Encapsulates the information needed when determining a parameter value's name.

```
struct AudioUnitParameterValueName {
    AudioUnitParameterID    inParamID;
    Float32*                inValue;
    CFStringRef             outName;
}
```

Fields

`inParamID`
The parameter in question.

`inValue`
The value being queried upon.

`outName`
The name corresponding to the value.

AudioUnitParameterNameInfo

Encapsulates the information needed when determining a parameter value's name.

```
struct AudioUnitParameterNameInfo {
    UInt32          inID;
    SInt32          inDesiredLength;
    CFStringRef     outName;
}
```

Fields

`inID`
The parameter in question.

`inDesiredLangth`
The desired length of the string.

`outName`
The name corresponding to the parameter.

AudioUnitParameterInfo

Encapsulates an audio units parameter information.

```
struct AudioUnitParameterInfo {
    char                    name[56];
    UInt32                  clumpID;
    CFStringRef             cfNameString;
    AudioUnitParameterUnit  unit;
    Float32                 minValue;
    Float32                 maxValue;
    Float32                 defaultValue;
    UInt32                  flags;
}
```

Fields

`name`
The name of the parameter.

`clumpID`
The grouping to which the parameter belongs.

`cfNameString`
The name of this parameter as a `CFString`.

`unit`
The parameter unit for this parameter.

`minValue`
The smallest value for the parameter.

`maxValue`
The largest value for the parameter.

`defaultValue`
The default value for this parameter.

`flags`
Any flags that the parameter has attached to it.

AudioUnitMIDIControlMapping

Encapsulates MIDI and corresponding audio unit information.

```
struct AudioUnitMIDIControlMapping {
    UInt16                  midiNRPN;
    UInt8                   midiControl;
    UInt8                   scope;
    AudioUnitElement        element;
    AudioUnitParameterID    parameter;
}
```

Fields

`midiNRPN`
The MIDI note information.

`midiControl`
The MIDI control information.

`scope`
The scope of the mapping.

`element`
Additional scope information.

`parameter`
The parameter to which the MIDI data is to be applied.

AudioOutputUnitStartAtTimeParam

Encapsulates the information needed when a parameter is to take effect at a certain time.

```
struct AudioOutputUnitStartAtTimeParams {
    AudioTimeStamp  mTimestamp;
    UInt32          mFlags;
}
```

Fields
`mTimestamp`
The start time for the parameter.
`mFlags`
The flags for this parameter event.

Functions

These functions provide the bulk of the functionality of the Audio Unit framework, and are needed when using or developing an audio unit.

AudioUnitInitialize

Initializes an audio unit instance.

```
ComponentResult AudioUnitInitialize(
AudioUnit           ci
)
```

Parameters
ci
The audio unit to be initialized.
Availability
Available in Mac OS X v10.0 and later.

Declared In
`AUComponent.h`

AudioUnitUninitialize

Uninitializes an audio unit instance.

```
ComponentResult AudioUnitUninitialize(
AudioUnit ci
)
```

Parameters

ci

The audio unit to be uninitialized.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUComponent.h

AudioUnitRender

Performs the action on a buffer of audio data.

```
ComponentResult AudioUnitRender(
  AudioUnit                       ci,
  AudioUnitRenderActionFlags*     ioActionFlags,
  const AudioTimeStamp*           inTimeStamp,
  UInt32                          inOutputBusNumber,
  UInt32                          inNumberFrames,
  AudioBufferList*                ioData
)
```

Parameters

ci

The audio unit to be changed.

ioActionFlags

Flags that provide information on the render; see "Render Action Flags" (page 128).

inTimeStamp

The time the render is begun.

inOutputBusNumber

The bus on which the output will be placed.

inNumberFrames

The number of frames to be rendered.

ioData

The audio data, before and after the render.

Availability
Available in Mac OS X v10.2 and later.

Declared In
AUComponent.h

AudioUnitReset

Resets an audio unit.

```
ComponentResult AudioUnitReset(
  AudioUnit          ci,
  AudioUnitScope     inScope,
  AudioUnitElement   inElement
)
```

Parameters

ci

The audio unit to be reset.

inScope

The scope in which the unit is to be reset.

inElement

Additional information about the scope of the audio unit.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUComponent.h

AudioUnitGetPropertyInfo

Returns the size of the data that will be returned when calling `AudioUnitGetProperty` (page 147) for the specified property.

```
ComponentResult AudioUnitGetPropertyInfo(
  AudioUnit             ci,
  AudioUnitPropertyID   inID,
  AudioUnitScope        inScope,
  AudioUnitElement      inElement,
  UInt32*               outDataSize,
  Boolean*              outWritable
)
```

Parameters

ci

The audio unit on which the property is to be queried.

inID

The property to be queried upon.

inScope

The scope in which the property is applicable.

inElement

Further specifies the scope of the property.

outDataSize

The size, in bytes, of the property.

outWritable

A boolean showing if the property is writable.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUComponent.h

AudioUnitGetProperty

Returns the size of the data for a specified property.

```
ComponentResult AudioUnitGetProperty(
  AudioUnit             ci,
  AudioUnitPropertyID   inID,
  AudioUnitScope        inScope,
  AudioUnitElement      inElement,
  void*                 outData,
  UInt32*               ioDataSize
)
```

Parameters

ci
The audio unit on which the property is to be queried.

inID
The property to be queried upon.

inScope
The scope in which the property is applicable.

inElement
Further specifies the scope of the property.

outData
A pointer to the data corresponding to the property.

ioDataSize
The expected data size and the actual data size returned.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUComponent.h

AudioUnitSetProperty

Sets a property's value to the supplied value.

```
ComponentResult AudioUnitSetProperty(
  AudioUnit             ci,
  AudioUnitPropertyID   inID,
  AudioUnitScope        inScope,
  AudioUnitElement      inElement,
  const void*           inData,
  UInt32                inDataSize
)
```

Parameters

ci

The audio unit on which the property is to be applied.

inID

The property to be modified.

inScope

The scope in which the property is applicable.

inElement

Further specifies the scope of the property.

inData

A pointer to the data to be applied to the property.

inDataSize

The size of the data being passed in.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUComponent.h

AudioUnitAddPropertyListener

Monitors an audio unit's property for changes and issues a callback notification upon the change.

```
ComponentResult AudioUnitAddPropertyListener(
  AudioUnit                       ci,
  AudioUnitPropertyID             inID,
  AudioUnitPropertyListenerProc   inProc,
  void*                           inProcRefCon
)
```

Parameters

ci

The audio unit to be monitored.

inID

The property to be monitored.

inProc

The callback to be made when a property is changed.

inProcRefCon

Additional parameters to be passed to the callback.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUComponent.h

AudioUnitRemovePropertyListener

Removes the specified listener from a property.

```
ComponentResult AudioUnitRemovePropertyListener(
  AudioUnit                       ci,
  AudioUnitPropertyID             inID,
  AudioUnitPropertyListenerProc   inProc
)
```

Parameters

ci

The audio unit being monitored.

inID

The property being monitored.

inProc

The callback to be removed.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.5.
Not available to 64-bit applications.

Declared In
AUComponent.h

AudioUnitAddRenderNotify

Specifies a callback to be used before and after an audio unit render occurs.

```
ComponentResult AudioUnitAddRenderNotify(
  AudioUnit         ci,
  AURenderCallback  inProc,
  void*             inProcRefCon
)
```

Parameters

ci

The audio unit to bemonitored.

inProc

The callback to be issued.

inProcRefCon

Additional parameters to be passed to the callback.

Availability
Available in Mac OS X v10.2 and later.

Declared In
AUComponent.h

AudioUnitRemoveRenderNotify

Removes a callback from an audio unit.

```
ComponentResult AudioUnitRemoveRenderNotify(
  AudioUnit          ci,
  AURenderCallback   inProc,
  void *             inProcRefCon
)
```

Parameters

ci

The audio unit being monitored.

inProc

The callback being issued.

inProcRefCon

Additional parameters to be passed to the callback.

Availability

Available in Mac OS X v10.2 and later.

Declared In

AUComponent.h

AudioUnitGetParameter

Returns the current value for a parameter.

```
ComponentResult AudioUnitGetParameter(
  AudioUnit             ci,
  AudioUnitParameterID  inID,
  AudioUnitScope        inScope,
  AudioUnitElement      inElement,
  Float32*              outValue
)
```

Parameters

ci

The audio unit being queried.

inID

The parameter being queried.

inScope

The scope in which the parameter works.

inElement

Additional information about the scope of the parameter.

outValue

The current value of the parameter.

Availability

Available in Mac OS X v10.0 and later.

Declared In

AUComponent.h

AudioUnitSetParameter

Returns the current value for a parameter.

```
ComponentResult AudioUnitSetParameter(
  AudioUnit             ci,
  AudioUnitParameterID  inID,
  AudioUnitScope        inScope,
  AudioUnitElement      inElement,
  Float32               inValue,
  UInt32                inBufferOffsetInFrames
)
```

Parameters

ci

The audio unit to be changed.

inID

The parameter to be changed.

inScope

The scope in which the parameter works.

inElement

Additional information about the scope of the parameter.

inValue

The new value for the parameter.

inBufferOffsetInFrames

When in the next render the parameter should be changed.

Availability

Available in Mac OS X v10.0 and later.

Declared In

AUComponent.h

AudioUnitScheduleParameters

Adds events that change certain parameters.

```
ComponentResult AudioUnitScheduleParameters(
  AudioUnit                         ci,
  const AudioUnitParameterEvent*    inParameterEvent,
  UInt32                            inNumParamEvents )
```

Parameters

ci

The audio unit to be changed.

inParameterEvent

An event or events to be placed.

inNumParamEvents

The number of events being added.

Availability

Available in Mac OS X v10.2 and later.

Declared In
AUComponent.h

Callbacks

These callbacks are provided by you and are used throughout the Audio Unit framework.

AURenderCallback

A callback set by `kAudioUnitProperty_SetRenderCallback` for performing an audio unit's render.

```
typedef CALLBACK_API_C( OSStatus , AURenderCallback )(void *inRefCon,
AudioUnitRenderActionFlags *ioActionFlags, const AudioTimeStamp *inTimeStamp, UInt32
 inBusNumber, UInt32 inNumberFrames, AudioBufferList *ioData)

OSStatus AURenderCallback(
void *      inRefCon;
AudioUnitRenderActionFlags      * ioActionFlags;
const AudioTimeStamp    *   inTimeStamp;
UInt32       inBusNumber;
UInt32       inNumberFrames;
AudioBufferList*       ioData
)
```

Parameters

inRefCon
Parameters passed to the callback.

ioActionFlags
Flags for rendering options.

inTimeStamp
The time that the callback is invoked.

inBusNumber
The bun on which data will be supplied.

inNumberFrames
The number of frames to be rendered.

ioData
The audio data to be rendered upon.

Availability
Available in Mac OS X v10.2 and later.

Declared In
AUComponent.h

AudioUnitPropertyListenerProc

A callback set by `kAudioUnitProperty_SetRenderCallback` for performing an audio unit's render.

```
typedef CALLBACK_API_C( void , AudioUnitPropertyListenerProc )(void *inRefCon,
AudioUnit ci, AudioUnitPropertyID inID, AudioUnitScope inScope, AudioUnitElement
inElement);

void AudioUnitPropertyListenerProc(
void *    inRefCon,
AudioUnit     ci,
AudioUnitPropertyID     inID,
AudioUnitScope     inScope,
AudioUnitElement     inElement
)
```

Parameters

inRefCon

Parameters passed to the callback.

ci

The audio unit whose property was modified.

inID

The property that was changed.

inScope

The scope of the unit and the property that was changed.

inElement

The value that the property was changed to.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUComponent.h

AudioUnitGetParameterProc

A callback for returning developer specified parameter values.

```
typedef CALLBACK_API_C( ComponentResult , AudioUnitGetParameterProc )(void
*inComponentStorage, AudioUnitParameterID inID, AudioUnitScope inScope,
AudioUnitElement inElement, Float32 *outValue);

ComponentResult AudioUnitGetParameterProc(
void*      inComponentStorage,
AudioUnitParameterID      inID,
AudioUnitScope      inScope,
AudioUnitElement      inElement,
Float32*      outValue
)
```

Parameters

inComponentStore

A pointer to the audio unit.

inID

The parameter being queried.

inScope
The scope of the parameter.

inElement
Additional information about the scope of the parameter.

outValue
The value returned for the parameter.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUComponent.h

AudioUnitSetParameterProc

A callback for setting developer specified parameter values.

```
typedef CALLBACK_API_C( ComponentResult , AudioUnitSetParameterProc )(void
*inComponentStorage, AudioUnitParameterID inID, AudioUnitScope inScope,
AudioUnitElement inElement, Float32 inValue, UInt32 inBufferOffsetInFrames);

ComponentResult AudioUnitSetParameterProc(
void*       inComponentStorage
AudioUnitParameterID      inID
AudioUnitScope      inScope
AudioUnitElement      inElement
Float32      inValue
UInt32      inBufferOffsetInFrames
)
```

Parameters

inComponentStore
A pointer to the audio unit.

inID
The parameter to be set.

inScope
The scope of the parameter.

inElement
Additional information about the scope of the parameter.

inValue
The value for the parameter to be set.

inBufferOffsetInFrames
The place in the buffer where the parameter change is to happen.

Availability
Available in Mac OS X v10.0 and later.

Declared In
AUComponent.h

AudioUnitRenderProc

A callback for setting developer specified parameter values.

```
typedef CALLBACK_API_C( ComponentResult , AudioUnitRenderProc )(void
*inComponentStorage, AudioUnitRenderActionFlags *ioActionFlags, const AudioTimeStamp
 *inTimeStamp, UInt32 inOutputBusNumber, UInt32 inNumberFrames, AudioBufferList
*ioData);

ComponentResult AudioUnitRenderProc(
void *      inComponentStorage,
AudioUnitRenderActionFlags*       ioActionFlags,
const AudioTimeStamp    *   inTimeStamp,
UInt32       inOutputBusNumber,
UInt32       inNumberFrames,
AudioBufferList*       ioData
)
```

Parameters

inComponentStorage

A pointer to the audio unit.

ioActionFlags

Flags that provide information on the render; see "Render Action Flags" (page 128).

inTimeStamp

The time the render is begun.

inOutputBusNumber

The bus on which the output will be placed.

inNumberFrames

The number of frames to be rendered.

ioData

The audio data, before and after the render.

Availability

Available in Mac OS X v10.2 and later.

Declared In

AUComponent.h

HostCallback_GetBeatAndTempo

A callback for setting developer specified parameter values.

```
typedef OSStatus (*HostCallback_GetBeatAndTempo) (void * inHostUserData, Float64
*outCurrentBeat, Float64    * outCurrentTempo);

OSStatus HostCallback_GetBeatAndTempo (
void    inHostUserData,
Float64 *   outCurrentBeat,
Float64 *    outCurrentTempo
)
```

Parameters

inHostUserData

Any arguments needed by the callback.

outCurrentBeat

The beat of the buffered data.

outCurrentTempo

The tempo of the buffered data.

Availability
Available in Mac OS X v10.2 and later.

Declared In
AudioUnitProperties.h

HostCallback_GetBeatAndTempo

A callback for setting developer specified parameter values.

```
typedef OSStatus (*HostCallback_GetMusicalTimeLocation) (void *inHostUserData,
UInt32 *outDeltaSampleOffsetToNextBeat, Float32 *outTimeSig_Numerator, UInt32
*outTimeSig_Denominator, Float64 *outCurrentMeasureDownBeat);

OSStatus HostCallback_GetMusicalTimeLocation(
void*    inHostUserData,
UInt32*    outDeltaSampleOffsetToNextBeat,
Float32*    outTimeSig_Numerator,
UInt32*    outTimeSig_Denominator,
Float64*    outCurrentMeasureDownBeat
)
```

Parameters

inHostUserData

Any arguments needed by the callback.

outDeltaDampleOffsetToNextBeat

The average time between beats.

outTimeSig_Numerator

The numerator of the time signature.

outTimeSig_Denominator

The denominator of the time signature.

outCurrentMeasureDownBeat

The beats in the current measure.

Availability
Available in Mac OS X v10.2 and later.

Declared In
AudioUnitProperties.h

HostCallback_GetBeatAndTempo

A callback for setting developer specified parameter values.

```
typedef OSStatus (*HostCallback_GetMusicalTimeLocation) (void *inHostUserData,
UInt32 *outDeltaSampleOffsetToNextBeat, Float32 *outTimeSig_Numerator, UInt32
*outTimeSig_Denominator, Float64 *outCurrentMeasureDownBeat);

OSStatus HostCallback_GetMusicalTimeLocation(
void*    inHostUserData,
UInt32*    outDeltaSampleOffsetToNextBeat,
Float32*    outTimeSig_Numerator,
UInt32*    outTimeSig_Denominator,
Float64*    outCurrentMeasureDownBeat
)
```

Parameters

inHostUserData

Any arguments needed by the callback.

outDeltaDampleOffsetToNextBeat

The average time between beats.

outTimeSig_Numerator

The numerator of the time signature.

outTimeSig_Denominator

The denominator of the time signature.

outCurrentMeasureDownBeat

The beats in the current measure.

Availability

Available in Mac OS X v10.2 and later.

Declared In

AudioUnitProperties.h

Core Audio Types Reference

This chapter describes the structures and constants shared throughout all portions of Core Audio.

The `CoreAudioTypes.h` file contains general structures and typedefs that are found and used throughout Core Audio, including structures that represent an audio buffer, a structure describing the particular format of an audio stream, channel layout structures, and structures for timing information.

Audio Value Structures

AudioValueRange

Represents a continuous range of values.

```
typedef struct AudioValueRange{
Float64 mMinimum;
Float64 mMaximum;
} AudioValueRange;
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`CoreAudioTypes.h`

AudioValueTranslation

Contains an input and output buffer and associated size values, for translation use.

```
typedef struct AudioValueTranslation {
void* mInputData;
UInt32 mInputDataSize;
void* mOutputData;
UInt32 mOutputDataSize;
} AudioValueTranslation;
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`CoreAudioTypes.h`

Audio Buffer Structures

AudioBuffer

A single buffer and the associated data.

```
typedef struct AudioBuffer {
UInt32 mNumberChannels;
UInt32 mDataByteSize;
void* mData;
} AudioBuffer;
```

Discussion
This structure is not used on its own, but with `AudioBufferList` as one of its data members. An instance of `AudioBuffer` keeps track of the number of channels in the buffer, the size of the buffer, and a pointer to the buffer data.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`CoreAudioTypes.h`

AudioBufferList

Keeps track of multiple buffers.

```
typedef struct AudioBufferList {
UInt32 mNumberBuffers;
AudioBuffer mBuffers[1];
} AudioBufferList;
```

Discussion
When audio data is interleaved, only one buffer is needed in the AudioBufferList; when dealing with multiple mono channels, each will need its own buffer. This is accomplished by allocating the needed space and pointing *`mBuffers`* to it.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`CoreAudioTypes.h`

Audio Stream Basic Description

AudioStreamBasicDescription

Contains all the information needed for describing streams of audio data.

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

Discussion
The `AudioStreamBasicDescription` is the fundamental descriptive structure in Core Audio. The "Audio Format" (page 45) API deals extensively with `AudioStreamBasicDescription`, as do most other parts of Core Audio.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`CoreAudioTypes.h`

Format IDs

These values reflect various audio formats available from within Core Audio, and are used to populate the *`mFormatID`* field in `AudioStreamBasicDescription` instances.

```
kAudioFormatLinearPCM = 'lpcm'
kAudioFormatAC3 = 'ac-3'
kAudioFormat60958AC3 ='cac3'
kAudioFormatMPEG = 'mpeg'
kAudioFormatAppleIMA4 = 'ima4'
kAudioFormatMPEG4AAC = 'aac '
kAudioFormatMPEG4CELP = 'celp'
kAudioFormatMPEG4HVXC = 'hvxc'
kAudioFormatMPEG4TwinVQ = 'twvq'
kAudioFormatTimeCode = 'time'
kAudioFormatMIDIStream = 'midi'
kAudioFormatParameterValueStream = 'apvs'
```

Format Flags

These values are used to fill the *`mFormatFlags`* field of an `AudioStreamBasicDescription`, and reflect the formatting of the audio stream data.

Standard flags:

```
kAudioFormatFlagIsFloat = (1L << 0)
kAudioFormatFlagIsBigEndian = (1L << 1)
kAudioFormatFlagIsSignedInteger = (1L << 2)
kAudioFormatFlagIsPacked = (1L << 3)
kAudioFormatFlagIsAlignedHigh = (1L << 4)
kAudioFormatFlagIsNonInterleaved = (1L << 5)
```

```
kAudioFormatFlagsAreAllClear = (1L << 31)
```

Linear PCM flags:

```
kLinearPCMFormatFlagIsFloat = kAudioFormatFlagIsFloat
kLinearPCMFormatFlagIsBigEndian = kAudioFormatFlagIsBigEndian
kLinearPCMFormatFlagIsSignedInteger = kAudioFormatFlagIsSignedInteger
kLinearPCMFormatFlagIsPacked = kAudioFormatFlagIsPacked
kLinearPCMFormatFlagIsAlignedHigh = kAudioFormatFlagIsAlignedHigh
kLinearPCMFormatFlagIsNonInterleaved = kAudioFormatFlagIsNonInterleaved
kLinearPCMFormatFlagsAreAllClear = kAudioFormatFlagsAreAllClear
```

Audio Stream Packet Description

AudioStreamPacketDescription

```
typedef struct AudioStreamPacketDescription {
SInt64 mStartOffset;
UInt64 mLength;
} AudioStreamPacketDescription;
```

Availability
Available in Mac OS X v10.2 and later.

Declared In
`CoreAudioTypes.h`

SMPTE Time

SMPTETime

SMPTE time is a format used to sync audio and video streams, based on video framing.

```
typedef struct SMPTETime {
UInt64 mCounter;
UInt32 mType;
UInt32 mFlags;
SInt16 mHours;
SInt16 mMinutes;
SInt16 mSeconds;
SInt16 mFrames;
} SMPTETime;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`CoreAudioTypes.h`

SMPTE Types

These constants are used for the *mType* value in SMPTETime to specify different frame rates.

```
kSMPTETimeType24 = 0
kSMPTETimeType25 = 1
kSMPTETimeType30Drop = 2
kSMPTETimeType30 = 3
kSMPTETimeType2997 = 4
kSMPTETimeType2997Drop = 5
```

SMPTE Time Stamps

These constants are used for the *mFlags* value in SMPTETime.

```
kSMPTETimeValid = (1L << 0)
kSMPTETimeRunning = (1L << 1)
```

Audio Time Stamp

AudioTimeStamp

Encapsulates time stamp information in various formats.

```
typedef struct AudioTimeStamp {
Float64 mSampleTime;
UInt64 mHostTime;
Float64 mRateScalar;
UInt64 mWordClockTime;
SMPTETime mSMPTETime;
UInt32 mFlags;
UInt32 mReserved;
} AudioTimeStamp;
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
CoreAudioTypes.h

Time Stamp Flags

The *mFlags* value in AudioTimeStamp uses these values, which signify which time formats are valid.

```
kAudioTimeStampSampleTimeValid = (1L << 0),
kAudioTimeStampHostTimeValid = (1L << 1),
kAudioTimeStampRateScalarValid = (1L << 2),
kAudioTimeStampWordClockTimeValid = (1L << 3),
kAudioTimeStampSMPTETimeValid = (1L << 4)
```

Audio Channel Layouts

AudioChannelDescription

Contains information describing a single channel.

```
typedef struct AudioChannelDescription {
AudioChannelLabel mChannelLabel;
UInt32 mChannelFlags;
Float32 mCoordinates[3];
} AudioChannelDescription;
```

Discussion
This structure is used by `AudioChannelLayout` to describe the position of a speaker. The *`mChannelFlags`* data member contains a "Channel Labels" (page 165) value, while *`mChannelFlags`* marks which coordinate system is in use. The position of the speaker is kept in *`mCoordinates`*, as per the "Channel Flags" (page 166) and "Channel Coordinates" (page 166).

Availability
Available in Mac OS X v10.2 and later.

Declared In
`CoreAudioTypes.h`

AudioChannelLayout

Specifies the channel layout in files and hardware.

```
typedef struct AudioChannelLayout {
AudioChannelLayoutTag mChannelLayoutTag;
UInt32 mChannelBitmap;
UInt32 mNumberChannelDescriptions;
AudioChannelDescription mChannelDescriptions[1];
} AudioChannelLayout;
```

Discussion
This structure is used to keep track of the channel arrangements. A "Channel Layout Tags" (page 167) value, stored in *`mChannelLayoutTag`*, signifies which layout scheme is in use, or, if the scheme is not available there, *`mChannelBitmap`* may contain a bitmap describing the layout. The bitmap is formed using "Channel Bitmaps" (page 166).

Availability
Available in Mac OS X v10.2 and later.

Declared In
`CoreAudioTypes.h`

Defined Data Types

Typedefs are provided to help describe channel layouts.

```
typedef UInt32 AudioChannelLabel;
typedef UInt32 AudioChannelLayoutTag;
```

Channel Labels

Specifies which channel is described by an `AudioChannelDescription`.

Unknown/unused:

```
kAudioChannelLabel_Unknown = 0xFFFFFFFF
kAudioChannelLabel_Unused = 0
```

Standard channels:

```
kAudioChannelLabel_Left = 1
kAudioChannelLabel_Right= 2
kAudioChannelLabel_Center = 3
kAudioChannelLabel_LFEScreen = 4
kAudioChannelLabel_LeftSurround = 5
kAudioChannelLabel_RightSurround = 6
kAudioChannelLabel_LeftCenter = 7
kAudioChannelLabel_RightCenter = 8
kAudioChannelLabel_CenterSurround = 9
kAudioChannelLabel_LeftSurroundDirect = 10
kAudioChannelLabel_RightSurroundDirect = 11
kAudioChannelLabel_TopCenterSurround = 12
kAudioChannelLabel_VerticalHeightLeft = 13
kAudioChannelLabel_VerticalHeightCenter = 14
kAudioChannelLabel_VerticalHeightRight = 15
kAudioChannelLabel_TopBackLeft = 16
kAudioChannelLabel_TopBackCenter = 17
kAudioChannelLabel_TopBackRight = 18
kAudioChannelLabel_RearSurroundLeft = 33
kAudioChannelLabel_RearSurroundRight = 34
kAudioChannelLabel_LeftWide = 35
kAudioChannelLabel_RightWide = 36
kAudioChannelLabel_LFE2 = 37
kAudioChannelLabel_LeftTotal = 38
kAudioChannelLabel_RightTotal = 39
kAudioChannelLabel_HearingImpaired = 40
kAudioChannelLabel_Narration = 41
kAudioChannelLabel_Mono = 42
kAudioChannelLabel_DialogCentricMix = 43
```

First order Ambisonic channels:

```
kAudioChannelLabel_Ambisonic_W = 200
kAudioChannelLabel_Ambisonic_X = 201
kAudioChannelLabel_Ambisonic_Y = 202
kAudioChannelLabel_Ambisonic_Z = 203
```

Mid/side Recording:

```
kAudioChannelLabel_MS_Mid = 204
kAudioChannelLabel_MS_Side = 205
```

X-Y Recording:

```
kAudioChannelLabel_XY_X = 206
kAudioChannelLabel_XY_Y = 207
```

Other channels:

```
kAudioChannelLabel_HeadphonesLeft = 301
kAudioChannelLabel_HeadphonesRight = 302
kAudioChannelLabel_ClickTrack = 304
kAudioChannelLabel_ForeignLanguage = 305
```

Channel Bitmaps

Used in the *mChannelBitmap* field of an `AudioChannelLayout` to specify a custom layout.

```
kAudioChannelBit_Left = (1L<<0)
kAudioChannelBit_Right = (1L<<1)
kAudioChannelBit_Center = (1L<<2)
kAudioChannelBit_LFEScreen = (1L<<3)
kAudioChannelBit_LeftSurround = (1L<<4)
kAudioChannelBit_RightSurround = (1L<<5)
kAudioChannelBit_LeftCenter = (1L<<6)
kAudioChannelBit_RightCenter = (1L<<7)
kAudioChannelBit_CenterSurround = (1L<<8)
kAudioChannelBit_LeftSurroundDirect = (1L<<9)
kAudioChannelBit_RightSurroundDirect = (1L<<10)
kAudioChannelBit_TopCenterSurround = (1L<<11)
kAudioChannelBit_VerticalHeightLeft = (1L<<12)
kAudioChannelBit_VerticalHeightCenter = (1L<<13)
kAudioChannelBit_VerticalHeightRight = (1L<<14)
kAudioChannelBit_TopBackLeft = (1L<<15)
kAudioChannelBit_TopBackCenter = (1L<<16)
kAudioChannelBit_TopBackRight = (1L<<17)
```

Channel Flags

Specifies which coordinate system is in use, and if the distances are measured in meters; stored in *mChannelFields* in an `AudioChannelDescription` instance.

```
kAudioChannelFlags_RectangularCoordinates = (1L<<0)
kAudioChannelFlags_SphericalCoordinates = (1L<<1)
kAudioChannelFlags_Meters = (1L<<2)
```

Channel Coordinates

Specifies the meaning of indices of *mCoordinates* in `AudioChannelDescription`.

For rectangular coordinates:

```
kAudioChannelCoordinates_LeftRight = 0
kAudioChannelCoordinates_BackFront = 1
```

```
kAudioChannelCoordinates_DownUp = 2
```

For spherical coordinates:

```
kAudioChannelCoordinates_Azimuth = 0
kAudioChannelCoordinates_Elevation = 1
kAudioChannelCoordinates_Distance = 2
```

Channel Layout Tags

Specifies which channel layout is in use; stored in *mChannelLayout* in `AudioChannelLayout`.

Other/unknown:

```
kAudioChannelLayoutTag_UseChannelDescriptions = 0
kAudioChannelLayoutTag_UseChannelBitmap = 1
kAudioChannelLayoutTag_AllUnknown = 9
```

General layouts:

```
kAudioChannelLayoutTag_Mono = 100
kAudioChannelLayoutTag_Stereo = 101
kAudioChannelLayoutTag_StereoHeadphones = 2
kAudioChannelLayoutTag_MatrixStereo = 3
kAudioChannelLayoutTag_MidSide = 4
kAudioChannelLayoutTag_XY = 5
kAudioChannelLayoutTag_Binaural = 6
kAudioChannelLayoutTag_Quadraphonic = 7
kAudioChannelLayoutTag_Ambisonic_B_Format = 8
kAudioChannelLayoutTag_AudioUnit_5_0 = 107
```

MPEG defined layouts:

```
kAudioChannelLayoutTag_MPEG_1_0 = 100
kAudioChannelLayoutTag_MPEG_2_0 = 101
kAudioChannelLayoutTag_MPEG_3_0_A = 102
kAudioChannelLayoutTag_MPEG_3_0_B = 103
kAudioChannelLayoutTag_MPEG_4_0_A = 104
kAudioChannelLayoutTag_MPEG_4_0_B = 105
kAudioChannelLayoutTag_MPEG_5_0_A = 106
kAudioChannelLayoutTag_MPEG_5_0_B = 107
kAudioChannelLayoutTag_MPEG_5_0_C = 108
kAudioChannelLayoutTag_MPEG_5_0_D = 109
kAudioChannelLayoutTag_MPEG_5_1_A = 110
kAudioChannelLayoutTag_MPEG_5_1_B = 111
kAudioChannelLayoutTag_MPEG_5_1_C = 112
kAudioChannelLayoutTag_MPEG_5_1_D = 113
kAudioChannelLayoutTag_MPEG_6_1_A = 114
kAudioChannelLayoutTag_MPEG_7_1_A = 115
kAudioChannelLayoutTag_MPEG_7_1_B = 116
kAudioChannelLayoutTag_MPEG_7_1_C = 117
kAudioChannelLayoutTag_Emagic_Default_7_1 = 118
kAudioChannelLayoutTag_SMPTE_DTV = 119
```

ITU defined layouts:

```
kAudioChannelLayoutTag_ITU_1_0 = 100
kAudioChannelLayoutTag_ITU_2_0 = 101
kAudioChannelLayoutTag_ITU_2_1 = 120
kAudioChannelLayoutTag_ITU_2_2 = 121
kAudioChannelLayoutTag_ITU_3_0 = 102
kAudioChannelLayoutTag_ITU_3_1 = 104
kAudioChannelLayoutTag_ITU_3_2 = 106
kAudioChannelLayoutTag_ITU_3_2_1 = 110
kAudioChannelLayoutTag_ITU_3_4_1 = 117
```

DVD defined layouts:

```
kAudioChannelLayoutTag_DVD_0 = 100
kAudioChannelLayoutTag_DVD_1 = 101
kAudioChannelLayoutTag_DVD_2 = 120
kAudioChannelLayoutTag_DVD_3 = 121
kAudioChannelLayoutTag_DVD_4 = 122
kAudioChannelLayoutTag_DVD_5 = 123
kAudioChannelLayoutTag_DVD_6 = 124
kAudioChannelLayoutTag_DVD_7 = 102
kAudioChannelLayoutTag_DVD_8 = 104
kAudioChannelLayoutTag_DVD_9 = 106
kAudioChannelLayoutTag_DVD_10 = 125
kAudioChannelLayoutTag_DVD_11 = 126
kAudioChannelLayoutTag_DVD_12 = 110
kAudioChannelLayoutTag_DVD_13 = 104
kAudioChannelLayoutTag_DVD_14 = 106
kAudioChannelLayoutTag_DVD_15 = 125
kAudioChannelLayoutTag_DVD_16 = 126
kAudioChannelLayoutTag_DVD_17 = 110
kAudioChannelLayoutTag_DVD_18 = 127
kAudioChannelLayoutTag_DVD_19 = 107
kAudioChannelLayoutTag_DVD_20 = 111
```

Document Revision History

This table describes the changes to *Core Audio*.

Date	Notes
2008-10-15	This document has been replaced by *Core Audio Overview* and other documents in the ADC Reference Library.
2004-03-25	Second Preliminary Draft completed.
2003-08-25	Preliminary Draft completed for Panther.
2003-08-08	Beta Draft seeded internally for comments and corrections.
2003-06-19	Seed Draft completed for WWDC, based on input from the Core Audio engineering team. Document to be distributed privately at WWDC.

Document Revision History